Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

Desenvolvimento e implementação de blocos digitais para um transceptor UWB baseado em um RISC-V

Hércules Ismael de Abreu Santos

DISSERTAÇÃO DE MESTRADO

PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS MECATRÔNICOS

Brasília

2025

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

Desenvolvimento e implementação de blocos digitais para um transceptor UWB baseado em um RISC-V

Hércules Ismael de Abreu Santos

Dissertação de Mestrado submetida ao Departamento de Engenharia Mecânica da Universidade Brasília como parte dos requisitos necessários para a obtenção do grau de Mestre

Orientador: Prof. Dr. Jones Yudi Mori Alves da Silva
Coorientador: Prof. Dr. Gilmar Silva Beserra

Brasília
2025

Santos, Hércules Ismael de Abreu.
S769 Desenvolvimento e implementação de blocos digitais para um transceptor UWB baseado em um RISC-V / Hércules Ismael de Abreu Santos; orientador Jones Yudi Mori Alves da Silva; co-orientador Gilmar Silva Beserra. -- Brasília, 2025.
80 p.

Dissertação de Mestrado (Programa de Pós-Graduação em Sistemas Mecatrônicos) -- Universidade de Brasília, 2025.

1. RISC-V. 2. IEEE 802.15.6. 3. IoT. I. Silva, Jones Yudi Mori Alves da, orient. II. Beserra, Gilmar Silva, coorient. III. Título

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

Desenvolvimento e implementação de blocos digitais para um transceptor UWB baseado em um RISC-V

Hércules Ismael de Abreu Santos

Dissertação de Mestrado submetida ao Departamento de Engenharia Mecânica da Universidade Brasília como parte dos requisitos necessários para a obtenção do grau de Mestre

Trabalho aprovado. Brasília, 28 de março de 2025:

Prof. Dr. Jones Yudi Mori Alves da Silva, UnB/FT/ENM
Orientador

Prof. Dr. Gilmar Silva Beserra, UnB/FCTE
Co-Orientador

Prof. Dr. Leonardo Londero de Oliveira, UFSM/DELC
Examinador externo

Prof. Dr. Wellington Avelino do Amaral, UnB/FCTE
Examinador interno

Brasília
2025

Agradecimentos

Quero agradecer primeiramente a minha família por me dar suporte no decorrer do curso. Agradeço a meu pai, que já não está entre os vivos, mas representa para mim o maior exemplo de homem que tive, e agradeço a minha mãe por me dar suporte e estar comigo. Agradeço também as meus tios José Martins e Creuza Abreu por participarem de uma parte importante da minha criação.

Agradeço a meus professores pelas aulas instrutivas e agradeço também ao meu orientador Jones Yudi e meu coorientador Gilmar Beserra, por me orientarem, tirando dúvidas e fornecendo direções para realizar este trabalho.

“All models are wrong, but some are useful.”
(George Box)

Resumo

Sistemas IoT atuais requerem a integração de diversas funcionalidades como leitura de sensores, comunicação sem fio, processamento de dados e demandam realizar estas funções em baixo consumo de energia. Uma tecnologia promissora de comunicação sem fio que pode ser utilizada em aplicações IoT de baixa potência é o UWB, que permite a transmissão de dados em uma velocidade considerável com um consumo baixo de energia. Para implementar esta tecnologia, uma boa prática é seguir o padrão IEEE 802.15.6 para redes WBAN, que padroniza a implementação de sistemas UWB. Este trabalho tem o objetivo de implementar um sistema DSP para controlar um SoC que transmite dados via UWB. Para isto foi implementado a parte digital da camada física do UWB, prevista na norma IEEE 802.15.6. Foi implementado um filtro média móvel eficiente para realizar um pré-processamento de dados de sensores, com o intuito de reduzir ruídos. E finalmente estes blocos foram integrados com um processador RISC-V, que é uma opção promissora de processador para realizar o controle de um SoC como esse. Estes blocos foram implementados em tecnologia CMOS e comparados com circuitos semelhantes na literatura, demonstrando resultados promissores.

Palavras-chave: RISC-V. IEEE 802.15.6. IoT.

Abstract

Recent IoT systems require the integration of various functions like sensor reading, wireless communication, data processing and demand the execution of this tasks at a low power. A promising wireless communication technology for low power applications is the UWB, that allows for a considerable speed on the data transmission with low power. In order to implement this technology, a good practice is to follow the IEEE 802.15.6 standard for WBAN networks, that standardizes the implementation of UWB systems. This work has the goal of implementing a DSP system for controlling a SoC that transmits data via UWB. In order to do this, it was implemented the digital part of the physical layer of the UWB, as the standard IEEE 802.15.6 specifies. It was implemented an efficient simple moving average filter for the pre-processing of sensor data, with the goal of reducing noise on the data acquisition. And finally, these blocks were integrated with a RISC-V processor, which is a promising processor for controlling of a SoC like this. These blocks were implemented using CMOS and were compared with similar circuits available on literature, showing primising results.

Keywords: RISC-V. IEEE 802.15.6. IoT.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Internet das Coisas (*IoT - Internet of Things*) é uma rede de dispositivos equipados com sensores e sistemas de comunicação que permitem a transmissão e recepção de dados. Um exemplo é a tecnologia emergente conhecida como WBAN (*Wireless Body Area Network*), que consiste em um conjunto de nós que possibilita a aquisição contínua de sinais fisiológicos por meio de biossensores que podem ser anexados ou até mesmo implantados no corpo. Os dados coletados podem ser transmitidos para médicos ou armazenados em bancos de dados. Assim, é possível realizar o monitoramento de pacientes em tempo real, o que facilita a realização de diagnósticos e intervenções médicas quando necessário.

Nesse contexto, encontra-se em desenvolvimento na Universidade de Brasília um SoC (*System-on-Chip*) baseado na arquitetura RISC-V com o objetivo de monitorar e processar sinais biológicos a partir de sensores, e realizar a transmissão e recepção de dados por meio da tecnologia UWB (*Ultra-Wideband*). Essa tecnologia oferece vantagens relevantes para este tipo de aplicação por apresentar um consumo baixo de energia, já que o envio de dados é feito por pulsos curtos de transmissão, e por utilizar uma banda larga, o que possibilita uma taxa alta de transmissão de dados. Além disso, essa tecnologia não necessita de uma portadora para a transmissão de dados, operando assim em banda base.

A arquitetura do SoC, mostrada na Figura 1, é adaptável e permite uma integração perfeita em aplicações WBAN, atuando como nó em uma rede de sensores sem fio. A antena UHF (*Ultra High Frequency*) alimenta o sistema de gerenciamento de energia, que recebe o sinal e o converte em uma tensão apropriada para o sistema, além de armazenar a energia na bateria. A antena UWB recebe e transmite dados, sendo estas funções selecionadas pelo *RF Switch* (comutador RF). O SoC recebe dados de um sensor, que é seguido de um conversor analógico para digital que enviará os dados convertidos para um DSP (*Digital Signal Processor*). Entre o DSP e o ADC (*Analog to Digital Converter*) há um bloco SMA (*Simple Moving Average*), que consiste em um filtro para retirar ruídos dos sinais recebidos do sensor, antes serem processados no DSP.

O uso de um DSP permite processar e selecionar os dados recebidos do sensor de tal forma que não seja necessário transmitir uma grande quantidade de informação. Isso promove uma maior eficiência energética, pois a transmissão de dados em sistemas IoT é uma grande fonte de consumo energético. Desse modo, processar os dados antes de serem transmitidos pode ser uma oportunidade para promover uma maior eficiência energética.

O bloco DSP será composto por uma microarquitetura especificamente projetada para tarefas de processamento de sinais digitais, garantindo ao mesmo tempo eficiência energética. A ISA (*Instruction Set Architecture*) RISC-V foi escolhida porque é composta por um conjunto

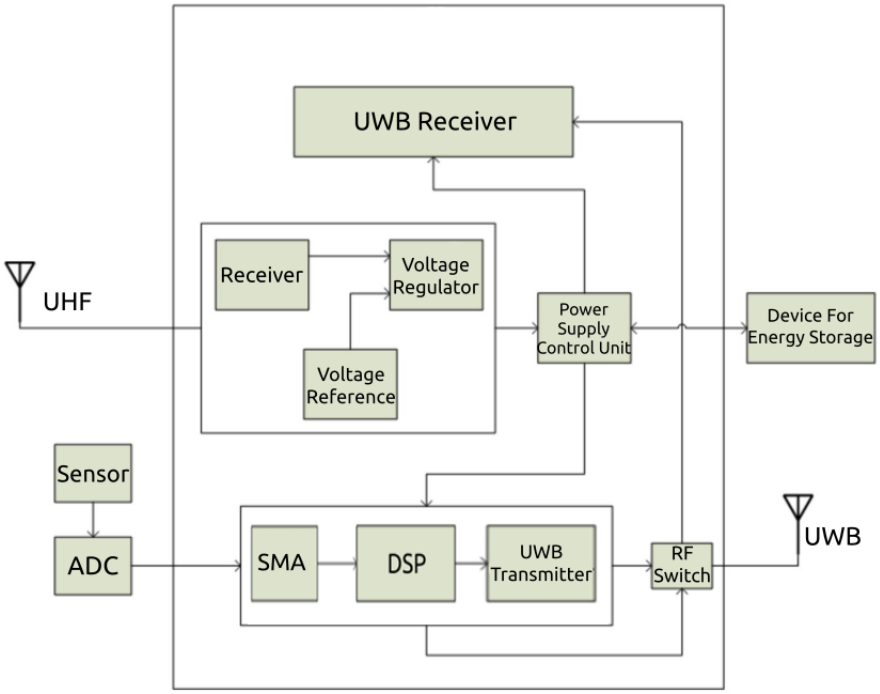


Figura 1 – Diagrama de blocos da arquitetura do SoC.

básico de instruções que pode ser adaptado e otimizado para as necessidades específicas da aplicação. Uma implementação adequada é o PicoRV32, um núcleo de microcontrolador simples que permite diferentes configurações e é aberto sob a licença ISC. Esse bloco também será responsável pela implementação da camada física do transceptor UWB, que enquadra a mensagem a ser transmitida em um quadro de comunicação adequado para a tecnologia. É na camada física que são implementadas as operações que inserem bits de verificação na mensagem e permitem a correção de erros, bem como as operações de embaralhamento da mensagem que promovem segurança e uma distribuição melhor do espectro potência do sinal.

1.1 Objetivos Geral e Específicos

O objetivo geral deste trabalho é desenvolver e implementar em ASIC dois blocos digitais para o SoC: o filtro média móvel e a camada física do transceptor UWB conforme a norma IEEE 802.15.6. Para isso, foram listados os seguintes objetivos específicos:

a) Especificação de um filtro média móvel para filtrar o sinal de um sensor.

b) Especificação de um bloco dedicado que implementa a camada física do transceptor UWB conforme a norma IEEE 802.15.6.

c) Codificação dos blocos em *Verilog* e validação por meio de simulações.

d) Integração dos blocos com o *PicoRV32* e validação por meio de simulações.

e) Execução do fluxo digital para sínteses lógica e física, com geração e verificação dos layouts.

f) Caracterização dos circuitos a partir de dados de consumo energético, área e atraso.

1.2 Contribuições do Trabalho

Dentre as contribuições deste trabalho, destacam-se as listadas abaixo:

a) Circuito do filtro média móvel enviado para *tape-out* e fabricado na tecnologia CMOS UMC 180nm, com artigo publicado em conferência (SANTOS, 2023).

b) Proposta de uma arquitetura para o processamento em banda-base que utiliza aritmética em corpos finitos.

c) Integração dos blocos ao PicoRV32.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte forma: o capítulo 2 contém o estado da arte e a fundamentação teórica, na qual são explicados todos os fundamentos importantes para a compreensão deste trabalho; no capítulo 3, o fluxo de desenvolvimento utilizado neste trabalho foi detalhado; no capítulo 4, foram apresentadas as propostas de circuito para cada um dos blocos; no capítulo 5, os resultados da implementação dos circuito propostos foram discutidos, sendo apresentados dados de consumo de área, potência e desempenho; por fim, as conclusões foram apresentadas no capítulo 6.

2 Fundamentação teórica

2.1 Filtro FIR e Média Móvel Simples

Um filtro de resposta finita ao impulso (FIR) é um filtro baseado no modelo FIR que é um modelo com entradas exógenas, que pode ser expressado pela equação 2.1 (BILLINGS, 2013):

$$y(k) = b_1 u(k-1) + b_2 u(k-2) + \cdots + b_n u(k-n) \tag{2.1}$$

onde u representa a entrada do filtro, e b_i são os parâmentros do filtro. Esse tipo de filtro funciona de forma a realizar uma operação de soma ponderada de n amostras da entrada, funcionando efetivamente como um filtro passa-baixas. Desta forma, ele atenua componentes de alta frequência no sinal de entrada e tem diversas aplicações em várias áreas (ALJUFFRI et al., 2015) (ANNANGI; PULI, 2017) (CHAUHAN et al., 2018) (MORALES-MENDOZA et al., 2008).

Um filtro Média Móvel Simples (SMA) é um caso particular de filtros FIR em que todos os parâmetros são iguais e a soma destes é igual a 1. A equação 2.2 mostra o modelo matemático deste filtro.

$$y(k) = \frac{u(k-1) + u(k-2) + u(k-3) + \cdots + u(k-n)}{n} \tag{2.2}$$

Embora filtros de média móvel simples não tenham a versatilidade de um filtro FIR, ainda sim, eles tem alto valor prático em diversas aplicações, como mostra (MUÑOZ et al., 2017; PASTRANA et al., 2022). Um exemplo do uso de filtros de média móvel simples é o uso em sensores ruidosos para obter leituras mais estáveis. Em um estudo conduzido por (MAGSI et al., 2018), vários filtros são analisados e a efetividade da redução de ruído é avaliada através de simulações. Os resultados indicam que o filtro é capaz de reduzir até 95% do ruído. Esse resultado sugere que filtros de média móvel simples tem uma boa performance na redução de ruídos de alta frequência, de modo que esta pode ser uma ferramenta relevante para a redução de ruídos.

2.2 Padrão IEEE 802.15.6

O padrão IEEE 802.15.6, é uma norma que foi criada com o objetivo de padronizar a comunicação sem fio de aparelhos que se enquadram na categoria de redes BAN (*Body Area Network*). Essa categoria é voltada a aparelhos vestíveis sendo em alguns casos equipamentos

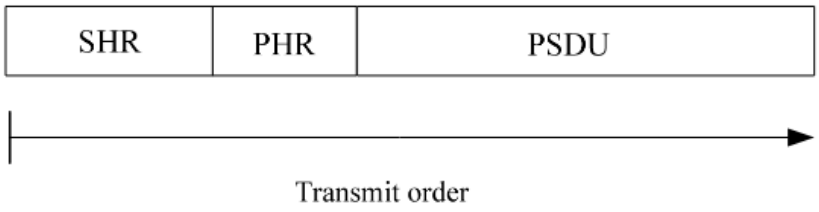


Figura 2 – Estrutura da mensagem do UWB.

com aplicações médicas. Nesta categoria a comunicação ocorre em curto alcance, ocorrendo dentro de um corpo humano (não limitado a humanos), ou na vizinhança próxima deste.

Dado esta aplicação, este padrão oferece formas de comunicação que utiliza bandas de frequência que sejam aplicáveis para aplicações médicas. Também permite que os aparelhos comuniquem em baixa potência, de modo a otimizar o consumo de bateria e minimizar a energia eletromagnética que pode ser absorvida pelo corpo humano. E também oferece segurança na transmissão dos dados, de modo que se proteja comunicações que carreguem dados sensíveis.

Este padrão define a camada física, e a subcamada de MAC (*Medium Access Control*), que é complementar, e serve para construir os quadros de comunicação a serem enviados pela camada física. Também neste padrão são especificados 3 tipos de camada física (PHY), o *Narrowband* (NB) , o *Ultra Wide Band* (UWB) e o *Human body Communications* (HBC).

Para o presente trabalho, iremos focar na camada física do UWB, de modo que a camada de MAC fica fora do escopo deste projeto, assim como o NB e o HBC. Além disso focaremos nas etapas de processamento digital que precedem os circuitos analógico e de RF responsáveis pela transmissão das mensagens.

Para transmitir uma mensagem por UWB, a norma define uma estrutura padrão que deve ser seguida, sendo que cada uma das partes da estrutura terá particularidades na implementação. Na figura 2 as 3 partes principais de uma mensagem UWB estão representadas.

O SHR (*Sinchronization Header*) é o cabeçalho de sincronização. Neste cabeçalho, uma sequência de bits é enviada repetidas vezes, e tem o objetivo de sinalizar que uma mensagem está chegando e oferecer uma forma de que o circuito possa se sincronizar com a transmissão.

O PHR (*Physical layer Header*) é o cabeçalho de camada física. Esse cabeçalho traz informações sobre a mensagem que está sendo enviada, como o tamanho do corpo da mensagem e a taxa de transmissão dos dados.

O PSDU (*Physical layer Service Data Unit*) é o corpo da mensagem. Nessa seção da mensagem, é incluso o quadro de comunicação que vem do MAC, que é chamado de MPDU. Esse quadro contém o corpo da mensagem, um cabeçalho do MAC, e bits de paridade.

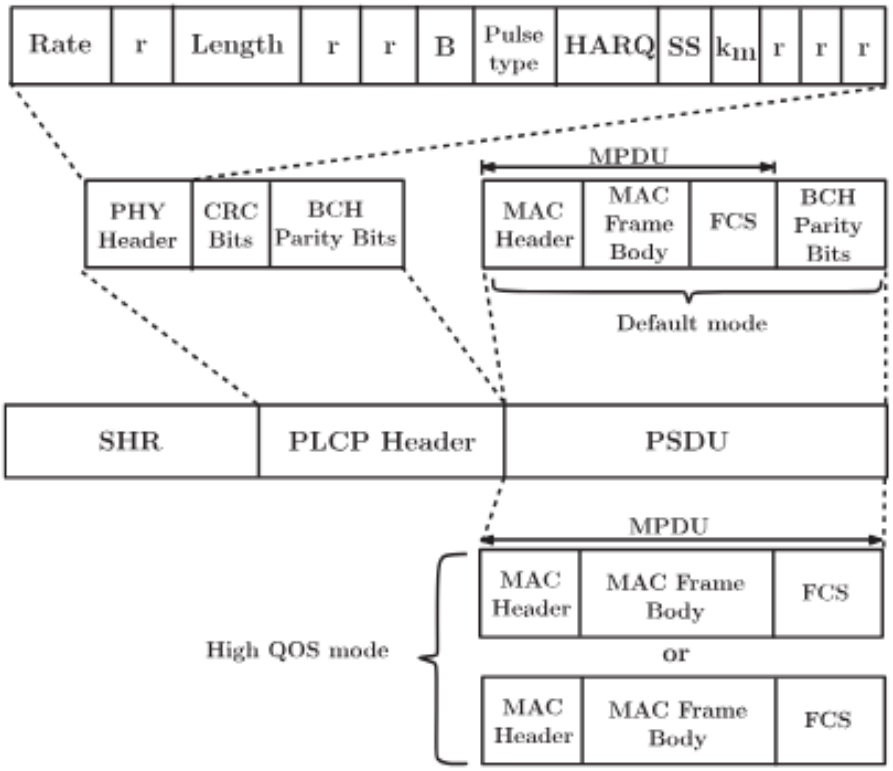


Figura 3 – Estrutura da mensagem expandida. (MANCHI; PAILY; GOGOI, 2017)

Na figura 3 uma visão expandida da mensagem é apresentada. Na imagem é possível ver o PHR (Na imagem PLCP Header), que é construído do cabeçalho propriamente dito, mais bits de CRC e bits de paridade BCH. Estes bits servem como uma forma de checar a integridade da mensagem. Sendo que com os bits de CRC é possível checar se a mensagem do cabeçalho está íntegra e o BCH também tem essa função, porém a partir do BCH, não só é possível verificar erros na mensagem como pode-se corrigir a mensagem automaticamente, até uma quantidade máxima de bits trocados. Mais detalhes sobre ambos o CRC e BCH serão apresentados em seções posteriores.

Também na figura 3 é possível ver a estrutura do PHR. Ele contém 24 bits, sendo os 3 primeiros apresentados na imagem como *"Rate"*, estes são os bits reservados para especificar a taxa de transmissão da mensagem. Todos os espaços preenchidos com a letra "r"são espaços reservados e não carregam informação a priori. O espaço *"Length"* se refere a 8 bits reservados para definir o tamanho do corpo da mensagem bytes, podendo variar entre 0 bytes até 255 bytes. O "B"é o *"burst mode"* que é um bit usado para definir se a mensagem está sendo enviada em modo *burst* ou não. Esse modo permite que se envie vários pacotes consecutivos sem precisar que o receptor envie uma mensagem de resposta, confirmando o correto recebimento da mensagem. *"Pulse type"* é um bit que define qual tipo de pulso deve ser usado na transmissão. O HARQ são dois bits que controlam o estado do HARQ, que é um fluxo de retransmissão. *"SS"* é uma semente para o *scrambler*, que é um bloco que será detalhado mais adiante. E finalmente o *"km"* define qual será a constelação utilizada.

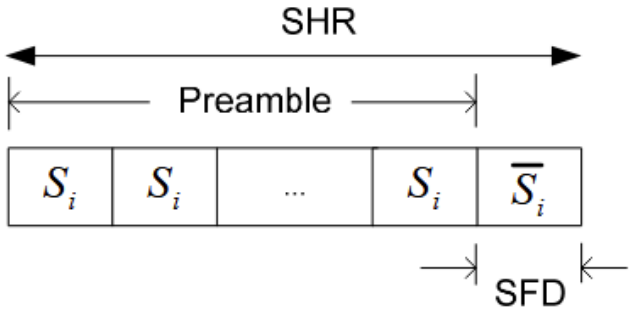


Figura 4 – SHR header. (IEEE..., 2012)

C_1	1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 1 1 1 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0
C_2	0 0 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1 1 0 0 1 1 0 0 1 0 1 0 1 1 1 0 0 0 1 1 0 1 0 1 0 1 0 0 1 0
C_3	1 0 0 0 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 0 1
C_4	0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1 1 1 1 1 0 0
C_5	1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 1 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 0
C_6	1 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1
C_7	0 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1
C_8	0 0 1 1 0 1 1 0 1 1 0 0 1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 1 1 1 1 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 1

Figura 5 – Sequência kasami. (IEEE..., 2012)

Para o PSDU pode-se ver na figura 3 a divisão do PSDU em um MPDU e os bits de paridade. O MPDU contém o cabeçalho MAC, tem o corpo do quadro do MAC, e contém o FCS(Frame Check Sequence) que são bits para verificar a integridade do quadro. O MPDU deve vir já formatado do MAC, com o formato adequado, para que seja transmitido pela camada física. O papel da camada física então é adicionar os bits de paridade BCH, que semelhante ao caso do PHR, servem para verificar a integridade da mensagem e permitem a correção automática da mensagem, dentro de um limite de bits trocados.

A seguir um pouco mais de detalhes sobre cada uma das seções da mensagem serão apresentados.

2.2.1 SHR

O SHR é dividido em 2 partes, o preâmbulo e o SFD (*Start-of-frame delimiter*). O preâmbulo serve para sincronização temporal e detecção de pacotes. A segunda parte serve para sincronizar quadro e delimita o começo deste.

Para construir o preâmbulo são utilizadas sequências Kasami de tamanho 63. Há 8 sequências que são definidas na figura 5

O preâmbulo deve consistir em 4 repetições do símbolo S_i, sendo que o símbolo S_i é a sequência Kasami, porém com uma quantidade de bits '0' entre cada um dos bits da sequência. Então envia-se um bit, depois envia-se zeros, depois envia-se outro bit, envia-se outra quantidade de zeros, até acabar a sequência. A imagem 6 ilustra esta operação.

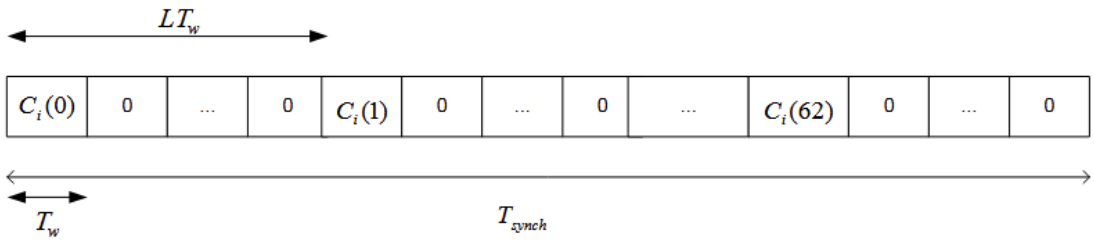


Figura 6 – Construção do S_i. (IEEE..., 2012)

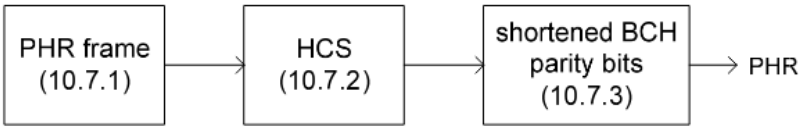


Figura 7 – Construção do PHR. (IEEE..., 2012)

Por fim o SFD consiste no envio de um símbolo S_i invertido, ou seja, com os bits da sequência Kasami invertidos. Os zeros que são inseridos entre os elementos da sequência Kasami não são invertidos, permanecendo com o valor zero, e apenas a sequência tem seus bits invertidos.

2.2.2 PHR

Para construir o PHR é necessário o PHR propriamente dito, mais o HCS (*Header Check Sequence*) e os bits de paridade do BCH.

O HCS é uma sequência de 4 bits de uma código de detecção de erros CRC-4 ITU. O CRC-4-ITH debe ser o complemento do resto da divisão em módulo-2 da informação do PHR pelo polinômio:

$$1 + x + x^4 \tag{2.3}$$

Esta operação pode ser implementada por um registrador de deslocamento, conforme arquitetura sugerida no documento da norma IEEE 802.15.6.

Após acrescentar os bits de CRC-4 ITU, 12 bits devem ser adicionados de um código BCH(40,28) encurtado.

2.2.3 PSDU

O PSDU é construído a partir do MPDU, que vem do MAC. O MPDU deve passar por um *scrambler*, um *encoder* BCH e um *interleaver*. A etapa *"Pad Bits"* é necessária apenas para algumas constelações específicas, para o caso deste projeto, essa etapa não será necessária.

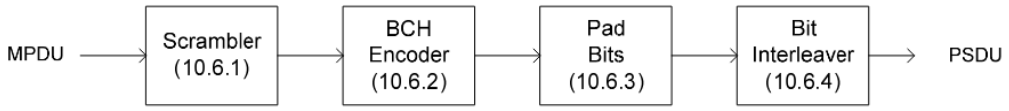


Figura 8 – Construção do PSDU. (IEEE..., 2012)

Scrambler seed (SS)	Initialization vector $x_{init} = x[-1]\ x[-2]\ \ldots\ x[-14]$
0	0 0 1 0 1 1 1 1 0 0 1 1 0 1
1	0 0 0 0 0 0 0 1 0 0 1 1 1 1

Figura 9 – Seleção da semente do scrambler. (IEEE..., 2012)

O *scrambler* deve ser aplicado para eliminar longas sequências de 0s e 1s contidas no MPDU, de forma que o espectro de potência do sinal não tenha dependência nos dados reais.

O *scrambler* deve ser um *scrambler* aditivo, com um polinômio gerador dado pela equação:

$$x[n] = x[n-2] \oplus x[n-12] \oplus x[n-13] \oplus x[n-14] \tag{2.4}$$

Tipicamente esse tipo de *scrambler* é implementado por uma LFSR (*Linear Feedback Shift Register*).

Os bits do *scrambler* devem ser iniciados com uma semente. Essa semente é definida pela opção SS do PHR, e pode selecionar duas opções, conforme a figura 9.

O codificador BCH a ser implementado, deve ser o BCH(n=63,k=51) para o modo padrão. Para implementar este codificador deve-se utilizar o polinômio gerador:

$$g(x) = 1 + x^3 + x^4 + x^5 + x^8 + x^10 + x^12 \tag{2.5}$$

Sendo que os bits de paridade devem ser determinados calculando-se o resto do polinômio r(x):

$$r(x) = \sum_{i=0}^{11} r_i x^i = x^{12} m(x) \mod g(x) \tag{2.6}$$

Onde m(x) é o polinômio da mensagem, que é definido da seguinte forma:

$$m(x) = \sum_{i=0}^{50} m_i x^i \tag{2.7}$$

Nas próximas seções, a teoria do BCH será discutida de maneira mais profunda.

Finalmente o *interleaver* deve ser aplicado antes da modulação para obter robustez contra a propagação de erros. O *interleaver* embaralha os bits da mensagem, mapeando blocos da mensagem em ordens diferentes. Isso deve ser feito conforme a equação:

$$\Pi(n) = nb_s \mod N_I \tag{2.8}$$

Onde N_I é o tamanho do *interleaver*, $\Pi(n)$ é a nova posição para a qual o índice n deve ser permutado.

O tamanho do *interleaver* deve ser $N_I = 192$, e $b_s = 37$.

O *interleaver* é aplicado em blocos de tamanho 192, porém, no último bloco, se a quantidade de bits restante for menor que 192, então N_I deve ser determinado como a quantidade de bits restantes.

2.3 Corpos Finitos(Corpos de Galois)

Corpos finitos ou corpos de Galois é um conceito matemático que define um conjunto finito de números, em que se pode fazer operações dentro deste conjunto. Este conceito é utilizado em códigos de correção de erros, como os códigos Bose-Chaudhuri-Hocquenghem (BCH), o Reed-Solomon (RS) e o *low-density parity-check* (LDPC), além de ser utilizado em esquemas de criptografia como o *Advanced Encryption Standard* (AES). Operações em corpos finitos tem propriedades únicas que podem ser exploradas para simplificar estruturas de hardware, e pode tornar atrativo a implementação de sistemas utilizando corpos finitos.

Para entender o conceito de corpos finitos, algumas definições devem ser feitas.

2.3.1 Corpo

Um corpo, em álgebra abstrata, é um conjunto de elementos com 2 operações, a operação de adição e a operação de multiplicação, que são denotados pelos sinais + e · respectivamente. Essas operações devem seguir as seguintes condições:

- Os elementos do conjunto devem formar um grupo comutativo em relação a + e deve ter um elemento identidade, que é chamado zero do corpo, e é denotado por 0.

- Os elementos do conjunto que não são iguais a zero, devem formar um grupo comutativo em relação a · e deve ter um elemento identidade em relação a · que é chamado de a identidade do corpo, e é denotado por 1.

- A operação de multiplicação é distributiva em relação a operação de adição:

$$a \cdot (b + c) = a \cdot b + a \cdot c \tag{2.9}$$

Um corpo finito então, é um corpo formado por um conjunto finito de elementos. O número de elementos em um corpo finito pode ser um número primo ou uma potência de um primo. Então para representar um corpo finito de p^q elementos, sendo p um número primo, escreve-se $GF(p^q)$. O número de elementos de um corpo também é chamado de ordem do corpo.

2.3.2 Ordem de um elemento

Dado um campo finito $GF(p^q)$, a ordem de um elemento $\alpha \in GF(p^q)$ é o menor inteiro r, de tal modo que α^r é igual ao elemento identidade do campo. A ordem de $\alpha \in GF(p^q)$ sempre divide $p^q - 1$, deste modo α^{p^q-1} é sempre igual ao elemento identidade.

2.3.3 Elemento primitivo

Definido a ordem de um elemento é possível definir um elemento primitivo. Este é definido como o elemento $\alpha \in GF(p^q)$ que tem ordem igual a $p^q - 1$

2.3.4 Polinômio Irredutível

Um polinômio irredutível é uma polinômio que não pode ser fatorado em polinômios de graus menores.

Em campos finitos, um polinômio que tem coeficientes que são elementos de um campo $GF(p^q)$ é dito ser um polinômio sobre $GF(p^q)$.

2.3.5 Polinômio Primitivo

Um polinômio irredutível f(x) sobre $GF(p^q)$ de grau q é dito ser um polinômio primitivo se o p menor inteiro s, para o qual f(x) divide $x^s - 1$, for $p^q - 1$.

Todo elemento que que é raiz de um polinômio primitivo de grau q sobre $GF(p)$ é um elemento primitivo de $GF(p^q)$.

2.3.6 Polinômio Mínimo

Seja $\alpha \in GF(p^q)$. O polinômio mínimo de α em relação a $GF(p^q)$ é o polinômio de menor grau m(x) sobre $GF(p^q)$ que satisfaz $m(\alpha) = 0$.

Polinômios mínimos são irredutíveis.

Para cada elemento $\alpha in GF(p^q)$ existe um polinômio mínimo sobre $GF(p)$ que tem o coeficiente de termo de maior grau do polinômio igual a identidade.

2.3.7 Base canônica

Uma base canônica de um campo finito $GF(2^q)$ é definida como o conjunto $\{1, \alpha, \alpha^2, ..., \alpha^{q-1}\}$, onde α é a raiz de um polinômio irredutível de grau q sobre $GF(2)$. Esta é chamada de uma base canônica de $GF(2^q)$ sobre $GF(2)$.

Usando uma base canônica é possível representar um campo finito utilizando polinômios. Essa representação traz algumas vantagens quando se faz operações dentro de um corpo $GF(2^q)$. A soma, para um $GF(2)$ é definida como um soma em módulo 2, que é equivalente a uma operação de XOR. Isso significa que a soma, utilizando uma representação em base canônica de um $GF(2^q)$, é equivalente a fazer uma operação XOR entre os coeficientes dos polinômios a serem somados.

Exemplo: $\{1, \alpha, \alpha^2, \alpha^3\}$ é uma base canônica de $GF2^4$, sendo α a raiz do polinômio primitivo $x^4 + x + 1$.

2.4 Aritmética em Corpos Finitos

A aritmética em corpos finitos pode ser feita de maneiras diferentes, a depender da representação utilizada para os corpos finitos. Além da representação usando base canônica, existem representações em potência, base normal, dual. Cada uma dessas representações fornece formas diferentes de se fazer aritmética. Neste trabalho, iremos focar em operações utilizando a base canônica, em corpos $GF(2^q)$.

2.4.1 Soma

A soma em corpos finitos, em particular para corpos do tipo $GF(2^q)$, pode ser feita utilizando uma operação XOR bit a bit. Considerando a representação em base canônica, é possível representar cada coeficiente do polinômio, como um bit, e fazer uma operação XOR em cada bit.

Por exemplo, considere $\{1, \alpha, \alpha^2, \alpha^3\}$ uma base canônica de $GF(2^4)$, sendo α a raiz do polinômio primitivo $x^4 + x + 1$. A soma de um elemento $\alpha^2 + 1$ com um elemento $\alpha^3 + \alpha^2 + \alpha$, pode ser feita da forma:

$$(\alpha^2 + 1) + (\alpha^3 + \alpha^2 + \alpha) => 0101 + 1110 \tag{2.10}$$

$$\alpha^3 + (\alpha^2 \oplus \alpha^2) + \alpha + 1 => 0101 \oplus 1110 \tag{2.11}$$

$$\alpha^3 + \alpha + 1 => 1010 \tag{2.12}$$

2.4.2 Subtração

A subtração em um corpo $GF(2^q)$ tem uma particularidade. Ela é exatamente igual a uma soma. Em outras palavras, para realizar uma subtração, basta realizar uma operação XOR bit a bit, exatamente como no caso da soma.

Por exemplo, realizando uma subtração com os elementos do exemplo do caso da soma resulta na mesma coisa que uma soma:

$$(\alpha^2 + 1) - (\alpha^3 + \alpha^2 + \alpha) => 0101 - 1110 \tag{2.13}$$

$$\alpha^3 + (\alpha^2 \oplus \alpha^2) + \alpha + 1 => 0101 \oplus 1110 \tag{2.14}$$

$$\alpha^3 + \alpha + 1 => 1010 \tag{2.15}$$

2.4.3 Multiplicação

A multiplicação para um corpo finito, utilizando um base canônica, é feita como uma multiplicação de polinômios com módulo com um polinômio irredutível p(x). Para entender o que isso quer dizer, podemos começar com um exemplo.

Considere um campo $GF(2^6)$, representado pela base canônica $\{1,\alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5\}$, sendo α raiz do polinômio mínimo $\alpha^6 + \alpha + 1$. Considere também $a = \alpha^5 + \alpha^4$, e $b = \alpha^1 + 1$. A multiplicação $a \cdot b$ será:

$$a \cdot b = (\alpha^5 + \alpha^4) \cdot (\alpha^1 + 1) \tag{2.16}$$

$$a \cdot b = \alpha^6 + (\alpha^5 + \alpha^5) + \alpha^4 = \alpha^6 + \alpha^4 \tag{2.17}$$

Para representar em base canônica, o maior grau de α que pode-se usar neste exemplo é 5, deste modo é necessário simplificar α^6. Para isso consideramos a definição do polinômio mínimo:

$$\alpha^6 + \alpha + 1 \tag{2.18}$$

De onde pode-se obter:

$$\alpha^6 = \alpha + 1 \tag{2.19}$$

Deste modo a resposta da multiplicação pode ser obtida como:

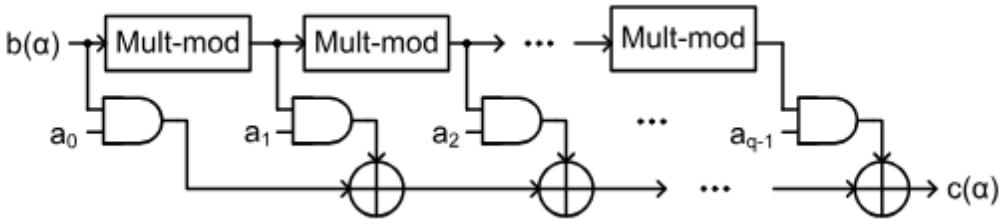


Figura 10 – Arquitetura de um multiplicador em base canônica. (ZHANG, 2016)

$$a \cdot b = \alpha^4 + \alpha + 1 \tag{2.20}$$

Esta é a forma analítica de se obter o resultado de uma multiplicação usando uma base canônica. O passo de reduzir α^6 para $\alpha + 1$, pode ser entendido como a forma analítica de realizar a operação de módulo com o polinômio mínimo.

A multiplicação em base canônica, pode ser definida também da seguinte forma:

Considere um corpo $GF(2^q)$ com base canônica $\{1,\alpha, \alpha^2, ..., \alpha^{q-1}\}$, construído com um polinômio irredutível p(x) de grau q. Sejam $a(\alpha) = a_0 + a_1\alpha + ... + a_{q-1}\alpha^{q-1}$ e $b(\alpha) = b_0 + b_1\alpha + ... + b_{q-1}\alpha^{q-1}$ $(a_i, b_i \in GF(2))$, dois elementos de $GF(2^q)$. O produto de $a(\alpha)$ e $b(\alpha)$ é:

$$c(\alpha) = a(\alpha)b(\alpha) \mod p(\alpha) \tag{2.21}$$

Essa operação pode ser reescrita como:

$$c(\alpha) = a_0 b(\alpha) \mod p(\alpha) + a_1(b(\alpha)\alpha) \mod p(\alpha) + ... + a_{q-1}(b(\alpha)\alpha^{q-1} \mod p(\alpha) \tag{2.22}$$

Os termos $b(\alpha)\alpha^i \mod p(\alpha)$ podem ser calculados de forma iterativa, como $(b(\alpha)\alpha^{i-1} \mod p(\alpha))\alpha$. Esse cálculo iterativo pode ser utilizado para criar uma estrutura em hardware capaz de fazer a multiplicação, como na imagem 10. Com um módulo "Mult-mod"que faz a operação $b(\alpha) \cdot \alpha$, é possível realizar a multiplicação. Para realizar um módulo desse tipo, é possível derivar uma arquitetura, de forma analítica, baseado no corpo finito em que se quer fazer a multiplicação. A seguir um exemplo desta arquitetura será explorado, que também é o módulo que será utilizado posteriormente nas arquiteturas propostas deste trabalho.

Exemplo: Considere um campo $GF(2^6)$, representado pela base canônica $\{1,\alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5\}$, sendo α raiz do polinômio mínimo $p(\alpha) = \alpha^6 + \alpha + 1$. A multiplicação $b(\alpha) \mod p()$ é:

$$b(\alpha)\alpha \mod p(\alpha) = (b_5\alpha^5 + b_4\alpha^4 + b_3\alpha^3 + b_2\alpha^2 + b_1\alpha + b_0)\alpha \tag{2.23}$$

$$b(\alpha)\alpha \mod p(\alpha) = b_5\alpha^6 + b_4\alpha^5 + b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + b_0\alpha \tag{2.24}$$

$$b(\alpha)\alpha \mod p(\alpha) = b_5(\alpha + 1) + b_4\alpha^5 + b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + b_0\alpha \tag{2.25}$$

$$b(\alpha)\alpha \mod p(\alpha) = b_4\alpha^5 + b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + (b_0 \oplus b_5)\alpha + b_5 \tag{2.26}$$

Deste modo pode-se implementar o "mult-mod"utilizando apenas uma porta XOR.

2.5 BCH

Códigos Bose-Chaudhuri-Hocquenghem (BCH) geralmente são usados em aplicações que que tem erros aleatórios em bits, e precisam de codificadores de baixa complexidade. Exemplos de aplicação são comunicações óticas, transmissão de video digital (DVB) e memórias FLASH. Tais códigos se baseiam completamente nos conceitos de campos finitos, e usam estes conceitos como alicerce para a construção de um algoritmo de correção de erros.

Um código BCH(n,k) codifica uma mensagem de k bits em palavras de n bits. Para códigos construídos sobre $GF(2^q)$, n pode ser máximo $2^q - 1$, sendo que quando $n = 2^q - 1$, o código é chamado de código primitivo.

Considere m(x), a mensagem a ser transmitida, e c(x) a palavra codificada, c(x) então é definido como:

$$c(x) = m(x)g(x) \tag{2.27}$$

Sendo que g(x) é o polinômio gerador. O polinômio m(x), é construído utilizando os bits da mensagem como coeficientes do polinômio, e o código gerado também tem como resultado os coeficientes de c(x). A seguir será apresentado um exemplo para ilustrar.

Exemplo: Considere um código BCH(15, 7), construído sobre $GF(2^4)$, e o polinômio gerador $g(x) = x^8 + x^7 + x^6 + x^4 + 1$, e considere a mensagem m=0001011. Podemos representar m(x) como:

$$m(x) = x^3 + x + 1 \tag{2.28}$$

Deste modo, para calcular c(x), se faz:

$$c(x) = (x^3 + x + 1) \cdot (x^8 + x^7 + x^6 + x^4 + 1) \tag{2.29}$$

$$c(x) = (x^1 1 + x^1 0 + x^9 + x^7 + x^3) + (x^9 + x^8 + x^7 + x^5 + x) + (x^8 + x^7 + x^6 + x^4 + 1) \quad (2.30)$$

$$c(x) = x^{11} + x^{10} + x^7 + x^6 + x^5 + x^3 + x + 1 \quad (2.31)$$

De modo que c(x) pode ser representado por 0000110011101011.

Uma forma alternativa de se realizar a codificação em BCH, é a codificação sistemática, e ela é realizada conforme a expressão:

$$c(x) = m(x)x^{n-k} + (m(x)x^{n-k})_{g(x)} \quad (2.32)$$

Sendo que $(\cdot)_{g(x)}$ nesta expressão representa a operação de resto polinomial da divisão por g(x). Essa operação funciona, como uma codificação BCH, porque, assim como na equação 2.27, nessa definição, c(x) é divisível por g(x), e isso é suficiente para realizar a decodificação. A codificação sistemática é preferível, pois nesta, a mensagem original permanece intacta nos primeiros bits da palavra codificada, e apenas são acrescentados os bits do código em seguida a mensagem.

A ideia geral do BCH pode ser sintetizada da seguinte forma. O polinômio g(x) é construído de forma que ele tenha raízes consecutivas $(\alpha, \alpha^2, ..., \alpha^(2t))$. Isso significa que a mensagem codificada c(x), também tem raízes nos mesmos elementos. Se a mensagem codificada c(x) tem algum de seus bits alterados, é possível verificar que, o resultado de $c(\alpha^i)$ não igualaria a zero, mas traria um resultado diferente de zero. E a partir desses resultados, é possível aplicar algoritmos com os quais é possível identificar em qual bit ocorreu o erro, e assim corrigi-lo, quando a quantidade de erros é menor ou igual a t.

2.5.1 Decodificação BCH

A decodificação do BCH pode ser feita seguindo vários métodos diferentes. Pode-se citar os decodificadores do tipo *Hard-Decision*, os decodificadores do tipo *Chase BCH*, e os decodificadores do tipo *Interpolation-based Chase*. Os decodificadores do tipo *Hard-Decision* são os mais simples, e mais indicados para códigos BCH menores, com menor complexidade e é justamente essa categoria que focaremos neste trabalho.

Mesmo nos decodificadores *Hard-Decision*, existem diversas variações que podem ser exploradas, mas este tipo de decodificador segue uma estrutura com 3 passos: o cálculo de síndromes, *Key Equation Solver* (KES) e o *Chien Search*. A partir destes 3 passos é possível decodificar um código do tipo BCH.

2.5.2 Síndromes

Uma palavra c(x) é um código BCH, se e somente se ela tiver raiz nos mesmos elementos que são raiz de g(x), que é o polinômio gerador. Deste modo, uma palavra recebida, pode ser representada por:

$$r(x) = c(x) + e(x) \tag{2.33}$$

sendo c(x) o código original, e(x) o polinômio de erro e r(x) é o código recebido. Uma síndrome é definida como:

$$S_j(\alpha^j) = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) = e(\alpha^j) = \sum_{i=0}^{n-1} e_i(a^j)^i, 1 \leq j \leq 2t \tag{2.34}$$

Assumindo que r tem v erros nas posições i_1, i_2, ..., i_v, e assumindo $X_l = \alpha^{i_l}$ e considerando-se também que no caso do BCH, e_i pode assumir apenas os valores 0 e 1, pode-se escrever:

$$S_j = \sum_{l=1}^{v} (\alpha^j)^{i_l} = \sum_{l=1}^{v} X_l^j, 1 \leq j \leq 2t \tag{2.35}$$

Neste caso, X_l representa a localização dos erros, e portanto é chamado de localizador de erro.

A partir destas síndromes é possível chegar em um sistema de equações com 2t equações e v variáveis.

$$\begin{aligned} S_1 &= X_1 + X_2 + ... + X_v \\ S_2 &= X_1^2 + X_2^2 + ... + X_v^2 \\ &... \\ S_{2t} &= X_1^{2t} + X_2^{2t} + ... + X_v^{2t} \end{aligned} \tag{2.36}$$

Este sistema de equações torna possível a localização dos erros encontrados em r(x). Para resolver este sistema, uma opção é utilizar o algoritmo de Peterson, que é detalhado na próxima seção

2.5.3 Algoritmo de Peterson

Peterson (PETERSON, 1960) demonstrou que o sistema de equações 2.36 pode ser resolvido de maneira mais fácil utilizando um polinômio localizador de erros na forma:

$$\Lambda(x) = \prod_{l=1}^{v}(1 - X_l x) = \Lambda_0 + \Lambda_1 x + ... + \Lambda_v x^v \tag{2.37}$$

Neste caso pode-se relacionar as síndromes, com os coeficientes de $\Lambda(x)$ com as equações:

$$\begin{aligned}
&S_1 + \Lambda_1 = 0 \\
&S_2 + \Lambda_1 S_1 + 2\Lambda_2 = 0 \\
&S_3 + \Lambda_1 S_2 + \Lambda_2 S_1 + 3\Lambda_3 = 0 \\
&... \\
&S_v + \Lambda_1 S_{v-1} + \Lambda_2 S_{v-2} + ... + \Lambda_{v-1} S_1 + v\Lambda_v = 0 \\
&S_{v+1} + \Lambda_1 S_v + \Lambda_2 S_{v-1} + ...\Lambda_v S_1 = 0 \\
&... \\
&S_{2t} + \Lambda_1 S_{2t-1} + \Lambda_2 S_{2t-2} + ... + \Lambda_v S_{2t-v} = 0
\end{aligned} \tag{2.38}$$

Sobre um corpo finito $GF(2^q)$ pode-se escrever:

$$S_{2j} = \sum_{l=1}^{v} X_l^{2j} = (\sum_{l=1}^{v} X_l^j)^2 = S_j^2 \tag{2.39}$$

Com essa propriedade é possível obter que as equações pares, no sistema de equações 2.38, são redundantes as equações ímpares, de modo que é possível eliminá-las. Este sistema de equações pode ser representado em forma matricial:

$$A'\Lambda = \begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ S_2 & S_1 & 1 & ... & 0 & 0 \\ S_4 & S_3 & S_2 & ... & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ S_{2t-4} & S_{2t-5} & S_{2t-6} & ... & S_{t-2} & S_{t-3} \\ S_{2t-2} & S_{2t-3} & S_{2t-4} & ... & S_t & S_{t-1} \end{bmatrix} \begin{bmatrix} \Lambda_1 \\ \Lambda_2 \\ \Lambda_3 \\ \vdots \\ \Lambda_{t-1} \\ \Lambda_t \end{bmatrix} = \begin{bmatrix} S_1 \\ S_3 \\ S_5 \\ \vdots \\ S_{2t-3} \\ S_{2t-1} \end{bmatrix} \tag{2.40}$$

Este sistema de equações tem uma solução única, se e somente se A' for inversível. Por causa da propriedade $S_{2j} = S_j^2$, A' será inversível se houver t ou t-1 erros. Se a matriz não for invertível, as 2 colunas mais a direita, e as 2 linhas mais abaixo são removidas, até se obter uma matriz invertível. Deste modo pode-se resolver o sistema de equações para se obter os coeficientes Λ_i.

Quando t é pequeno a matriz pode ser analiticamente e pode-se obter os resultados de forma direta. Por exemplo, os casos de t=1 até t=3 podem ser obtidos como:

Caso t=1:

$$\Lambda_1 = S_1 \tag{2.41}$$

Caso t=2:

$$\begin{aligned} \Lambda_1 &= S_1 \\ \Lambda_2 &= \frac{S_3 + S_1 S_2}{S_1} \end{aligned} \tag{2.42}$$

Caso t=3:

$$\begin{aligned} \Lambda_1 &= S_1 \\ \Lambda_2 &= \frac{S_1^2 + S_3 + S_5}{S_1^3 + S_3} \\ \Lambda_3 &= (S_1^3 + S_3) + S_1 \Lambda_2 \end{aligned} \tag{2.43}$$

Nestes caso não é necessário o processo iterativo para a solução do sistema de equação, o que pode fornecer uma otimização significativa da implementação em hardware.

2.5.4 Chien Search

Para resolver a equação 2.37 deve-se aplicar uma técnica para encontrar a raiz de uma equação em corpos finitos. Uma técnica que é bastante utilizada é o *Chien Search*, que consiste em buscar o resultado de forma exaustiva testando todos os resultados possíveis. Existem várias arquiteturas de *Chien Search* na literatura, sendo utilizadas buscas serialmente, em paralelo, ou parcialmente em paralelo, entre outras.

2.6 RISC-V

O RISC-V é uma ISA (*Instruction Set Architecture*) de código aberto que utiliza a licença BSD (*Berkeley Software Distribution*) . Dizer que uma ISA é de código aberto com licença BSD significa dizer que o conjunto de instruções especificados por ela é livre para uso para qualquer fim sendo o código aberto ou fechado. Esta característica é um atrativo para esta ISA, pois permite o uso desta ISA para desenvolver processadores a nível de pesquisa e utilizar os compiladores e ferramenta já disponíveis para esta arquitetura.

O RISC-V foi inicialmente desenvolvido na Universidade da Califórnia, em Berkeley, com o início do projeto em 2010 (PATTERSON; HENNESSY, 2016). Ele se baseia em princípios RISC (*Reduced Instruction Set Computing*), que tem o objetivo de desenvolver

processadores com um conjunto reduzido de instruções. Este tipo é diferente de arquiteturas do tipo CISC(*Complex Instruction Set Computing*), que implementam operações mais complexas que podem ser chamadas em uma linha de código *Assembly* para realizar uma sequencia complexa de operações que podem demandar vários ciclos de *clock*. No caso de uma arquitetura RISC, a ideia é que as instruções possam ser executadas em 1 ciclo de *clock*, e então uma operação complexa pode ser executada utilizando várias linhas de *Assembly*. Esse tipo de abordagem tem a vantagem de poder se implementar arquiteturas mais simples e mais eficientes, por outro lado, arquiteturas CISC podem otimizar em tempo de execução, ao implementar operações complexas de forma otimizada.

Dessa forma a base do RISC-V é composta por um conjunto simples de operações. No entanto, uma peculiaridade da especificação do RISC-V, é que ele permite utilizar conjuntos diferentes de operações, a depender da necessidade do projetista. Para que isso seja possível, a especificação do RISC-V introduz o conceito de extensões. Cada extensão consiste em um conjunto de comandos que podem ser acrescentados em uma implementação de um processador RISC-V. As extensões base são extensões que contém as instruções mínimas para a implementação de um processador RISC-V. Estas extensões contém operações básicas de registro em memória e operações básicas com inteiros. As extensões base variam a depender do número de bits que se quer utilizar na arquitetura e o número de registradores. A especificação do RISC-V permite implementações de 32 bits, 64 bits e 128 bits. Além disso, há a opção de utilizar 32 registradores ou 16 registradores. Na tabela 1as extensões base são apresentadas.

Tabela 1 – Exemplos de extensões base possíveis para uma implementação do *RISC-V*

Extensão Base	**Descrição**
RV32I	Módulo base de 32 bits e 32 registradores.
RV64I	Módulo base de 64 bits e 32 registradores.
RV128I	Módulo base de 128 bits e 32 registradores.
RV32E	Módulo base de 32 bits e 16 registradores.
RV64E	Módulo base de 64 bits e 16 registradores.
RV128E	Módulo base de 128 bits e 16 registradores.

A partir de alguma destas extensões base, podem ser acrescentadas operações, conforme a necessidade do projeto, como por exemplo, operações de multiplicação e divisão, operações com números flutuantes, e outros. Para fazer isso basta acrescentar extensões em cima de uma extensão base. Na tabela 2, algumas das extensões disponíveis na especificação são apresentadas.

Tabela 2 – Exemplos de extensões possíveis para uma implementação do *RISC-V*

Extensão	Descrição
M	Extensão com instruções de multiplicação e divisão para inteiros.
A	Extensão que habilita instruções atômicas.
C	Extensão que habilita instruções comprimidas.
N	Extensão com instruções para o uso de interrupções a nível de usuário.
F	Extensão com instruções para realizar operações com números flutuantes conforme padrão IEEE 754-200.
D	Extensão com instruções para realizar operações com números flutuantes de precisão dupla conforme padrão IEE 754-200.
Q	Extensão com instruções para realizar operações com números flutuantes de precisão quádrupla conforme padrão IEEE 754-2008.

A partir destas possibilidades, diversas implementações do RISC-V estão disponíveis em código aberto que podem ser utilizadas para a implementação deste projeto. Uma implementação particular disponível em código aberto sob a licença ISC, é o *PicoRV32*, do qual serão apresentados mais detalhes a seguir.

2.6.1 *PicoRV32*

O *PicoRV32* é um projeto em código aberto de um RISC-V. Esse projeto implementa um núcleo CPU que implementa a extensão RV32IMC do RISC-V. Isso significa que ele possui operações básicas com inteiros, multiplicação, divisão e possui suporte para instruções comprimidas. A opção de instruções comprimidas é interessante pois permite diminuir o tamanho do programa, fazendo com que se economize espaço de memória do programa.

Este CPU é customizável, permitindo que se implemente a CPU com as extensões RV32E, RV32I, RV32IC, RV32IM ou RV32IMC, além de possuir opções para a implementação de um controlador interno de interrupções. O código é implementado em *Verilog*, e as customizações da CPU podem ser realizadas por meio de macros disponíveis, que definem como que o código em *Verilog* será compilado, adicionando ou não funcionalidades à CPU.

Outra opção de customização disponível é a utilização de interfaces de memória. Existe a opção de implementação de uma interface AXI4-Lite, uma interface *Wishbone* ou uma interface simplificada, própria do projeto, permitindo uma ampla compatibilidade com blocos externos.

2.6.2 PicoSoC

Também disponível no repositório do *PicoRV32* existe o *PicoSoC*. Este é uma implementação de um *PicoRV32*, com uma interface UART (*Universal Asynchronous Receiver/-Transmitter*), uma implementação de SRAM e um controlador de memória QSPI (*Quad Serial Peripheral Interface*). Nesta implementação o programa é armazenado em uma memó-

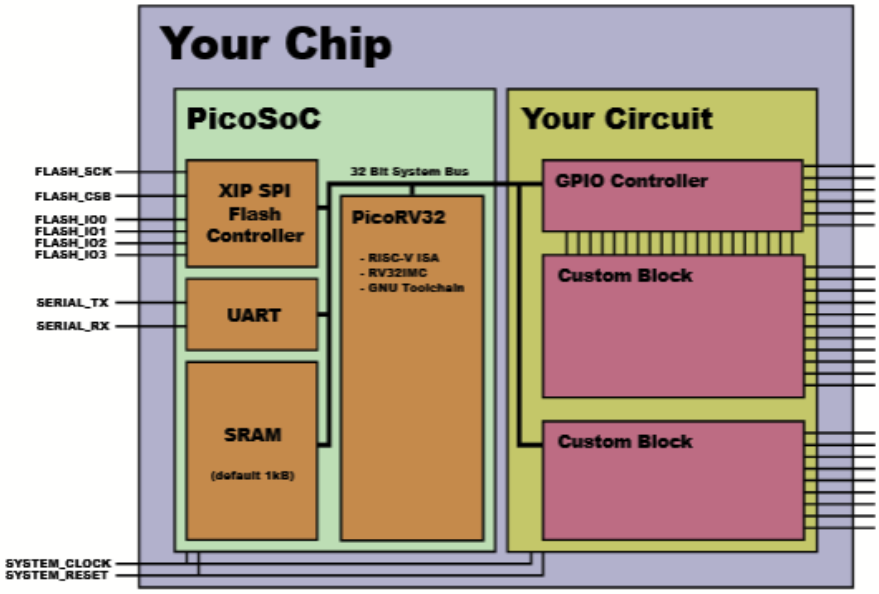


Figura 11 – Blocos do *PicoRV32*, disponível em (PICORV32…, 2025)

ria flash com interface QSPI, e pode ser acessado por meio do controlador. O bloco de UART permite comunicação de forma serial, podendo servir como interface para se comunicar com o processador. A memória SRAM é a memória que pode ser usada pelo programa em tempo de execução.

Esse SoC contém implementações para FPGAs de vários modelos diferentes, permitindo o uso dos recursos da FPGA, como memória flash e blocos de memória RAM, que podem ser acessíveis por meio de macros. Caso se deseje implementar este SoC em ASIC, deve-se utilizar IPs para implementar o bloco de SRAM, não podendo se utilizar o código como é descrito no repositório. Para este projeto, não será implementado o bloco de SRAM, sendo esta integração com IPs próprios, objeto de trabalhos futuros.

Na figura 11 um diagrama do *PicoSoC* é apresentado, no qual pode-se observar os blocos descritos. Além dos blocos descritos, existe a possibilidade de acrescentar mais blocos através de um barramento de 32 bits. Esse barramento segue o padrão de interface próprio do *PicoSoC* e não implementa padrões como o *Wishbone* ou o AXI4-Lite.

Para este trabalho, será utilizado o *PicoSoC* e a integração com o *PicoSoC* seguirá o padrão do diagrama apresentado na figura 11. Os blocos serão implementados em *Verilog* e serão integrados a partir do barramento de 32 bits apresentado no diagrama.

2.7 Estado da Arte

Um levantamento do estado da arte foi realizado para obter implementações de camadas físicas compatíveis com o padrão IEEE 802.15.6. A partir do levantamento foi

possível encontrar implementações em ASIC do padrão, assim como um implementação em FPGA. Também foram encontradas algumas implementações da comunicação UWB, mas em mais quantidade foram encontradas implementações da comunicação em NB, que são incluídas aqui, visto a semelhança na implementação da camada física. As implementações encontradas são datadas entre 2014 e 2017, isso acontece pois este período teve um maior esforço na pesquisa em torno deste assunto, visto que o padrão em questão foi publicado em 2012.

No artigo de (CHEN et al., 2013) um processador banda base para um transmissor WBAN é implementado. Neste caso, apenas o transmissor é implementado. Para este trabalho, foi implementado um transmissor NB, utilizando o padrão IEEE 802.15.6. A implementação para o processamento em banda base, segue os blocos descritos na norma, para o caso do NB, e se utiliza de uma FSM para controlar os blocos. Além disso, é utilizada a técnica de *clock gating*, para diminuir o consumo energético, garantindo que quando o sistema estiver ocioso, os blocos que não estão sendo usados, estejam desligados. Este sistema foi primeiro simulado em MATLAB, e depois implementado para ASIC, em tecnologia CMOS de 130 nm.

É importante notar que os requisitos do padrão para comunicação em NB, são bastante semelhantes ao UWB, no que tange ao processamento digital em banda base. Um ponto importante de distinção entre o NB e o UWB, é que no caso do NB é necessário um bloco a mais chamado de *spreader*. Esse bloco é responsável por repetir o bit que está chegando na entrada, dependendo da taxa de transmissão.

Uma outra implementação da norma para o caso de comunicação NB é (MATHEW et al., 2014) em que um transceptor banda base é implementado, utilizando o padrão IEEE 802.15.6. Para este caso , diferente do trabalho acima citado, o circuito é implementado utilizando FPGA. Uma particularidade dessa implementação é o decodificador BCH. Para realizar a etapa de KES, utiliza-se um algoritmo de Berlekamp Massey sem inversão, que é uma variante deste algoritmo que permite economizar energia na execução da operação. Antes da implementação em FPGA, foi simulado o sistema utilizando MATLAB, para validá-lo, e a partir disso o sistema foi implementado em FPGA, utilizando VHDL. Por fim, os resultados da execução foram monitorados utilizando um barramento PCIe, para monitorar o fluxo de dados, e o correto funcionamento do algoritmo.

O bloco BCH em particular é um dos blocos mais complexos do padrão, e ele permite uma variedade de abordagens na sua implementação, sendo assim um ponto chave de distinção entre os artigos.

No trabalho de (BACHMANN et al., 2014) uma abordagem mais versátil é apresentada com a implementação de um transceptor banda base multi padrão. O processador banda base digital deste artigo tem a capacidade de operar em 3 padrões diferentes, o IEEE 802.15.4 (ZigBee), IEEE 802.15.6 (WBAN) e o padrão *Bluetooth Smart* (*low energy*, BLTE). Neste caso, o circuito é feito com blocos que sejam capazes de atender a cada um dos padrões. Um sistema

com vários *MUX's* é utilizado, que alterna entre os blocos necessários para cada padrão. São implementados 2 BCHs nesse caso, um BCH(51, 63) e um BCH(31, 19), que são alternados dependendo do padrão. Além disso, blocos de *spreader*, *scrambler*, *interleaver* e *whitening* são alternados pelos *MUX's* também. Este é um circuito que possui uma maior versatilidade, pois permite escolher entre diferentes padrões. Além disso são aplicadas técnicas para reduzir a energia consumida, como conversores DC-DC e uma unidade de gerenciamento de *clock*. O circuito foi implementado em tecnologia CMOS 40nm.

Ainda outra implementação de comunicação NB é apresentada em (EL-MOHANDES; SHALABY; SAYED, 2018), um transceptor banda base no padrão IEEE 802.15.6. Neste trabalho, o circuito foi implementado em ASIC. Para realizar isto, foi utilizado um fluxo de trabalho que inicia com simulações no MATLAB, e a partir destas, foram gerados IPs que foram simulados, testados em FPGA, e depois implementados em ASIC. A tecnologia usada neste trabalho é de 130nm. Outro ponto de distinção neste caso é a arquitetura do BCH, que utiliza uma arquitetura de Berlekamp-Massy e também utiliza um algoritmo de *Chien Search* sequencial que leva 63 ciclos para finalizar um operação.

Já em (MANCHI; PAILY; GOGOI, 2017) um transceptor UWB banda base compatível com o padrão IEEE 802.15.6 é implementado. Neste artigo é implementado um receptor e um transmissor, sendo implementada a porção digital da camada física descrita na norma. Este circuito é implementado tanto em tecnologia de 180nm, como também em tecnologia de 90m. O circuito é validado em uma FPGA modelo ZYNQ, e utilizando simulação no MATLAB, para gerar sinais de entrada no circuito digital, e partir de um analisador lógico, obter os resultados. O circuito proposto, tem como diferencial uma arquitetura de BCH que contém uma arquitetura otimizada do KES, e uma arquitetura de *Chien Search* com terminação antecipada. Com essas modificações é possível economizar 42% em área e 38% em consumo de energia.

A partir dos resultados obtidos neste último trabalho, é possível intuir que uma implementação com arquiteturas mais simples para o BCH, pode levar a uma eficiência maior do circuito. Isso foi possível pela utilização do KES otimizado, em oposição a uma arquitetura como Berlekamp-Massy e também pela utilização de uma arquitetura otimizada do *Chien Search* com terminação antecipada. Com base nestas observações, uma arquitetura similar do KES será utilizada neste trabalho. Para o *Chien Search* será utilizada uma arquitetura com maior capacidade de paralelismo para obter maior desempenho, em oposição a (EL-MOHANDES; SHALABY; SAYED, 2015) que optou por uma arquitetura sequencial que demora 63 ciclos para realizar apenas este passo.

3 Aspectos Metodológicos

Circuito Integrados de Aplicação Específica, ou ASIC (*Application Specific Integrated Circuit*) são circuitos desenvolvidos para um propósito específico utilizando uma tecnologia de fabricação de circuitos integrados. Exemplos de ASICs são microcontroladores, microprocessadores, chips controladores USB, memórias RAM, EEPROM entre outros. Esses tipos de circuitos são largamente utilizados na eletrônica. Isso se deve ao fato de que esses circuitos podem oferecer funções complexas, com alto desempenho e área reduzida, além de oferecer um custo menor por unidade, comparado ao mesmo circuito realizado utilizando componentes discretos.

Sistemas ASIC se diferenciam de FPGAs (*Field Programmable Gate Array*), pois FPGAs oferecem a possibilidade de utilizar uma matriz de portas lógicas e registradores interconectadas que podem ser reconfiguradas para realizar diversas funções, e assim é possível utilizar FPGAs para fazer a função de diversos circuitos diferentes, apenas rearranjando a matriz de células deste dispositivo. No caso do ASIC o circuito é desenvolvido e decidido antes da fabricação e não permite o rearranjo das suas conexões para trabalhar como um outro circuito.

Grande parte dos circuitos integrados são desenvolvidos em tecnologia CMOS (*Complementary Metal-Oxide-Semiconductor*). CMOS é uma tecnologia de fabricação que consiste na fabricação de transistores MOSFET (*Metal Oxide Semiconductor Field Effect Transistor*) do tipo NMOS e PMOS em um mesmo substrato de silício (RAZAVI, 2021). Essa tecnologia permitiu avanços no desenvolvimento de circuitos integrados como microprocessadores, e é largamente utilizada na indústria de semicondutores.

O uso desta tecnologia permitiu a implementação de circuitos cada vez mais eficientes, visto que com o passar do tempo é possível produzir transistores cada vez menores, o que leva a possibilidade fazer circuitos menores, que ocupam menos espaço, produzindo chips mais compactos. Outra vantagem é a tendência que há quando se reduz um transistor de este ficar mais rápido, isso acontece pois transistores menores tem menor capacitância nos seus terminais, que oferece um atraso menor, resultando em circuitos mais rápidos. Além disso o consumo de energia diminui com transistores menores. Isso permite o desenvolvimento de chips cada vez mais complexos e eficientes.

3.1 Níveis de Abstração

O grande desafio em projetos VLSI (*Very Large Scale Integration*) de circuitos integrados atuais é administrar a complexidade destes sistemas. Um SoC (*System on Chip*) moderno

conta com memórias, processadores, circuitos dedicados de como ADCs e interfaces IO de alta velocidade. Para produzir tais sistemas é necessário integrar centenas de milhões de transistores, ou bilhões de transistores. Para lidar com esta complexidade é necessário utilizar o conceito de níveis de abstração.

Para que seja possível desenvolver sistemas com tamanha complexidade é necessário dividir o projeto do sistema em diversos níveis de abstração, ocultando os detalhes sempre que eles não forem necessários. Projetos digitais em VLSI costumam ser particionados em 5 níveis de abstração: projeto de arquitetura, projeto de microarquitetura, projeto lógico, projeto de circuito e projeto físico. Esses níveis podem ser melhor explanados utilizando um exemplo, a arquitetura RISC-V especifica um conjunto de instruções, um modelo de registradores e memória. A partir desta arquitetura podem se especificar microarquiteturas que dividem a arquitetura em registradores e blocos funcionais. Exemplos de microarquiteturas de RISC-V que podem ser citados é a ESP32-C3, SiFive FE310 e GD32VF103. O projeto lógico define como que os blocos funcionais são estruturados, por exemplo pode se utilizar máquinas de estado que codificam seus estados utilizando uma codificação binária, em código Gray ou em codificação *One-Hot*. Descendo mais um nível de abstração, no projeto de circuito é descrito como que os transistores se interconectam para implementar a lógica. Nesse caso pode-se utilizar diferentes circuitos para implementar portas lógicas ou registradores, podendo-se inclusive projetar circuitos com o propósito de economizar energia, para projetos mais eficientes energeticamente. E finalmente o projeto físico lida com como esses transistores vão ser posicionados no substrato, e como que cada dispositivo irá se conectar fisicamente.

Outra prática importante para lidar com a complexidade é a prática do projeto estruturado. Esta prática usa os princípios de hierarquia, regularidade, modularidade e localidade. O princípio de hierarquia consiste em particionar um projeto em várias hierarquias, onde hierarquias mais altas representam blocos mais complexos, e um grau de abstração maior, enquanto que hierarquias menores representam blocos menos complexos, até chegar no transistor que é o nível mais baixo de abstração. O uso de hierarquias torna mais fácil entender o circuito quando se analisa os componentes do circuito como caixas pretas com interfaces e funções bem definidas, ao invés de analisar cada transistor individualmente. Regularidade é o princípio de administrar a complexidade projetando o mínimo de blocos diferentes possível. Quando se cria blocos padronizados, é possível reutilizá-los, e desse modo é possível reduzir o trabalho, tornando o projeto mais eficiente. O princípio de modularidade refere-se ao uso de interfaces bem definidas para os blocos, de modo a evitar comportamentos inesperados. Finalmente o princípio de localidade refere-se a manter informação onde ela é usada, fisicamente e temporalmente.

Outra forma de ver o particionamento de projeto que é conhecido na literatura é o diagrama Y. Esse diagrama divide o projeto de circuitos integrados digitais em 3 eixos, que representam 3 aspectos diferentes do projeto, esses são os 3 domínios: Estrutural, Compor-

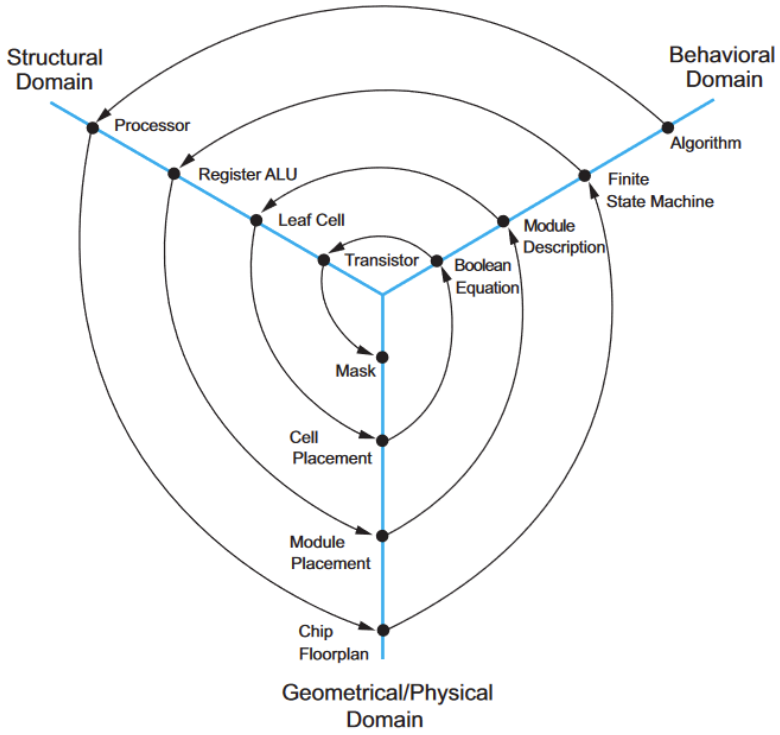


Figura 12 – Diagrama Y de graus de abstração (WESTE; HARRIS, 2015)

tamental e Geométrico/Físico. Cada domínio representa um aspecto diferente do projeto, e é dividido em níveis de abstração, sendo que cada nível tem um resultado concreto, que pode ser entregado, e dessa entrega é possível transformar um resultado de um domínio para o outro, e seguir descendo o nível de abstração até obter o maior nível de detalhe com a máscara completa, com todos os transistores e componentes detalhados fisicamente, pronto para ser entregue para a fábrica.

No domínio comportamental descreve-se o que o sistema faz. Em um nível mais abstrato pode-se descrever o que o chip como um todo faz, baseado em sua interface, e quais funções ele exerce. Conforme descemos o nível de abstração deve-se definir os blocos necessários para se realizar estas funções. No diagrama é possível ver que o nível mais alto de abstração é o algoritmo, que define o comportamento geral do circuito. Ao seguir descendo os níveis, temos a máquina de estados, que será responsável por estas funções, a descrição de módulos mais básicos, até chegar nas equações booleanas que vão descrever a lógica combinacional para os blocos mais básicos do sistema.

No domínio estrutural descreve-se como que se irá conectar os módulos do sistema para realizar um determinado comportamento. Nesse caso, o diagrama temos os níveis processador, registrador ULA(Unidade Lógico Aritmética), célula padrão. Deste modo, em nível mais abstrato pode-se definir os blocos de um processador, por exemplo, pode-se definir como memórias vão se conectar, banco de registradores, interfaces IO e assim em diante. Depois pode-se definir como que registradores e ULAs vão se interconectar dentro desses

blocos. Finalmente deve-se definir como que cada célula padrão vai se interconectar para formar o circuito.

No domínio físico descreve-se como construir fisicamente cada nível de abstração. No caso do diagrama temos os níveis de *floorplan* ("Planta baixa"), posicionamento dos módulos, posicionamento das células, máscara. Isso significa que em um nível mais abstrato, deve-se considerar como que o chip como um todo será posicionado fisicamente, considerando os blocos de nível de abstração mais alto. Seguindo mais abaixo, deve-se posicionar os módulos de cada bloco, depois deve-se posicionar cada uma das células, até ser possível formar a máscara, que tem todos os detalhes necessários para se fabricar o chip.

O processo de projeto de um ASIC então pode ser encarado como o processo de transformar cada um desses artefatos, de um domínio para outro, descendendo os níveis de abstração até ter todos os detalhes necessários para a fabricação do chip. Essas transformações podem ser feitas tanto por um engenheiro manualmente, como também por ferramentas de software. De modo que este processo, definidas as ferramentas e os métodos para chegar no produto final é denominado como o fluxo de projeto.

3.2 Fluxo de projeto Digital

Nesta seção será detalhado o fluxo de projeto utilizado para este trabalho. Para realizar um projeto digital e lidar com a complexidade de um projeto de circuitos integrados, o uso de ferramentas de software para auxiliar na automatização do projeto é essencial. Na imagem 13 o fluxo de projeto digital é sintetizado, e os estágios gerais do desenvolvimento são listados. Para este projeto é utilizado a suíte de ferramentas de desenvolvimento da Cadence de circuitos integrados digitais.

3.2.1 *Hardware Description Language* (HDL)

Linguagens de Descrição de Hardware (HDL) são linguagens utilizadas para descrever circuitos, majoritariamente circuitos digitais. Este tipo de linguagem começou a ser utilizada entre as décadas de 1970 e 1980 para descrever circuitos eletrônicos sem precisar utilizar as representações gráficas de esquemáticos, que conforme os circuitos atingem grande complexidade, esquemáticos acabam perdendo a utilidade em auxiliar o desenvolvimento (VAHID, 2010). Essas linguagem são capazes de definir a conexão entre componentes e também o comportamento de circuitos. Exemplos destas linguagens são o VHDL, o *Verilog* e o *SystemC*.

As HDLs são essenciais no fluxo de desenvolvimento digital, pois com elas é possível descrever blocos de circuitos digitais, no que tange ao seu comportamento. Com essas descrições comportamentais é possível utilizar ferramentas de síntese para transformar código HDL em circuitos lógicos.

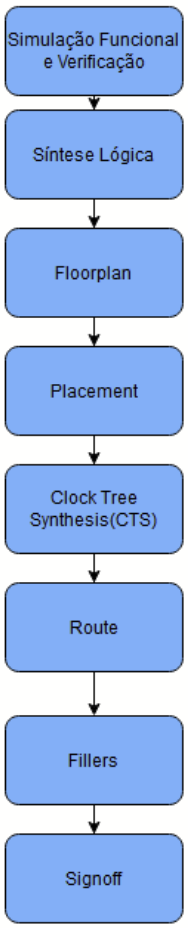


Figura 13 – Fluxo de Projeto Digital

Para este trabalho foi utilizado a linguagem *Verilog* para descrever os circuitos utilizados neste projeto. O desenvolvimento de código HDL é o primeiro passo do fluxo de desenvolvimento, após ser decidido quais circuitos serão implementados e qual a funcionalidade geral do bloco a ser desenvolvido.

3.2.2 Simulação Funcional e Verificação

Após desenvolver o código é necessário simular este código para entender se o comportamento descrito em HDL se alinha com o comportamento esperado nos requisitos do projeto.

A ferramenta *Xcelium* é usada neste fluxo de desenvolvimento. Com ela é possível compilar códigos HDL, e é possível fazer simulações em que se pode realizar depuração do código para atestar o correto funcionamento do circuito.

Nesta etapa de verificação é possível utilizar várias técnicas de verificação para validar a descrição em hardware, como o uso de técnicas de verificação estática, verificação formal, verificação baseada em asserções entre outras. Para este projeto foi utilizado um tipo de

verificação que se dá o nome de verificação baseada em simulação, onde se usa um *testbench* que é um código em HDL que instancia o módulo a ser testado, e insere estímulos nas entradas dos circuitos e analisa-se as saídas para verificar que o funcionamento do circuito respeita os requisitos do projeto.

3.2.3 Síntese Lógica

O próximo passo no fluxo de desenvolvimento é a síntese lógica. Esse passo consiste em transformar o código HDL em registradores e portas lógicas genéricas. Este passo é automatizado e conta com o auxílio de ferramentas de software para fazer o mapeamento do código HDL para um circuito usando portas lógicas e registradores.

Na etapa de síntese o circuito também deve ser mapeado para uma biblioteca de células padrão, que deve ser uma biblioteca com células básicas para alguma tecnologia de fabricação. Mapeando para essa biblioteca, são realizadas otimizações de área, potência e de sincronização do circuito. Neste passo é utilizado um arquivo de *constraints* no qual se define regras do projeto, como o período do *clock* a ser utilizado ou o tempo de atraso de pinos de entrada do circuito.

Para realizar esta etapa, foi utilizado a ferramenta *Genus*, com qual pode-se realizar a síntese lógica e as otimizações necessárias neste passo, além de ser possível obter estimativas de consumo de área, potência e o *timing* do circuito.

Após gerar o circuito mapeado para uma biblioteca de células, é possível realizar novas simulações funcionais com *Xcelium*, para verificar que há equivalência entre o circuito gerado, e o comportamento descrito em HDL.

3.2.4 *Floorplan*

Nesta etapa inicia-se a implementação física do circuito, com o objetivo final de definir a disposição física de cada um dos componentes do circuito e gerar um arquivo com o qual a fábrica poderá gerar as máscaras para a produção do circuito integrado.

Na etapa de *floorplan*, são posicionados blocos de hierarquia mais alta. Nesta etapa é onde se decide onde posicionar por exemplo CPUs, interfaces de alta velocidade, GPUs, blocos de memória, ADCs entre outros. Esta é uma parte que exige um cuidado maior do engenheiro, para posicionar os blocos de maneira adequada conforme as regras do projeto.

Nesta etapa é realizado também o *powerplan*, que é uma etapa onde é distribuído a alimentação ao longo do chip. As malhas de alimentação são definidas e roteadas para cada bloco e distribuído para as fileiras onde as células serão posicionadas posteriormente.

Esta etapa, assim como as demais etapas de implementação física, são realizadas com o auxílio da ferramenta Innovus, tem a função de realizar estes passos de implementação fí-

sica, fazer otimizações, e fazer verificações físicas no circuito. Estas etapas de implementação física também são conhecidas como *Place & Route*.

3.2.5 Placement

A etapa de *placement* consiste em posicionar as células padrão do circuito. Diferente da etapa anterior em que se posiciona blocos de hierarquia maior, neste caso serão posicionados os blocos de menor hierarquia, como portas lógicas e registradores. Outra diferença da etapa anterior é que nesta etapa o posicionamento das células é automático, e feito pelo software.

Nesta etapa são feitas otimizações, de potência, área e *timing*. Células podem ser adicionadas, como *buffers*, para resolver por exemplo problemas com capacitâncias altas de fios, quando duas células estão longe uma da outra, de modo a gerar conexões com capacitâncias muito alta entre elas. Vários caminhos combinacionais de lógica podem ser rearranjados e alterados para otimizar o consumo de área e potência, mantendo a equivalência lógica.

3.2.6 *Clock Tree Synthesis*

Na etapa de síntese de árvore de relógio, as conexões entre os *clocks* do circuito e as células sequenciais que utilizam o *clock* são realizadas.

Esta é uma etapa importante para o correto funcionamento do circuito, porque o roteamento dessas conexões implica em conexões de tamanhos diferentes para cada elemento sequencial do circuito, visto que as células estão distribuídas ao longo do chip. Isso significa que o sinal de *clock* tem atrasos de magnitudes diferentes para cada um dos registradores no circuito. Além disso, não é razoável ter conexões de muito longas, visto que isso gera fios com capacitâncias muito altas que podem degradar o circuito, promovendo comportamentos não esperados.

Para resolver isso é necessário construir uma árvore de *clock*. A árvore é construída por *buffers* que repetirão o sinal de *clock*, e serão posicionados de modo a garantir a integridade do sinal de *clock*.

Nesta etapa é feito também o balanceamento desta árvore de *clock*. Cada um dos *buffers* adicionados, geram uma quantidade de atraso no *clock*, que serão percebidas na entrada de *clock* de cada elemento sequencial. Para garantir o bom funcionamento do circuito, esses *buffers* são arranjados de tal forma que cada elemento sequencial do circuito receba uma quantidade similar de atraso do *clock*, assim garantindo uma sincronia adequada destes elementos.

Outro passo importante a ser citado na síntese da árvore de relógio, é a otimização pós síntese da árvore. Esta é uma otimização que ocorre após a síntese da árvore, em que

é analisado o *timing* do circuito, assim como potência e área, e a árvore de *clock* é então ajustada de forma a obter resultados otimizados.

A otimização da árvore de relógio, ou as otimizações, podem ser customizadas de maneira a obter resultados diferentes. Pode-se focar a otimização em baixo consumo de energia, em otimização de violações de *timing* em *hold*, ou violações *timing* em *setup*.

3.2.7 *Route*

Após rotear e otimizar a árvore de *clock*, todas as conexões restantes devem ser roteadas. O passo de *route* então consiste no roteamento de cada uma das conexões do circuito.

Assim como os passos anteriores, esta etapa é realizada, e otimizações são realizadas para aprimorar as métricas projeto. Nesta etapa ainda podem ser inseridas células e rearranjados caminhos combinacionais, conforme estimativas melhores sejam realizadas das métricas do projeto, com o circuito agora roteado.

3.2.8 *Fillers*

Após realizar o roteamento do circuito, a etapa que se segue é a adição de células *fillers* e metais *filler*. Essas células tem o objetivo de preencher lacunas que restem entre as células do circuito implementado, para que seja possível completar a malha de alimentação do circuito. Além disso, para que o circuito integrado seja fabricado adequadamente, a fábrica exige uma densidade mínima de metal, para cada uma das camadas de metal do chip. Desta forma é necessário adicionar formas de metal, que não conectam no circuito e nem tem funcionalidades, mas tem o objetivo de aumentar a densidade de metal do circuito para atingir a densidade mínima exigida.

3.2.9 Verificações Finais (*Signoff*)

Após a implementação de todos estes passos é necessário verificar o circuito. É necessário fazer as verificações de *static timing analysis* (STA), para checar o *timing* do circuito, e verificar a sincronia dos elementos sequenciais do circuito.

Também é necessário checar por violações de DRC (*Design Rule Check*). Essas são violações as regras de projeto da fábrica. As fábricas de circuitos integrados contam com uma vasta quantidade de regras para o projeto do circuito, que são necessárias para garantir a correta fabricação do chip. Caso encontradas violações de DRC, faz-se necessárias corrigí-las.

Aqui também é possível realizar as estimativas finais de área e consumo energético do circuito.

3.2.10 PDK

Para que todos estes passos sejam executados, é necessário o uso de um PDK (*Process design kit*). PDKs são kits de desenvolvimento que são fornecidos pela fábrica responsável pela fabricação do chip. Esses kits de desenvolvimento contam com diversas bibliotecas, que incluem modelos dos transistores utilizados, diodos, células de lógica digital, portas de IO, dentre outras coisas.

Para este projeto, foram utilizados 2 PDKs diferentes. O primeiro é um PDK da UMC, de 180nm e o segundo é um PDK da TSMC de 22 nm. Estes são PDKs que incluem células digitais para o desenvolvimento dos circuitos presentes neste trabalho.

O uso destas 2 tecnologias foi feito por conta da disponibilidade destes PDKs ao longo do projeto, tendo sido necessário alternar entre as 2 tecnologias.

3.2.11 Especifidades do projeto

Para realizar o desenvolvimento deste projeto, serão projetados alguns blocos de circuitos digitais. Para os blocos de camada física UWB e o bloco de filtro média móvel, seguirá o fluxo de desenvolvimento conforme descrito nesta seção, sendo os blocos descritos em *Verilog* desde o início, e passando por todos os passos do fluxo. Para o caso do RISC-V, o projeto será realizado a partir de um código aberto disponível em um repositório público. Deste modo, para o RISC-V o projeto parte de um código já feito e a partir deste, são realizadas adaptações para integrar os blocos descritos. Após fazer integração, também segue-se o fluxo digital já descrito.

3.3 Figuras de Mérito

Para avaliar o desempenho dos circuitos desenvolvidos, é necessário estabelecer figuras de mérito, com o intuito de avaliar o circuito de forma quantitativa. Em particular, estabelecer uma boa figura de mérito para este trabalho é um desafio. Isto porque os trabalhos disponíveis na literatura apresentam características muito diferentes, principalmente em relação ao processo de fabricação que se mostra bastante heterogêneo nestes trabalhos. Isto apresenta uma dificuldade pois a comparação de área e consumo energético não pode ser realizada diretamente, visto que o processo de fabricação influencia diretamente nessas medidas, o que gera um ruído na comparação de desempenho entre diferentes projetos.

Para realizar comparações um pouco mais justas em relação a estes projetos serão propostas 2 figuras de mérito que tem o intuito de reduzir a influência do processo de fabricação em sua medida.

3.3.1 *Gate Equivalent* (GE)

O *Gate Equivalent* (GE) é uma unidade de medida que permite avaliar a complexidade de um circuito integrado de forma independente do processo de fabricação (KAESLIN, 2008). Em geral, para tecnologia CMOS atual o circuito mais simples utilizado por um circuito digital é uma porta lógica NAND de duas entradas e uma saída. No GE esta porta representa a unidade de área para se medir a complexidade do circuito. O GE é calculado na forma:

$$GE = \frac{Area\ Total}{Area\ de\ 1\ NAND} \tag{3.1}$$

Basta dividir a área total do circuito pela área da porta NAND mais simples disponível na tecnologia do circuito. Esta medida representa a quantidade equivalente de portas NANDs que seriam necessárias para obter a área do circuito a ser avaliado. Desta forma é possível avaliar o consumo de células do circuito, e também a complexidade do circuito de forma independente da tecnologia.

3.3.2 Adaptação para o GE

O GE é uma medida conhecida da literatura, porém para este trabalho será necessário adaptar esta medida, visto que não temos acesso a área de uma porta NAND das tecnologias empregadas nos trabalhos da literatura. Desta forma, é proposto substituir a área de uma porta NAND, pela área obtida com um quadrado de lado igual ao tamanho da largura do transistor (L) de uma tecnologia. Por exemplo, para uma tecnologia de 22 nm, utiliza-se um lado de tamanho 22 nm. Deste modo pode-se obter a medida:

$$GE_{adaptado} = \frac{Area\ Total}{L^2} \tag{3.2}$$

Esta medida oferece uma precisão menor em relação ao GE, pois pode haver variações na área do circuito, a depender da biblioteca de células utilizada, mas ainda sim oferece uma medida melhor do que o valor bruto de área que é apresentado nos trabalhos da literatura.

3.3.3 Potência Normalizada

Para realizar a comparação entre o consumo de potência entre arquiteturas disponíveis na literatura e as arquiteturas propostas neste trabalho, será utilizada a potência normalizada. Essa figura de mérito é uma medida proposta no trabalho de (LIU et al., 2010), e tem o objetivo de fornecer uma medida de potência que é normalizada em relação a alguns parâmetros do circuito. A potência normalizada é equacionada na forma:

$$\eta = \frac{P_T}{(\frac{A}{L^2})V^2 f} \tag{3.3}$$

Onde η é a potência normalizada. $\frac{A}{L^2}$ é proporcional a quantidade de portas lógicas utilizadas pelo circuito. Isto porque A é a área total do circuito e L indica a escala do processo de fabricação, por exemplo, em uma tecnologia de 180 nm, usa-se 180 para L. V é a tensão de alimentação. E finalmente f é a frequência do circuito.

A potência normalizada representa o quanto de potência é dissipada em cada porta do circuito, por unidade de tensão e por ciclo de relógio. Essa medida fornece uma pista que permite avaliar se o circuito avaliado tem potencial de funcionar em baixa potência, independente da tecnologia.

4 Proposta de Circuitos

Para este trabalho, foram desenvolvidos um circuito que implementa a camada física segundo o padrão IEEE 802.15.6 e um circuito de filtro de média móvel simples que são integrados com um processador RISC-V. O objetivo é que estes circuitos sejam usados como controlador em um SoC que se comunica em UWB, realizando o processamento digital em base banda para implementar a camada física do UWB e pré-processamento de sensores que será realizado pelo filtro média móvel em conjunto com o RISC-V, que também tem a função de ser o controlador do SoC.

A seguir, circuitos para cada um dos blocos serão propostos e detalhados.

4.1 Filtro de Média Móvel Simples

O circuito para o filtro média móvel se baseia no fato de que os dados de entrada chegarão de forma serial. A partir dessa especificação, é possível realizar o cálculo de média de forma incremental, conforme os dados chegam ao filtro. Desta forma pode-se reutilizar valores de média usados previamente para calcular a média para novos dados, resultando em maior eficiência computacional. Da equação 2.2 pode-se escrever:

$$y(k) = \frac{u(k-1)}{n} + \frac{u(k-2) + u(k-3) + \cdots + u(k-n)}{n} \tag{4.1}$$

A equação 4.1 pode ser reescrita na forma:

$$y(k) = \frac{u(k-1)}{n} + \frac{u(k-2) + \cdots + u(k-n-1)}{n} - \frac{u(k-n-1)}{n} \tag{4.2}$$

Finalmente:

$$ny(k) = y(k-1) - u(k-n-1) + u(k-1) \tag{4.3}$$

Aplicando essa definição, se torna possível calcular a média móvel utilizando um único bloco de adição, uma única subtração e uma operação de divisão. No entanto, é possível simplificar ainda mais a operação substituindo a operação de divisão por uma operação de deslocamento de bits. Isso torna a arquitetura menos flexível mas aumenta a eficiência de consumo de área e consumo energético. Para esse caso a equação 4.3 pode ser escrita como:

$$2^n y(k) = y(k-1) - u(k-2^n-1) + u(k-1) \tag{4.4}$$

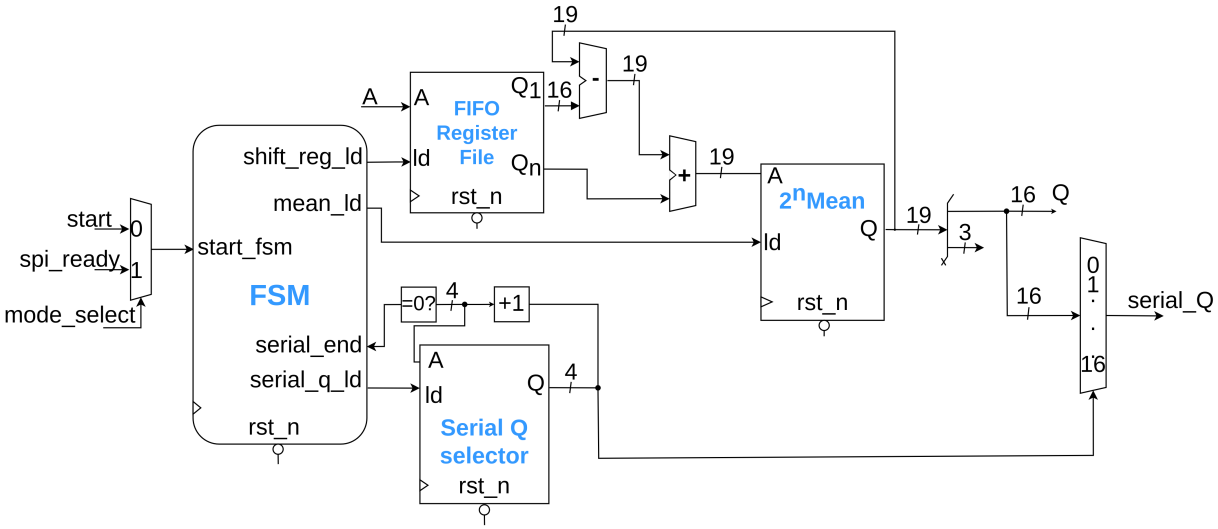


Figura 14 – Diagrama de blocos para o circuito do filtro de média móvel simples.

Com a equação 4.4 é possível calcular a média móvel utilizando deslocamento de bits. Na arquitetura proposta, o valor de n será de 3, resultando em um filtro média móvel de 8 taps.

A arquitetura do filtro média móvel está representada na figura 14. Ela tem uma máquina de estados (FSM) que controla as operação e o banco de registradores FIFO. Neste bloco, a entrada representada por A é carregada na primeira posição da FIFO quando a entrada *ld* é ativada. A saída $Q1$ corresponde a primeira posição da FIFO, que contém a primeira entrada lida, enquanto Qn representa a última posição da FIFO.

Os blocos de adição e subtração na saída da FIFO são responsáveis por realizar a equação 4.4, e o resultado de 19 bits é registrado no registrador 2^nMean. O próximo passo é performar a divisão por 2^n, que pode ser realizada por um deslocamento de bits do número para a esquerda. Essa operação emprega a concatenação para descartar os bits menos significativos, garantindo que o número mantém o tamanho de 16 bits e é deslocado 3 vezes para a esquerda.

O demultiplexador é usado para serializar os bits do resultado. Essa operação é útil para um protótipo inicial em que será realizado a transmissão dos dados de forma serial. Esse processo é feito com o auxílio do registrador *Serial Q selector*, que opera como um contador. Assim que o contador chega no seu valor máximo, ele gera o sinal *serial_end*, indicando para a FSM que o cálculo da média chegou ao fim.

4.1.1 Arquitetura do filtro para protótipo inicial

O circuito deste filtro foi enviado para uma rodada de fabricação, junto com outros circuitos analógicos do projeto Cedro. Para realizar testes com o filtro nesta primeira rodada de fabricação e validar o circuito, uma arquitetura que inclui uma interface SPI foi projetada.

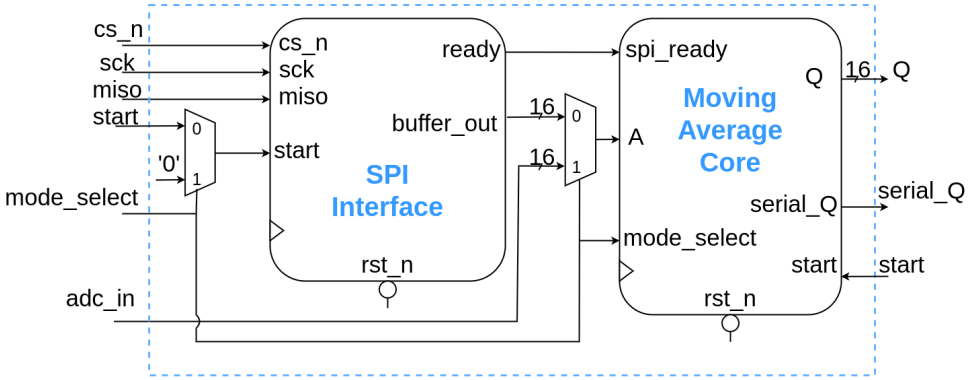


Figura 15 – Arquitetura do protótipo inicial do filtro média móvel.

Porta	Tamanho	Direção	Função
cs_n	1	Entrada	Interface SPI - Inicia comunicação em nível lógico baixo
sck	1	Entrada	Interface SPI - Clock
miso	1	Entrada	Interface SPI - Entrada de dados
start	1	Entrada	Sinal de controle para iniciar o bloco
adc_in	16	Entrada	Entrada paralela de 16 bits
serial_Q	1	Saída	Saída serial do sinal filtrado
Q	16	Saída	Saída paralela do sinal filtrado
mode_select	1	Entrada	Seleciona entre o modo SPI e o modo paralelo

Tabela 3 – Lista de portas do circuito de filtro média móvel e a descrição de suas funcionalidades

A figura 15 mostra um diagrama de blocos da arquitetura proposta. O bloco *SPI Interface* é responsável por fornecer uma interface serial entre o bloco de média móvel e um sensor. Desta forma é possível testar a eficiência do filtro média móvel em um caso em que se quer reduzir o ruído da leitura de um sensor.

A entrada de controle *mode_select* alterna entre 2 modos. O primeiro utiliza a interface SPI para se comunicar com o sensor, e o segundo modo recebe um número binário de 16 bits de forma paralela na entrada *adc_in*. A entrada *start* inicia o cálculo de média móvel e o resultado é apresentado em ambas as saídas *Q* e a versão serial *serial_Q*.

4.2 Camada física do UWB (IEEE 802.15.6)

Na figura 16 o diagrama geral do circuito desenvolvido para implementar a camada física do UWB é apresentado. Ele é composto de 4 partes principais, o transmissor, o receptor, as FIFOs e o controlador SPI. O receptor implementa as operações necessárias para a camada física em banda base para o processamento inicial da mensagem, implementando blocos de CRC, *de-interleaver*, SHR, BCH e *de-scrambler*. De forma similar, no transmissor, é feito o processamento da mensagem a ser enviada, sendo implementado cada um dos sub-blocos necessários para processar a mensagem antes de enviar.

Para um protótipo inicial, foi desenvolvido um controlador SPI. O objetivo é fornecer

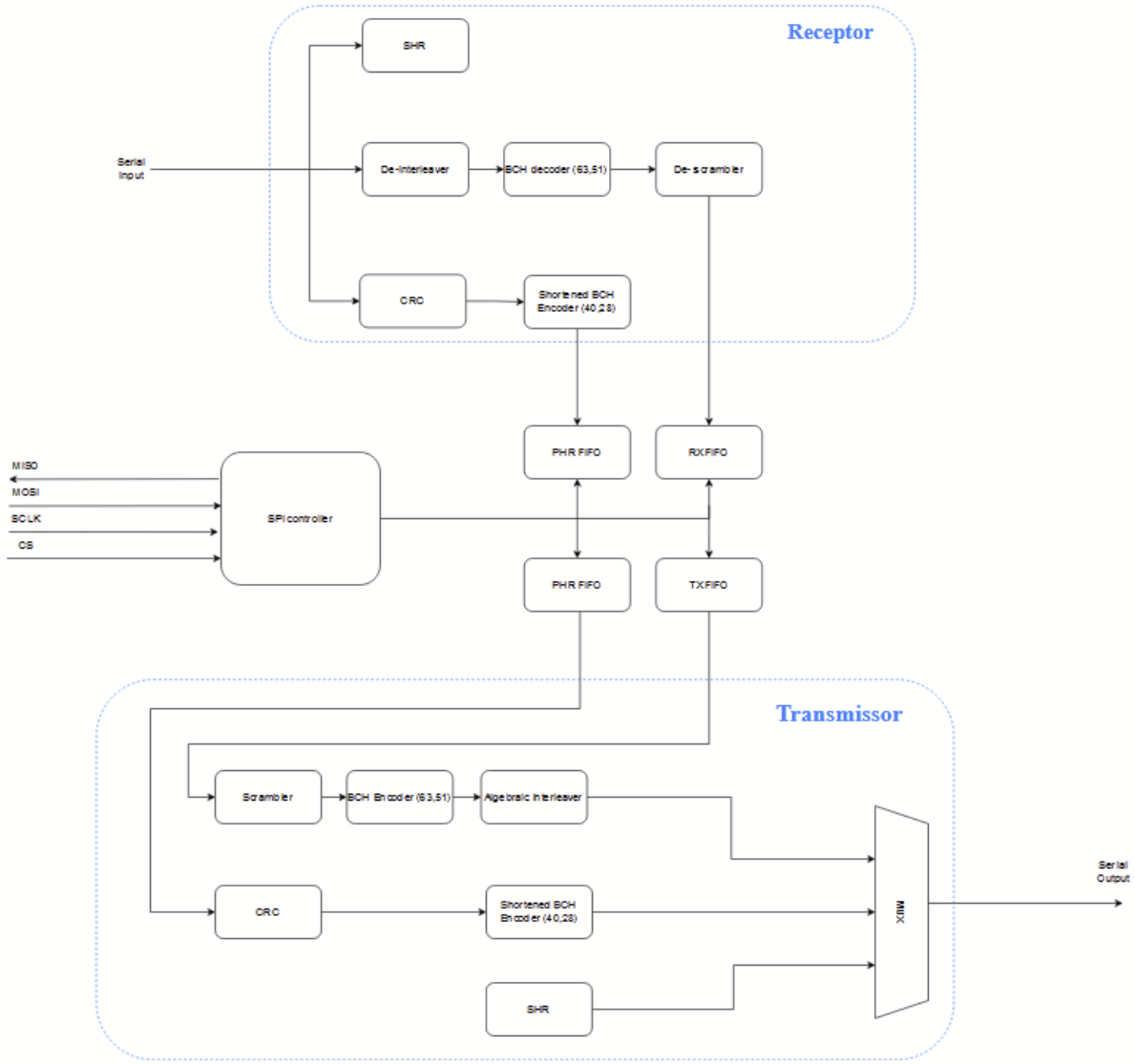


Figura 16 – Diagrama de blocos do circuito proposto para implementar a camada física do UWB, seguindo o padrão IEEE 802.15.6

Porta	**Tamanho**	**Direção**	**Função**
cs_n	1	Entrada	Interface SPI - Inicia comunicação em nível lógico baixo
sck	1	Entrada	Interface SPI - Clock
miso	1	Entrada	Interface SPI - Entrada de dados
start	1	Entrada	Sinal de controle para iniciar o bloco
adc_in	16	Entrada	Entrada paralela de 16 bits
serial_Q	1	Saída	Saída serial do sinal filtrado
Q	16	Saída	Saída paralela do sinal filtrado
mode_select	1	Entrada	Seleciona entre o modo SPI, e o modo paralelo

Tabela 4 – Lista de portas do circuito de camada física UWB e a descrição de suas funcionalidades

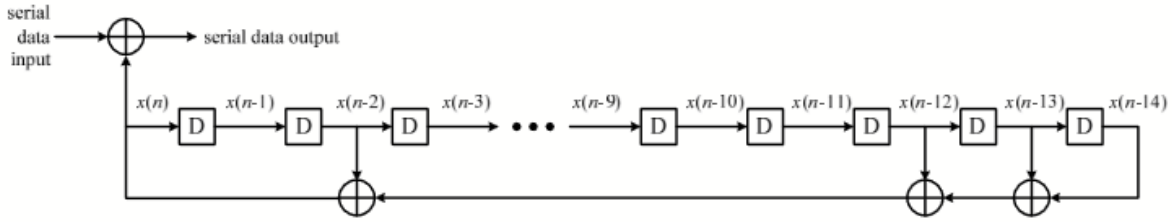


Figura 17 – Circuito para o *scrambler*. (IEEE..., 2012)

uma interface SPI, que possa comunicar com um microcontrolador de modo que se possa enviar mensagens a serem transmitidas e se possa ler mensagens recebidas. Isso é realizado através das FIFOs. O receptor recebe as mensagens por uma entrada serial digital, e após fazer o correto processamento, e a identificação da mensagem, é escrito o cabeçalho da mensagem na "PHR FIFO", e o corpo da mensagem no "PSDU FIFO". Estas FIFOs podem ser lidas pelo controlador SPI, de modo que ao ser solicitado, o controlador lê essas FIFOs, e envia os dados destas via SPI, para um microcontrolador externo.

De forma semelhante, pode-se solicitar o controlador SPI para escrever nas FIFOs do transmissor. Após escrever nestas FIFOs, o transmissor inicia o processamento destas mensagens e após finalizar, ele envia a mensagem formatada corretamente pela saída serial, de modo que é possível observar a saída, e identificar se os dados estão sendo transmitidos corretamente.

Em uma implementação posterior, foi realizada a integração deste bloco diretamente com um processador RISC-V, sem a necessidade do controlador SPI, de forma a ter o circuito com a integração completa.

Em seguida, detalharemos cada um dos circuitos destes sub-blocos.

4.3 *Scrambler*

Para implementar o *scrambler*, deve-se implementar a equação:

$$x[n] = x[n-2] \oplus x[n-12] \oplus x[n-13] \oplus x[n-14] \tag{4.5}$$

Esta equação pode ser implementada por uma LFSR, com 14 registradores, realimentados, conforme representado na imagem 17. Neste caso os bits da mensagem chegam serialmente e uma operação de XOR é feita com x(n), sendo a saída do circuito essa operação de XOR.

Além disso os registradores deste circuito devem ser inicializados com os valores especificados na figura 9, sendo que este valor vem do PHR.

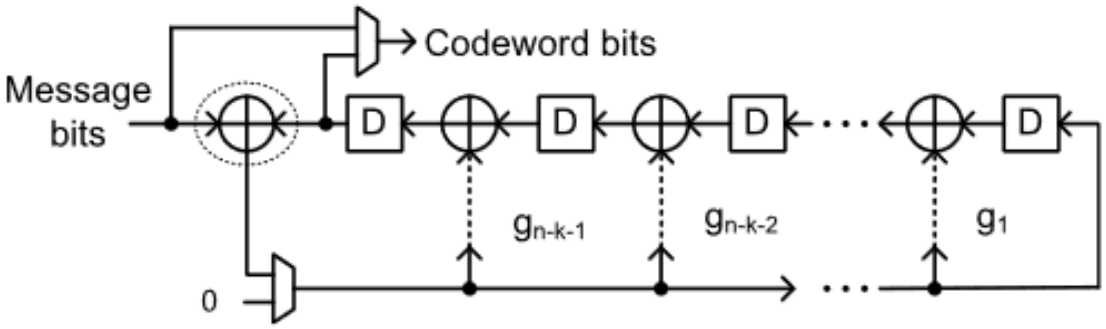


Figura 18 – Codificador BCH (ZHANG, 2016)

4.4 *De-Scrambler*

O *de-scrambler* é implementado com um circuito exatamente igual ao *scrambler*, devendo ser inicializado também da mesma maneira que *scrambler*. Isso acontece pois a operação de *scrambler* aditivo, que é utilizado aqui, tem como inversão exatamente a mesma operação, de modo que é possível reaproveitar o mesmo circuito.

4.5 *Interleaver*

Para o circuito de *interleaver* deve-se embaralhar os bits da mensagem, em blocos de 192 bits segundo a equação:

$$\Pi(n) = nb_s \mod N_I \tag{4.6}$$

Para realizar esta operação foi utilizado 2 bancos de registradores de 192 bits, e esses bancos são interligados conforme a fórmula para realizar o embaralhamento dos bits. Quando o banco é preenchido, o banco de registradores na saída é carregado com os bits embaralhados.

4.6 *De-Interleaver*

O *de-interleaver* faz a mesma operação do *interleaver*, porém na direção contrária, de modo que os bancos de registradores de entrada e saída são invertidos.

4.7 Codificador BCH

Para implementar um codificador BCH, a maneira que costuma se costuma utilizar é codificar o BCH de forma sistemática (equação 2.32). Para realizar a codificação em forma sistemática, é possível implementar um circuito LFSR para realizar esta operação.

Na figura 18, uma estrutura genérica de um codificador BCH é representada. Nesta estrutura a realimentação nos pontos $g_1, g_2, ..., g_{n-k-1}$ são definidas pelo polinômio gerador do BCH. No caso deste trabalho este polinômio é definido na equação 2.5. Isto significa que todos os termos do polinômio que são diferentes de zero, por exemplo x^12 devem se traduzir no circuito em uma conexão, neste exemplo g_12 e os demais termos que não fazem parte do polinômio, devem se traduzir em um circuito aberto, que significa que o circuito não será realimentado em tal ponto.

Este circuito recebe a mensagem de forma serial, e para os primeiros k ciclos, os bits da mensagem são recebidos, e são também colocados na saída do circuito. Após esses ciclos, o resto da divisão polinomial da mensagem por g(x) estará disponível nos registradores deste circuito. Então os *MUX's* do circuitos são chaveados, e a LFSR deste circuito é alimentada com bits zero, e a saída do circuito não é mais a mensagem recebida, mas os bits disponíveis nos registradores. Após n-k ciclos, a codificação está completa.

Para o caso específico deste trabalho, o BCH implementado é o BCH(n=63, k=51, t=2), de modo que o circuito conta com 12 registradores, e a mensagem não codificada tem tamanho 51.

4.8 Decodificador BCH

O decodificador BCH desenvolvido neste trabalho, segue uma arquitetura do tipo *Hard-Decision*. Desta forma, o decodificador funciona em 3 partes: o cálculo das síndromes, o KES e o *Chien Search*. A partir destes 3 blocos, é possível realizar a decodificação.

4.8.1 Cálculo das síndromes

Para calcular as síndromes deve-se calcular os polinômios gerados pela mensagem recebida, e checar o resultado, que se a mensagem não estiver comprometida, deve ser igual a 0. Considerando o polinômio da mensagem de 63 bits, r(x) um polinômio de grau 62, pode-se calcular a síndrome na forma:

$$S_j = r(\alpha^j) = r_{62} \cdot (\alpha^j)^{62} + r_{61} \cdot (\alpha^j)^{61} + ... + r_1 \cdot (\alpha^j) + r_0 \tag{4.7}$$

É possível reorganizar a equação para que seja possível calcular a síndrome de forma serial. Da seguinte forma:

$$S_j = [[[r_{63} \cdot \alpha^j + r_{62}]\alpha^j + r_{61}]\alpha^j + ... + r_1]\alpha^j + r_0 \tag{4.8}$$

Neste caso é possível realizar o cálculo da síndrome pelo circuito na figura 19. Este circuito implementa a equação 4.8. O cálculo é feito serialmente conforme os bits chegam, e

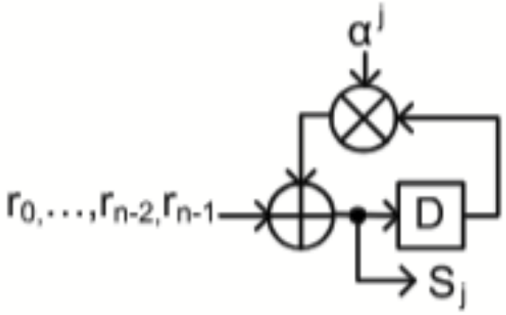


Figura 19 – Circuito de cálculo serial das síndromes. (ZHANG, 2016)

a cada bit, multiplica-se o bit recebido por α^j, e soma-se ao valor total. Ao fim do recebimento dos 63 bits da mensagem, o valor da síndrome estará disponível nos registradores do circuito

Para implementar uma multiplicação por α^j é possível utilizar uma LFSR, que implementa esta operação.

4.8.2 *Key Equation Solver*

Após o cálculo das síndromes, é necessário encontrar os coeficientes do polinômio localizador de erros. Dado que este trabalho visa implementar o BCH(n=63, k=51, t=2), então t=2, e da equação 2.37 podemos derivar:

$$\Lambda(z) = \prod_{t=1}^{2}(1 - X_l z) = \Lambda_2 z^2 + \Lambda_1 z + \Lambda_0 \tag{4.9}$$

Para o algoritmo de Peterson, o coeficiente $\Lambda_0 = 1$ sempre. Deste modo é necessário calcular os coeficientes Λ_1 e Λ_2. Como discutido em seções anteriores, quando t é pequeno, existem maneiras de calcular os coeficientes de forma direta, de modo que pode-se utilizar a equação 4.10 para calcular os coeficientes. Estas equações podem ser otimizadas ainda mais (ZHANG, 2016), se as multiplicarmos por S_1. Fazendo isso, é possível calcular os coeficientes sem precisar se utilizar de blocos de divisão, que é um bloco que consome muita potência e área. Então as equações ficam na forma:

$$\begin{aligned} \Lambda_1 &= S_1^2 \\ \Lambda_2 &= S_3 + S_1 S_2 \end{aligned} \tag{4.10}$$

4.8.3 *Chien Search*

Para realizar o cálculo da raiz do polinômio localizador erros, é possível utilizar um circuito de *Chien Search*. Basicamente o circuito deve fazer uma busca exaustiva para encontrar os pontos onde o polinômio é igual a zero. Na figura 20 é possível observar uma

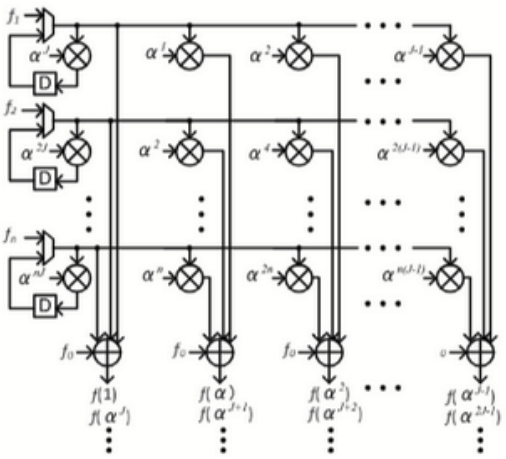


Figura 20 – Circuito *Chien Search*. (ZHANG, 2016)

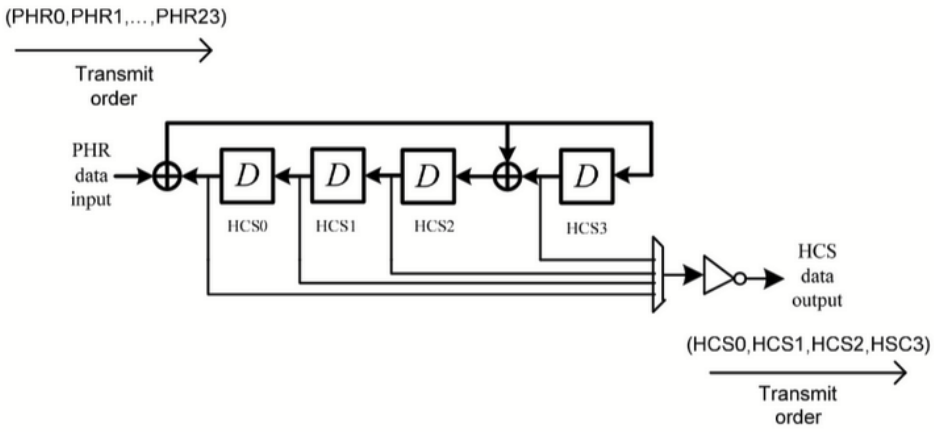


Figura 21 – Circuito LFSR para realizar o HCS. (IEEE…, 2012)

arquitetura de *Chien Search*. Os coeficientes f são injetados no circuito, e para cada possível solução é testado de forma paralela se o resultado para equação é igual a zero. Após o cálculo se descobre as raízes, e aí é possível saber a localização dos erros.

4.9 BCH encurtado para o PHR

Para o PHR utiliza-se um BCH(n=40, k=28, t=2). Este codificador pode ser derivado do BCH(n=63, k=51, t=2), especificado nas seções anteriores, de modo que é possível usar exatamente o mesmo circuito para implementar este BCH encurtado. Isso porque o BCH encurtado pode ser interpretado como o BCH(n=63, k=51, t=2), com a diferença que os bits extras são preenchidos com zeros. Fazendo isso é possível aproveitar os circuitos já descritos.

4.10 *Header Check Sequence*

O HCS pode ser implementado utilizando uma LFSR. Considerando a equação 2.3 especificada pela norma, o circuito recomendado também dentro da norma é o circuito da

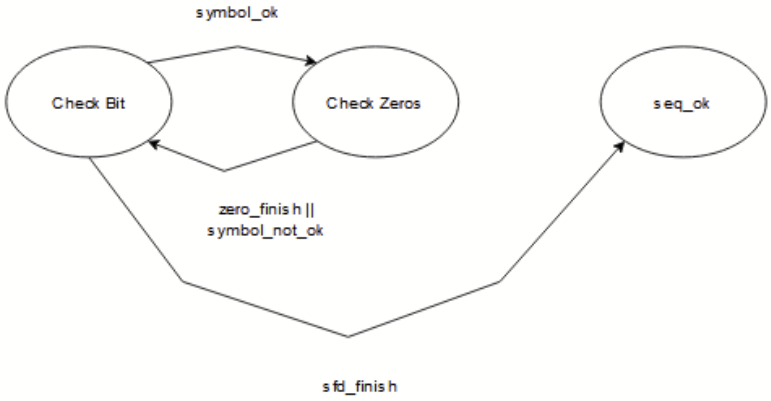


Figura 22 – Máquina de estados para checar sequência do SHR

figura 21. Os bits do PHR chegam serialmente, e após chegar os 24 bits do PHR, os 4 bits de paridade estarão disponíveis nos registradores da LFSR. Os registradores são inicializados em '1' e os bits dos registradores são o HCS negado, de modo que deve se aplicar uma inversora na saída do circuito.

4.11 SHR(Receptor)

Para realizar a checagem da sequência de Kasami, foi utilizado uma máquina de estados para detectar a sequência. A máquina de estados é apresentada na figura 22.

Cada bit Kasami é enviado, seguido de uma sequência de 15 zeros. No circuito proposto, a sequência é armazenada no circuito como uma constante, e os bits são checados no estado *"Check Bit"*, quando o bit é igual ao da sequência o circuito segue para checar a sequência de 15 zeros no estado *"Check Zeros"*, e caso os 15 zeros estejam corretos, segue-se a checagem dos bits. Em qualquer momento que a checagem estiver incorreta, o circuito é reiniciado e os contadores para os bits da sequência Kasami são zerados, reiniciando a contagem. Quando a checagem chega ao final, e é confirmado a sequência o circuito chega no estado *"seq_ok"* e uma *flag* é acionada, sinalizando que a sequência foi identificada. É esta *flag* que inicia o processo de recepção de uma mensagem no receptor.

4.12 SHR(Transmissor)

No transmissor, para realizar o envio da sequência Kasami, esta sequência é armazenada no circuito como uma constante, e com a ajuda de um contador, se acessar os bits da sequência, um a um, até que todos os bits sejam enviados. Esta parte do circuito é controlada pela a FSM do transmissor.

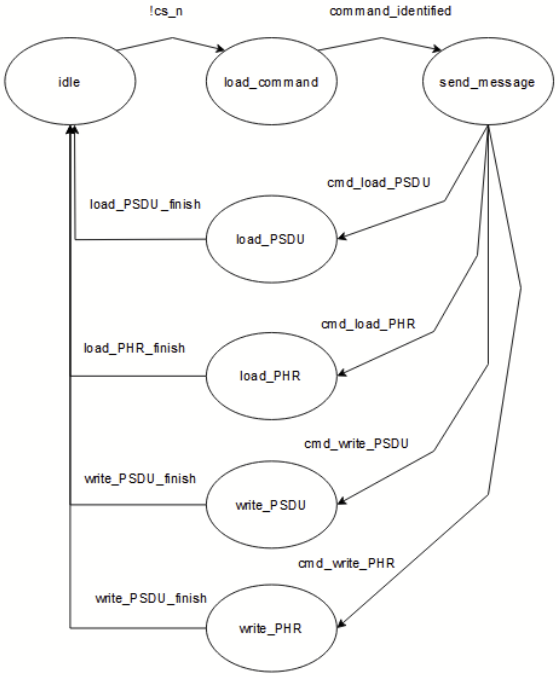


Figura 23 – Máquina de estados do controlador SPI

4.13 Controlador SPI

Para o controlador SPI, um circuito que implementa uma interface SPI foi gerado. Este circuito foi desenvolvido a partir de uma máquina de estados, que implementa a leitura do SPI, e o envio das mensagens via SPI.

Para tanto foram convencionados padrões de comunicação para definir alguns comandos. Os seguintes comandos foram definidos:

1010 0001 => Ler PSDU

1010 0010 => Ler PHR

1010 0011 => Escrever PSDU

1010 0100 => Escrever PSDU

Estes comandos basicamente definem qual FIFO será escrita ou lida. No caso da leitura, são lidas as FIFOs do receptor. No caso da escrita, são escritas as FIFOs do transmissor. Deste modo é possível checar se o circuito do receptor recebeu a mensagem corretamente, e é possível escrever uma mensagem para ser enviada pelo transmissor.

Na figura 23 a máquina de estados do controlador SPI é representada. A máquina parte de um estado ocioso *"idle"*. Quando a entrada *cs_n* é levada para um nível lógico baixo, a máquina de estados vai para o estado *"load_command"* onde os bits provenientes da entrada *"mosi"* serão lidos e armazenados em uma FIFO de 8 bits. Quando esta FIFO é preenchida com algum dos comandos possíveis, a máquina passa para o estado *"send_message"*. Neste

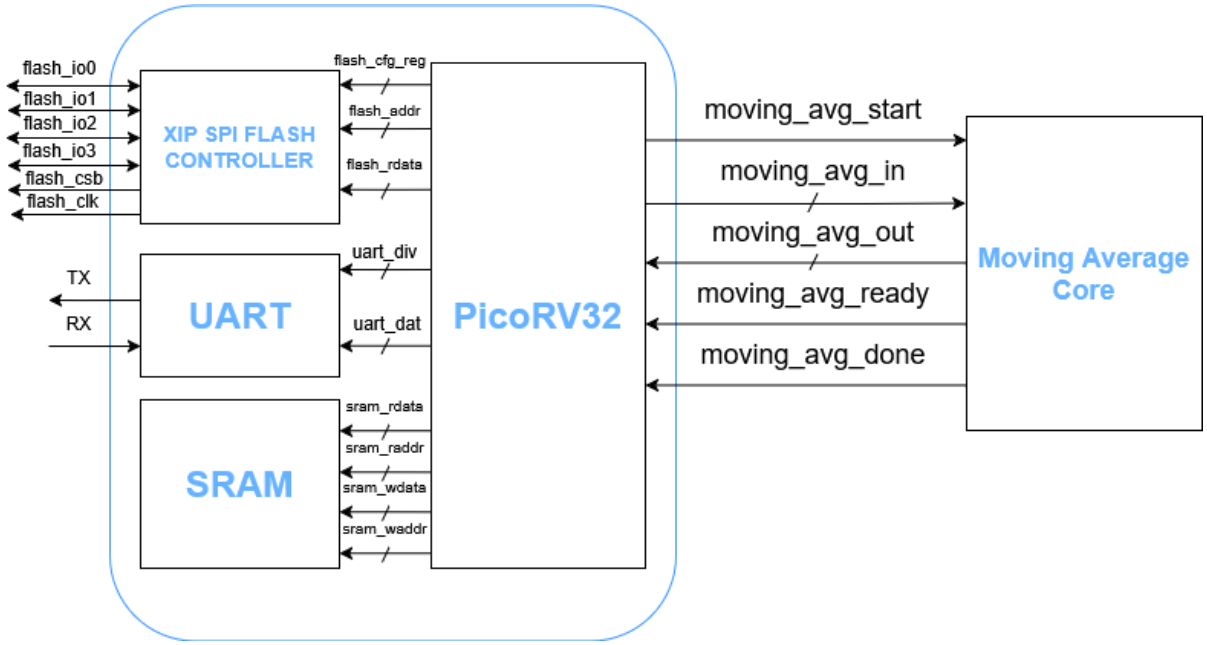


Figura 24 – Diagrama de blocos da integração entre o RISC-V e o filtro média móvel.

estado, é selecionado o comando correspondente e a máquina segue para um dos estados de execução dos comandos.

Nos estados de leitura, o controlador responde na saída *"miso"* mandando serialmente os bits da FIFO solicitada. Já no caso dos comandos de escrita o controlador começa a receber os bits que se quer escrever na FIFO solicitada, e estes bits são escritos até que a FIFO esteja completamente preenchida. Após os comandos finalizarem, eles retornam para o estado *"idle"*.

4.14 Integração com RISC-V

Os blocos implementados serão integrados com um processador RISC-V. A ideia é implementar os blocos como coprocessadores, que fazem suas operações enquanto o processador RISC-V executa um programa, de modo que o processador possa acessar os dados depois de prontos, sem precisar executar as operações em tempo de programa, mas de forma paralela.

Para integrar o RISC-V com o filtro média móvel, uma arquitetura é proposta e apresentada na figura 24. Para esta arquitetura será utilizado o projeto PicoSoC de código aberto, que conta com um controlador de memória flash, que utiliza a interface quad SPI, conta com uma interface UART e faz interface com o bloco de filtro média móvel. Para controlar o filtro média móvel, o sinal *"moving_avg_start"* é utilizado para acionar o filtro, iniciando o cálculo. Os sinais *"moving_avg_in"* e *"moving_avg_out"* são os dados de entrada do filtro e o resultado do filtro que é sua saída. Além disso, os sinais *"moving_avg_ready"* e *"moving_avg_done"* são os sinais que sinalizam o fim de um cálculo, e o fim de todas as

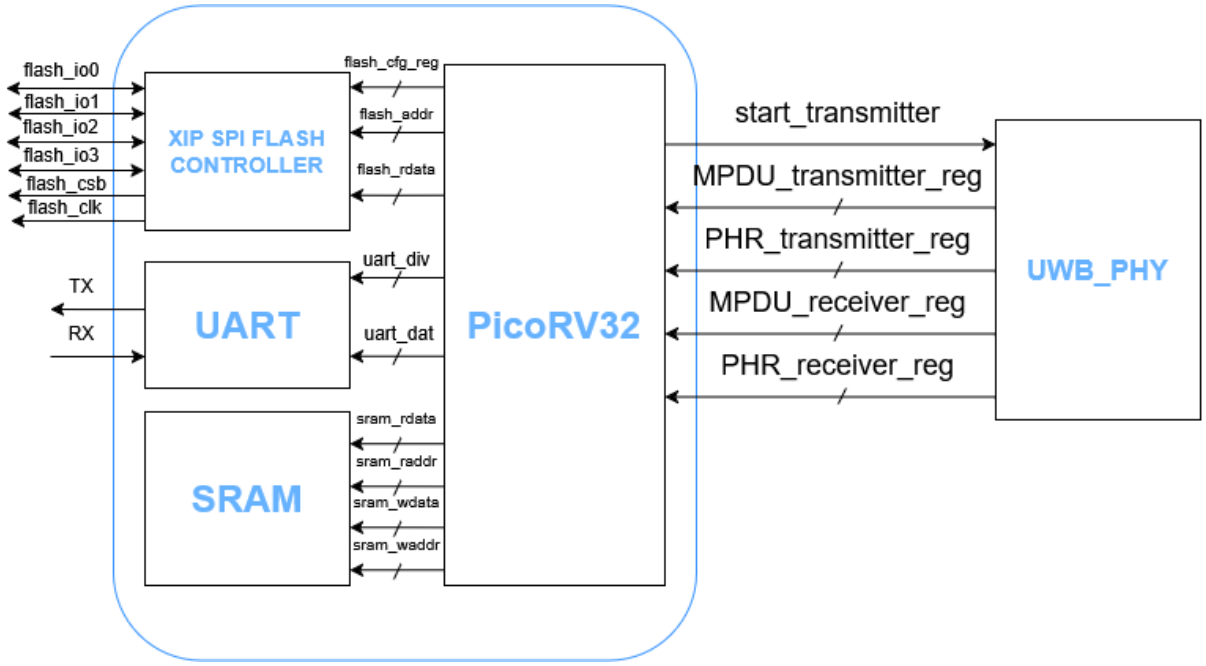


Figura 25 – Diagrama de blocos da integração entre o RISC-V e a implementação da camada física para o UWB.

operações respectivamente.

Estes sinais são controlados pela central de processamento, o *PicoRV32*, através de registradores. Para que seja possível mapear estes sinais para software, estes sinais são mapeados para registradores, que por sua vez tem endereços mapeados em software. Para isto foi utilizado um registrador para entrada do filtro, um para a saída, e um registrador de configuração que controla o sinal de *"start"* assim como dá acesso às *flags* de *"ready"* e *"done"* do bloco.

De forma semelhante, o bloco controlador de memória flash e o bloco de interface UART são controlados por registradores, que são mapeados em endereços de memória. Para o controlador de memória flash, temos um registrador de configuração *"flash_cfg_reg"* que controla diferentes configurações para o controlador. O registrador *"flash_addr"* determina o endereço do qual se quer obter dados na memória flash e finalmente o registrador *"flash_rdata"* armazena os dados obtidos da memória.

Para a interface UART, dois registradores são usados. O registrador *"uart_div"* configura o *baud rate* da interface UART, por meio de um divisor de *clock*. Já o registrador *"uart_dat"* contém os dados que são enviados por meio da interface UART.

Além disso temos o bloco de SRAM que é um bloco de memória RAM, que tem também sinais de endereço e de dados. Para este trabalho, este bloco não foi sintetizado em hardware, sendo esta uma tarefa para ser realizada em trabalhos futuros, visto que é necessário utilizar blocos de IP (*Intelectual Property*) para realizar esta tarefa.

A integração do RISC-V com a camada física para o UWB é realizada em uma outra

arquitetura, semelhante ao caso do filtro média móvel. Para o caso da camada física UWB, um sinal de *"start"* é utilizado para iniciar o transmissor, e 4 registradores são utilizados para comunicar dados, 2 para o transmissor e 2 para o receptor, sendo que para cada um há um registrador para o cabeçalho PHR e um registrador que contém o MPDU. A partir destes registradores pode-se escrever o que se quer enviar pelo transmissor e pode-se checar os dados que chegam no receptor.

5 Resultados e Discussão

Para este trabalho foram implementados os blocos de média móvel simples, e de processamento em banda base para um transmissor e um receptor que implementa o padrão IEEE 802.15.6. Após implementação dos blocos individuais, o transmissor e o receptor foram integrados com um processador RISC-V. O filtro média móvel foi implementado utilizando tecnologia de 180nm e além desta implementação, pode ser vista no apêndice uma implementação deste filtro em FPGA e sua integração com o RISC-V em FPGA. Para a camada física para UWB, foi utilizado tecnologia de 22 nm, bem como para a integração com o RISC-V.

5.1 Média móvel simples

O circuito de média móvel simples foi implementado em ASIC. Para esta implementação foi utilizada a tecnologia CMOS de 180nm. O circuito foi implementado em *Verilog*, e para realizar a síntese física do circuito foi utilizada a suíte de ferramentas de desenvolvimento de circuitos integrados digitais da *Cadence*.

5.1.1 Simulação do código HDL

Para validar a arquitetura do filtro o código em *Verilog* foi testado por meio de simulações utilizando o *Xcelium*. As simulações foram realizadas utilizando um *testbench*, que checa os resultados obtidos pela arquitetura e os confronta com os resultados obtidos em um modelo em software.

Para o modelo em software, foi criado um *script* em *Python* que realiza os cálculos que o filtro faria, lendo as entradas de um arquivo de texto com valores de teste. A partir destes valores e o código em *Python* é possível obter um arquivo com os resultados esperados.

Abaixo, um pedaço do código de *testbench* é apresentado:

```
initial
begin
    $readmemh("mem.mem", read_data);
    write_data = $fopen("out.mem");


    A = 8'b0; rst = 1'b1; start = 1'b0;
    #period;
    A = 8'b0; rst = 1'b0; start = 1'b0;
    #period;
```

```
11
12      for(i=0;i<500;i=i+1)
13      begin
14          // idle
15          A= read_data[i]; rst = 1'b0; start = 1'b1;
16          #period;
17          // Read
18          start = 1'b0;
19          #period;
20          // Register
21          #period;
22          // New Mean
23          #period;
24          $fdisplay(write_data, "%h", Q);
25      end
26      $fclose(write_data);
27  end
```

Este código consiste na leitura do arquivo com valores de entrada para o circuito e após isso realiza-se os cálculos de média sendo o valor armazenado em um arquivo. O comando *"readmemh"*, na primeira linha, recebe os valores de entrada. Na segunda linha, o arquivo de saída é aberto utilizando o comando *"fopen"*. A partir daí, realiza-se o um *"reset"* para iniciar o sistema e em seguida um laço de repetição é iniciado, no qual se lê cada um dos elementos presentes no arquivo de entrada, e com o comando *"fdisplay"* o resultado é salvo no arquivo de saída.

Com esses arquivos é possível comparar o resultado obtido no modelo em *Python* e o resultado obtido com o código em *Verilog*. A partir de um *script* em *Python* pode-se checar se os valores são iguais.

Para a arquitetura do filtro testado neste trabalho, os valores obtidos para o código em *Verilog* foram exatamente iguais aos valores obtidos em *Python*. Além disso, um diagrama foi gerado para demonstrar o resultado do filtro, em relação a entrada. Na figura 26 pode-se ver o resultado do filtro. Em laranja, a linha sólida representa a entrada do filtro e a linha preta tracejada é o resultado do filtro. É possível ver a ação do filtro em eliminar ruídos de alta frequência no sinal.

5.1.2 Síntese Física

Na figura 27 é possível ver o layout do circuito implementado em tecnologia CMOS. A forma do layout foi escolhida para ser retangular e mais fina para ser acomodada junto aos outros blocos analógicos que foram enviados para fabricação. O filtro foi posicionado próximo aos *PADs* e o posicionamento dos pinos tem o intuito de tornar a conexão para os *PADs* mais fácil. Foram utilizadas 4 camadas de metal, sendo as duas maiores utilizadas para a alimentação. As camadas superiores foram reservadas para o topo do chip.

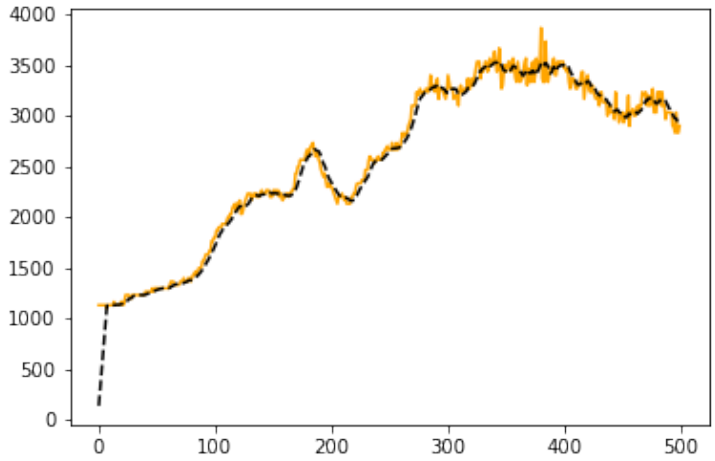


Figura 26 – Simulação do filtro média móvel. O eixo x representa o tempo da simulação e o eixo y representa o valor de entrada e saída.

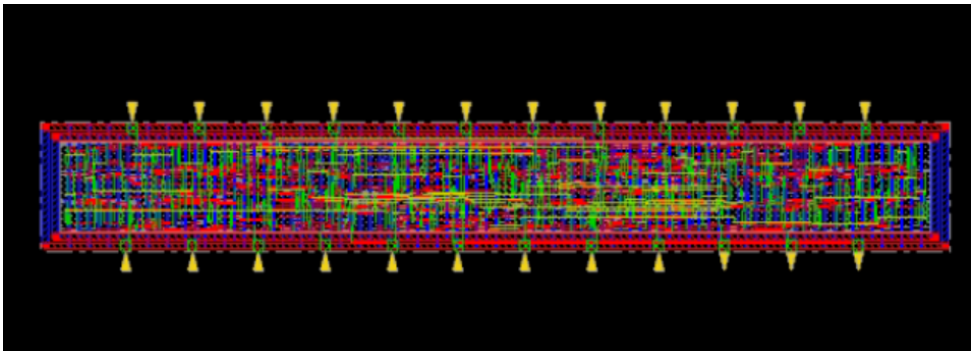


Figura 27 – Layout do bloco implementado em tecnologia CMOS 180nm.

timeDesign Summary

Setup mode	all	reg2reg	default
WNS (ns):	93,747	93,747	97,763
TNS (ns):	0,000	0,000	0,000
Violating Paths:	0	0	0
All Paths:	719	531	212

Figura 28 – Resultado do *timing* para o filtro média móvel. Tempo em nanossegundos e a frequência do *clock* simulado é de 1 MHz.

Já na figura 28 é possível ver o resultado da análise de *timing* do circuito. Nos resultados mostrados é possível ver que nenhuma violação de *timing* foi encontrada, e o circuito pôde ser sintetizado com uma boa folga de *timing*.

Na tabela 5 informações de área, consumo de células e potência são apresentadas. Os resultados obtidos são razoáveis com um consumo baixo de potência simulado em 1MHz utilizando o arquivo VCD da simulação para realizar a estimativa de potência. O circuito foi simulado utilizando um *testbench* que simula o comportamento real do circuito, e a atividade

Tabela 5 – Consumo de energia, área e células para o filtro média móvel (1 MHz)

Área	16 758 μm^2
Potência Total	13,170 μW
Potência Dinâmica	13,100 μW
Potência Estática	0,070 μW

das células obtidas nesta simulação a 1MHz foi utilizada para gerar o VCD utilizado para realizar uma simulação de potência.

Na tabela 6 uma comparação entre área e potência entre o circuito implementado neste trabalho e outras implementações de filtros FIR disponíveis na literatura é apresentada. É possível observar que o circuito proposto, obteve o menor consumo de potência e o menor GE entre os circuitos comparados. Isso se deve ao fato da arquitetura simplificada, aplicada para o circuito desenvolvido. Como não foi utilizado circuitos multiplicadores ou divisores, e foi utilizado uma quantidade baixa de *taps*, o circuito proposto tem uma das melhores performances em tamanho e potência. É importante notar, no entanto, que os circuitos comparados permitem uma maior flexibilidade do filtro, que o circuito proposto não tem, por ser uma arquitetura simplificada. Essa arquitetura porém, pode ser bem útil para circuitos que necessitem de baixo consumo de energia.

Tabela 6 – Comparação de área e potência com outras implementações de filtro FIR

	Taps	Tecnologia	Área (μm^2)	GE adap. ($\frac{mm^2}{\mu m^2}$)	Potência (mW)
(ALJUFFRI et al., 2015) (Sequencial)	8	150nm	30379	1,35	2,61
(ALJUFFRI et al., 2015) (Paralelo)	8	150nm	58368	2,59	0,17
(ANNANGI; PULI, 2017)	144	45nm	79220	39,12	20
(XU et al., 2013)	16	65nm	11335	2,68	1,30
(LOU; YE, 2017)	36	65nm	14810	3,50	4,43
Santos, 2025	8	180 nm	16758	0,52	0,013

5.2 Processador banda base UWB

A implementação do processador banda base UWB foi realizada em *Verilog* e seguiu-se o fluxo digital de desenvolvimento utilizando tecnologia de 22 nm para realizar a síntese do circuito. A seguir é detalhado os resultados de simulação do código em *Verilog*.

5.2.1 Simulação

Para realizar a simulação do código em *Verilog* foi primeiro gerado um modelo em python com o comportamento esperado do sistema. A partir do modelo é possível obter

mensagens sem o processamento em base banda, e o resultado do processamento, com todas as operações realizadas.

Com o modelo em *Python* pode-se avaliar o funcionamento do código em *Verilog* e verificar se o funcionamento do circuito é condizente com o que se espera. Para fazer esta checagem, utilizou-se um *testbench* para realizar a simulação.

Primeiramente no código do *testbench*, define-se a entrada do PHR e do PSDU, e o resultado esperado, após o processamento, como no exemplo a seguir:

```
reg [23:0] phr_transmitter = 24'b111000000000000000000100;
reg [39:0] phr_transmitter_expected =
   40'b1011111100101011111000000000000000000100;
reg [2119:0] psdu_transmitter = 2120'b01111 ... 0;
reg [2623:0] psdu_transmitter_expected = 2624'b111100 ... 0;
```

Após isso, o teste inicia testando primeiro o transmissor. O *"reset"* é acionado e o PHR é enviado via SPI para o sistema, utilizando um código como a seguir

```
    for(i = 7; i>=0; i=i-1) begin
      serial_input = 0; mosi = cmd_write_phr[i];
      #10;
    end

    #10;

    for(i = 0; i<24; i=i+1) begin
      serial_input = 0; mosi = phr_transmitter[i];
      #10;
    end
```

De forma semelhante, envia-se o PSDU. Após isso o transmissor iniciará a codificação, e quando terminar, começa a enviar o sinal codificado. Dessa forma, o *testbench* espera que o sistema finalize a codificação e inicie a transmissão.

Quando o sistema inicia a transmissão, a transmissão é iniciada pelo envio do SHR. O *testbench* fica preso em um laço de repetição que espera o SHR. Enquanto o SHR não é enviado, o código fica preso no laço.

Logo após isso, o sistema transmite o PHR e o PSDU. O *testbench* então armazena eles em uma variável como no código a seguir:

```
    for(i=0; i<40; i=i+1) begin
      phr_fifo[i] = serial_output;
      #10;
```

```
4       end
5
6       for(i=0; i<126; i=i+1) begin
7         psdu_fifo[i] = serial_output;
8         #10;
9       end
```

Neste código, *"serial_output"* é a saída do transmissor e esta saída é coletada de forma serial, sendo os bits coletados uma vez por ciclo e armazenados nas variáveis *"phr_fifo"* e *"psdu_fifo"*.

Após isso os resultados são imprimidos na tela, e é possível checar se o resultado está coerente com o modelo em *Python*.

Depois o *testbench* segue para o teste do receptor, que segue uma lógica semelhante ao teste do transmissor, sendo a diferença que a entrada do receptor é realizada através da entrada serial, e os resultados são coletados através do SPI. Ao fim do teste os resultados são imprimidos na tela e é possível checar se o teste foi bem-sucedido.

Os resultados dos testes para a arquitetura propostas foram bem-sucedidos e demonstraram ser iguais ao modelo em *Python*.

5.2.2 Síntese física

Este bloco foi implementado em tecnologia CMOS 22 nm, utilizando *Verilog*, e usando a suíte de ferramenta de desenvolvimento de circuitos integrados digitais da *Cadence*.

Na Figura 29 é possível ver o layout final do circuito implementado em tecnologia CMOS. A forma do layout foi escolhida inicialmente para ser um quadrado, porém pode ser alterada posteriormente, a depender do entorno do chip e o posicionamento de outros blocos. Foram utilizadas 9 camadas de metal, sendo as 2 maiores reservadas para o anel de alimentação, sendo utilizados as larguras recomendadas pelo PDK.

Na Figura 30 é possível ver o resultado de *timing* do circuito. Nenhuma violação foi encontrada, e o circuito tem uma folga positiva de *timing* com uma boa margem. Isso é devido a escolha da tecnologia combinada a um *clock* não muito alto, que é o *clock* requerido para o modo de transmissão obrigatório da norma.

Finalmente, na tabela 7 é possível ver os valores de consumo de área e de potência. É possível ver que o circuito tem a maior parte da potência dissipada estaticamente, isso acontece pois a frequência de operação não é muito alta, visto que pode-se atingir a faixa de GHz nesta tecnologia. Para realizar o cálculo de potência foi utilizado um arquivo de chaveamento VCD, que foi obtido por meio das simulações com *testbench*, semelhante ao caso do filtro. Os resultados obtidos serão comparados com valores de implementações disponíveis na literatura a seguir.

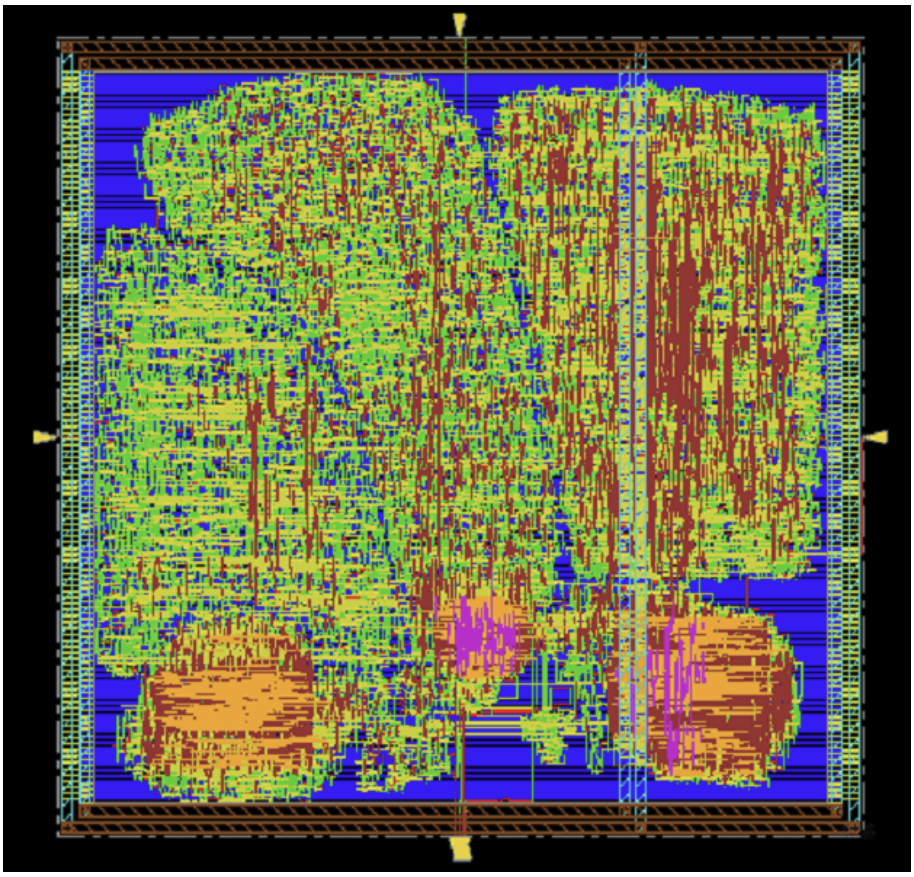


Figura 29 – Layout final do processador em banda base

timeDesign Summary

Setup mode	all	default	In2Out	In2Reg	Reg2Out	Reg2Reg
WNS (ns):	2049,4	0,000	N/A	2049,8	N/A	2049,4
TNS (ns):	0,000	0,000	N/A	0,000	N/A	0,000
Violating Paths:	0	0	N/A	0	N/A	0
All Paths:	8599	0	N/A	8327	N/A	8585
user_tt_st_typ	2049,9	0,000	N/A	2050,2	N/A	2049,9
	0,000	0,000	N/A	0,000	N/A	0,000
	0	0	N/A	0	N/A	0
	8599	0	N/A	8327	N/A	8585
user_ss_ht_cworst	2049,4	0,000	N/A	2049,8	N/A	2049,4
	0,000	0,000	N/A	0,000	N/A	0,000
	0	0	N/A	0	N/A	0
	8599	0	N/A	8327	N/A	8585
user_ss_ht_rcworst_T	2049,4	0,000	N/A	2049,8	N/A	2049,4
	0,000	0,000	N/A	0,000	N/A	0,000
	0	0	N/A	0	N/A	0
	8599	0	N/A	8327	N/A	8585

Figura 30 – Resultado de *timing* do circuito. Tempo em nanossegundos e a frequência do *clock* simulado é de 487,5 kHz.

Tabela 7 – Consumo de energia, área e células

Área	27 449 μm^2
Potência Total	108,842 μW
Potência Dinâmica	2,941 μW
Potência Estática	105,901 μW

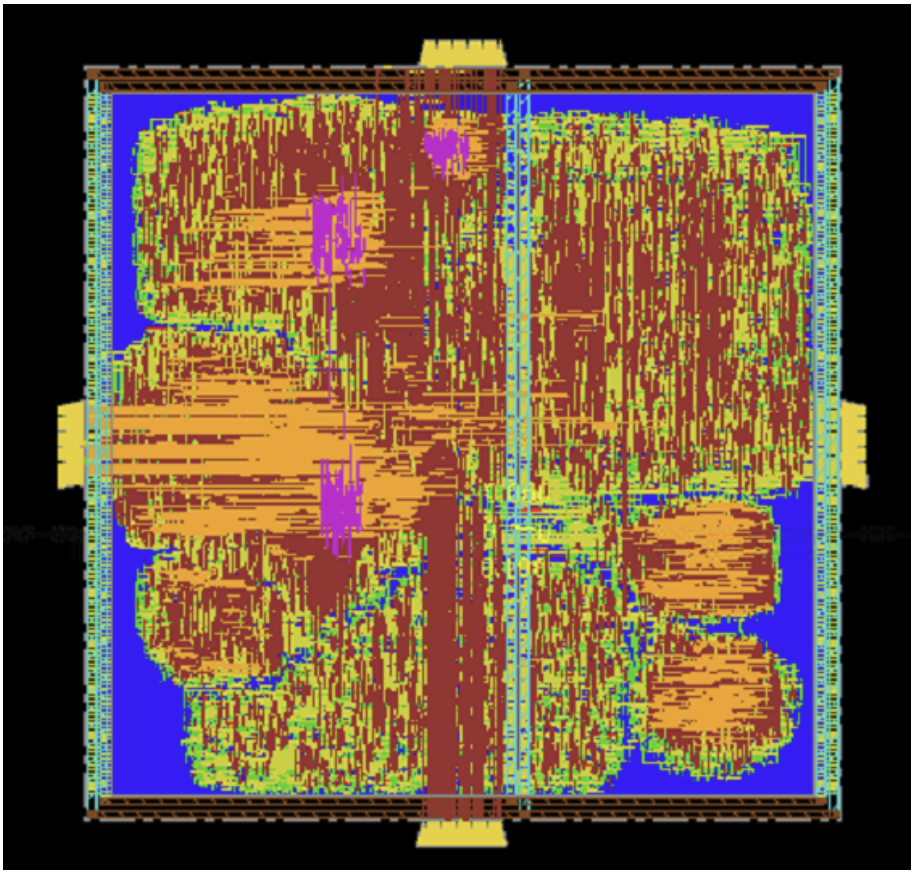


Figura 31 – Layout final do circuito

5.2.3 Integração com RISC-V

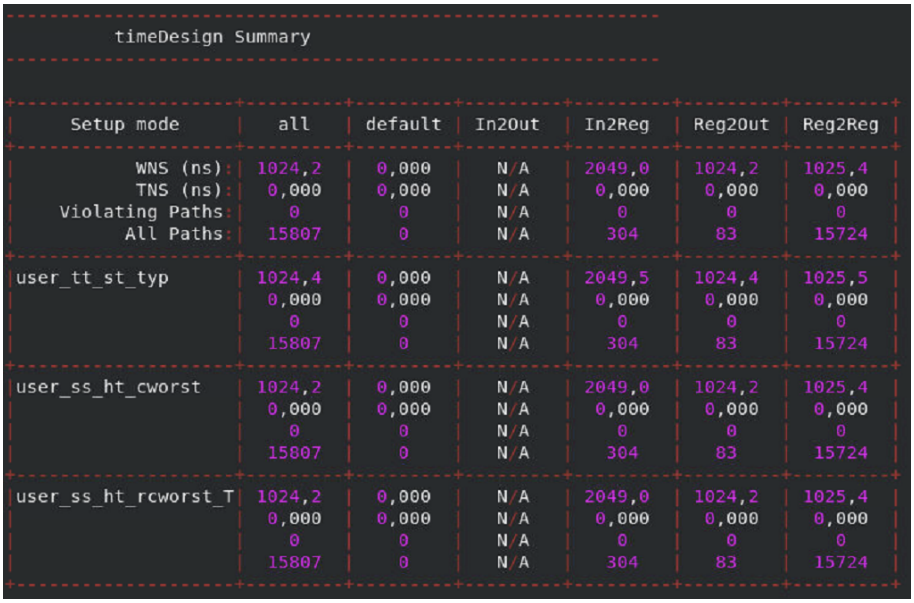
Por fim, foi integrado o bloco de processamento em banda base com o *PicoRV32*, de modo que o bloco se comporte como um coprocessador, junto com o RISC-V. Para isto, integrou-se o circuito por meio de registradores, e foi realizado o desenvolvimento do *firmware* utilizando código em C, para acessar os registradores, controlar o processador base banda e obter os resultados.

Os mesmos dados do modelo em *Python* foram utilizados para testar este bloco, sendo que foi obtido sucesso na simulação deste circuito.

Este bloco foi implementado em tecnologia CMOS 22 nm, utilizando *Verilog*, e usando a suíte de ferramenta de desenvolvimento de circuitos integrados digitais da *Cadence*.

Na Figura 31 é possível ver o layout final do circuito. Assim como o bloco anterior a forma do layout foi escolhida inicialmente para ser um quadrado, porém pode ser alterada posteriormente. Também foram utilizadas 9 camadas de metal, sendo as 2 maiores reservadas para o anel de alimentação.

Na Figura 32 é possível ver os resultados de *timing* do circuito. A partir destes resultados é possível ver que não houve nenhuma violação de *timing*. A folga é positiva e foi obtida uma boa margem na faixa de milhares de nanossegundos. Isso se deve ao fato da



timeDesign Summary

Setup mode	all	default	In2Out	In2Reg	Reg2Out	Reg2Reg
WNS (ns):	1024,2	0,000	N/A	2049,0	1024,2	1025,4
TNS (ns):	0,000	0,000	N/A	0,000	0,000	0,000
Violating Paths:	0	0	N/A	0	0	0
All Paths:	15807	0	N/A	304	83	15724
user_tt_st_typ	1024,4	0,000	N/A	2049,5	1024,4	1025,5
	0,000	0,000	N/A	0,000	0,000	0,000
	0	0	N/A	0	0	0
	15807	0	N/A	304	83	15724
user_ss_ht_cworst	1024,2	0,000	N/A	2049,0	1024,2	1025,4
	0,000	0,000	N/A	0,000	0,000	0,000
	0	0	N/A	0	0	0
	15807	0	N/A	304	83	15724
user_ss_ht_rcworst_T	1024,2	0,000	N/A	2049,0	1024,2	1025,4
	0,000	0,000	N/A	0,000	0,000	0,000
	0	0	N/A	0	0	0
	15807	0	N/A	304	83	15724

Figura 32 – Resultados de *timing* do *picosoc*. Tempo em nanossegundos e a frequência do *clock* simulado é de 487,5 kHz.

Tabela 8 – Comparação de potência entre transceptores UWB disponíveis na literatura

	Frequência	Tecnologia	Tensão da Fonte	Potência Total	η
Manchi, 2017	487,5 kHz	90 nm	0,9 V	146 μW	35,44
El-Mohandes, 2018	6 MHz	130 nm	1,2 V	402,54 μW	7.50
Bachmann, 2014	24 MHz	40 nm	0,74 V	140 μW	2,13
Chen, 2013 (Apenas transmissor)	562.5 KHz	130 nm	1 V	69,5 μW	1003,8
UWB SPI	487,5 kHz	22 nm	0,8 V	108,64 μW	279,08
PicoSoc + UWB	487,5 kHz	22 nm	0,8 V	155,61 μW	245,14

tecnologia utilizada e o *clock* que foi simulado em 487,5 kHz, que permitiu uma boa margem, isto porque a tecnologia permite alcançar a faixa de GHz.

Na tabela 8 é possível ver um comparativo do processador banda base para transceptor UWB implementado, e outros exemplos da literatura. Na tabela é possível ver valores de consumo de potência, frequência do *clock* e nodo da tecnologia utilizada, assim como o valor da potência normalizada η. Os circuitos encontrados na literatura apresentam tecnologias diferentes e para realizar uma comparação mais justa é utilizada a potência normalizada, que pode dar uma intuição da eficiência no consumo de potência do circuito. A partir desta medida, é possível ver que a arquitetura proposta não está entre as mais eficientes energeticamente. As arquiteturas mais eficientes segundo essa medida são as arquiteturas de (EL-MOHANDES; SHALABY; SAYED, 2018) e (BACHMANN et al., 2014). Estas arquiteturas operam em frequências mais altas e tem um consumo mais elevado de área. Deste modo, dado a complexidade do circuito e a velocidade, estas se apresentam como arquiteturas mais eficientes energeticamente, segundo a medida de potência normalizada.

Tabela 9 – Comparação de consumo de área entre transceptores UWB disponíveis na literatura

	Area (μm^2)	GE adaptado ($\frac{mm^2}{\mu m^2}$)
Manchi, 2017	940 000	116,04
El-Mohandes, 2018	807 000	47,75
Bachmann, 2014	200 000	125,00
Chen, 2013 (Apenas transmissor)	16 000	0,94
UWB SPI	27 449	56,71
Picosoc + UWB	44 760	92,48

Na Tabela 9 é possível ver um comparativo da área consumida para cada uma das implementações dos transceptores, segundo o padrão IEEE 802.15.6. É possível ver também o valor de GE adaptado que pe proporcional a quantidade de transistores utilizados nos circuitos. A partir desta medida, é possível observar que o circuito com menor complexidade é o de (CHEN et al., 2013). Que é esperado, visto que este circuito implementa apenas o transmissor. (EL-MOHANDES; SHALABY; SAYED, 2018) apresenta também uma complexidade baixa, e após este os circuitos menos complexos são o deste trabalho. Como esperado, o circuito de camada física, junto ao *piocosoc* tem uma complexidade maior, embora não seja o mais complexo da tabela.

6 Conclusão

Para este trabalho foram implementados circuitos para realizar a função de DSP dentro do projeto Cedro, que é o projeto de um SoC RFID híbrido de UWB/UHF. Foi implementado um filtro média móvel e um processador de banda base para implementar o padrão IEEE 802.15.6 para a comunicação UWB em redes WBAN. Cada um destes circuitos foi integrado separadamente a um processador RISC-V, comprovando a possibilidade de utilizar estes circuito junto a uma CPU RISC-V, que permite o controle do SoC RFID como um todo.

O circuito de filtro média móvel foi implementado tanto em FPGA quanto em tecnologia CMOS 180nm. Em FPGA, foi possível provar a integração do filtro média móvel com o RISC-V. Além disso, foi comparado o consumo de área e energia do circuito proposto com circuitos semelhantes disponíveis na literatura, de modo que os resultados demonstram que o circuito proposto tem uma boa eficiência energética.

O circuito de processamento em banda base para implementar o padrão IEEE 802.15.6 também foi implementado, nesse caso apenas em tecnologia CMOS 28 nm, e também foi comprovada a integração com o processador RISC-V. Comparações foram feitas com circuitos disponíveis na literatura, e é possível concluir que o circuito proposto tem uma eficiência energética inferior em comparação a outras arquiteturas e tem espaço para otimização. No entanto, se demonstrou um circuito com baixa complexidade em relação aos trabalhos na literatura, que implica que este circuito pode ser implementado utilizando pouca área de silício.

Para trabalhos futuros é possível realizar a integração do filtro média móvel e o processador banda base em único circuito integrado ao RISC-V e implementar também a integração desse circuito com IPs de memória SRAM apropriados a tecnologia CMOS, visando obter uma implementação mais completa.

Referências

ALJUFFRI, A. A.; ALNAHDI, M. M.; HEMAID, A. A.; ALSHAALAN, O. A.; BENSALEH, M. S.; OBEID, A. M.; QASIM, S. M. ASIC realization and performance evaluation of scalable microprogrammed FIR filters using Wallace tree and Vedic multipliers. In: IEEE. 2015 IEEE 15th International Conference on Environment and Electrical Engineering (EEEIC). 2015. P. 1995–1998. Citado nas pp. 17, 66.

ANNANGI, S.; PULI, R. ASIC implementation of efficient 16-parallel fast FIR algorithm filter structure. In: IEEE. 2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT). 2017. P. 1–5. Citado nas pp. 17, 66.

BACHMANN, C.; VAN SCHAIK, G.-J.; BUSZE, B.; KONIJNENBURG, M.; ZHANG, Y.; STUYT, J.; ASHOUEI, M.; DOLMANS, G.; GEMMEKE, T.; DE GROOT, H. 10.6 A 0.74 V 200μW multi-standard transceiver digital baseband in 40nm LP-CMOS for 2.4 GHz Bluetooth Smart/ZigBee/IEEE 802.15. 6 personal area networks. In: IEEE. 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). 2014. P. 186–187. Citado nas pp. 36, 71.

BILLINGS, S. A. **Nonlinear system identification: NARMAX methods in the time, frequency, and spatio-temporal domains**. John Wiley & Sons, 2013. Citado na p. 17.

CHAUHAN, M.; THORWE, P.; MUKHERJEE, M. J.; RAO, Y. S. Sensor Data Analysis Using Moving Average Filter and 256-Point FFT for Wireless Sensor Networks. In: IEEE. 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT). 2018. P. 1–7. Citado na p. 17.

CHEN, M.; HAN, J.; FANG, D.; ZOU, Y.; ZENG, X. An ultra low-power and area-efficient baseband processor for WBAN transmitter. In: IEEE. 2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference. 2013. P. 1–4. Citado nas pp. 36, 72.

IEEE Standard for Local and metropolitan area networks - Part 15.6: Wireless Body Area Networks. **IEEE Std 802.15.6-2012**, p. 1–271, 2012. DOI: 10.1109/IEEESTD.2012.6161600. Citado nas pp. 20–22, 53, 57.

KAESLIN, H. **Digital integrated circuit design: from VLSI architectures to CMOS fabrication**. Cambridge University Press, 2008. Citado na p. 47.

LIU, X.; ZHENG, Y.; ZHAO, B.; WANG, Y.; PHYU, M. W. An ultra low power baseband transceiver IC for wireless body area network in 0.18-

\

mu *mCMOStechnology*. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 19, n. 8, p. 1418–1428, 2010. Citado na p. 47.

LOU, X.; YE, W. Low complexity and low power multiplierless FIR filter implementation. In: IEEE. 2017 IEEE 12th International Conference on ASIC (ASICON). 2017. P. 596–599. Citado na p. 66.

MAGSI, H.; SODHRO, A. H.; CHACHAR, F. A.; ABRO, S. A. K. Analysis of signal noise reduction by using filters. In: IEEE. 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET). 2018. P. 1–6. Citado na p. 17.

MANCHI, P. K.; PAILY, R.; GOGOI, A. K. Low-Power Digital Baseband Transceiver Design for UWB Physical Layer of IEEE 802.15.6 Standard. **IEEE Transactions on Industrial Informatics**, v. 13, n. 5, p. 2474–2483, 2017. DOI: 10.1109/TII.2017.2717882. Citado nas pp. 19, 37.

MATHEW, P.; AUGUSTINE, L.; KUSHWAHA, D.; VIVIAN, D.; SELVAKUMAR, D. Hardware implementation of NB PHY baseband transceiver for IEEE 802.15. 6 WBAN. In: IEEE. 2014 International Conference on Medical Imaging, m-Health and Emerging Communication Systems (MedCom). 2014. P. 64–71. Citado na p. 36.

EL-MOHANDES, A. M.; SHALABY, A.; SAYED, M. S. Efficient Low-Power Digital Baseband Transceiver for IEEE 802.15. 6 Narrowband Physical Layer. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 26, n. 11, p. 2372–2385, 2018. Citado nas pp. 37, 71, 72.

EL-MOHANDES, A. M.; SHALABY, A.; SAYED, M. S. Robust low power NB PHY baseband transceiver for IEEE 802.15. 6 WBAN. In: IEEE. 2015 27th International conference on Microelectronics (ICM). 2015. P. 91–94. Citado na p. 37.

MORALES-MENDOZA, L. J.; SHMALIY, Y.; IBARRA-MANZANO, O. G.; ARCEO-MIQUEL, L.; MONTIEL-RODRIGUEZ, M. Moving average hybrid FIR filter in ultrasound image processing. In: IEEE. 18TH International Conference on Electronics, Communications and Computers (conielecomp 2008). 2008. P. 160–164. Citado na p. 17.

MUÑOZ, D. M.; FRANCISCANGELIS, C.; MARGULIS, W.; FRUETT, F.; SÖDERQUIST, I. Low latency disturbance detection using distributed optical fiber sensors. In: IEEE. 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC). 2017. P. 372–377. Citado na p. 17.

PASTRANA, M.; SANTOS, K. R.; FARIAS, A. B. de; MUÑOZ, D. M. Data-Driven Control and Behavior-Based Control Applied to a SISO Mobile Robot. In: IEEE. 2022 Latin American Robotics Symposium (LARS), 2022 Brazilian Symposium on Robotics (SBR), and 2022 Workshop on Robotics in Education (WRE). 2022. P. 395–400. Citado na p. 17.

PATTERSON, D. A.; HENNESSY, J. L. **Computer organization and design ARM edition: the hardware software interface**. Morgan kaufmann, 2016. Citado na p. 32.

PETERSON, W. Encoding and error-correction procedures for the Bose-Chaudhuri codes. **IRE Transactions on information theory**, IEEE, v. 6, n. 4, p. 459–470, 1960. Citado na p. 30.

PICORV32 - A Size-Optimized RISC-V CPU. 2025. Disponível em: <https://github.com/YosysHQ/picorv32>. Citado na p. 35.

RAZAVI, B. **Fundamentals of microelectronics**. John Wiley & Sons, 2021. Citado na p. 38.

SANTOS, B. Design and Implementation of a Simple Moving Average Filter for a UWB/UHF Hybrid RFID Tag. **Microelectronics Students Forum**, SBMicro, 2023. Citado na p. 16.

VAHID, F. **Digital design with RTL design, VHDL, and Verilog**. John Wiley & Sons, 2010. Citado na p. 41.

WESTE, N. H.; HARRIS, D. **CMOS VLSI design: a circuits and systems perspective**. Pearson Education India, 2015. Citado na p. 40.

XU, C.; YIN, S.; QIN, Y.; ZOU, H. A novel hardware efficient FIR filter for wireless sensor networks. In: IEEE. 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN). 2013. P. 197–201. Citado na p. 66.

ZHANG, X. **VLSI architectures for modern error-correcting codes**. Crc Press Boca Raton, FL, USA: 2016. v. 1. Citado nas pp. 27, 54, 56, 57.

APÊNDICE A – Implementações em FPGA

A.1 Implementação do Filtro Média móvel em FPGA

A implementação em FPGA do circuito foi realizada para testar o circuito. Para observar os resultados o circuito foi implementado na FPGA, e este foi conectado a um sensor de temperatura. O sensor de temperatura utilizado foi um termopar, utilizando o chip MAX6675 para realizar a conversão analógico digital do circuito e transmití-lo via SPI para o circuito implementado. Os dados foram coletados do sensor, e o resultado da média móvel no circuito foi verificada.

O consumo de recursos da média móvel na FPGA são apresentados na Tabela 10. É possível ver que a implementação consome poucos recursos, ocupando um espaço pequeno na FPGA.

Tabela 10 – Consumo de recursos do protótipo em FPGA do filtro média móvel

Recurso	Utilização	Disponível	Utilização %
LUT	358	63400	0.17
LUTRAM	16	19000	0.08
FF	134	126800	0.11
IO	41	210	19.52

Na tabela 11 são mostradas as informações a respeito da potência do circuito implementado em FPGA. É importante notar que a maior parte do consumo energético é devido a potência estática. Isso significa que o circuito consome para estar ligado, mas não consome muito para operar. Porque o circuito é pequeno, a maior parte do consumo, é o consumo da própria FPGA para ficar ligada.

A.2 Integração do filtro com RISC-V em FPGA

Após implementação do circuito em FPGA, utilizando SPI para comunicar com um sensor de temperatura, simulando um ambiente de sensoriamento, foi integrado esse bloco a um processador RISC-V. Foi utilizado como base o projeto PicoRV32, do qual se utilizou

Tabela 11 – Consumo de energia do protótipo em FPGA do filtro média móvel

	Frequência	Tecnologia	Tensão da Fonte	Potência Total (μW)
Média móvel simples - SPI	1 MHz	NEXYS 4	3.3	98000

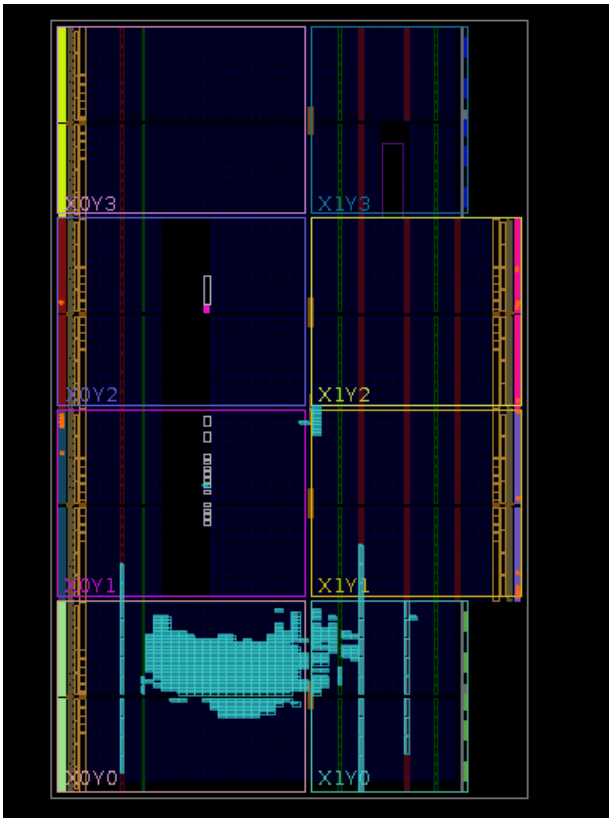


Figura 33 – Layout FPGA, picosoc + SMA

o núcleo de processamento e o código disponível no repositório para integrar com FPGA, que utiliza uma arquitetura com comunicação UART, utiliza uma memória FLASH para armazenar o programa e tem uma integração com blocos terceiros que pode ser realizada por meio de registradores.

Deste modo, foi realizada a integração e foi simulado o circuito utilizando testbenches. A partir do resultado da simulação, foi possível atestar que o resultado é consistente com o modelo em python.

Para realizar a simulação e o teste do circuito após carregar o bitstream na FPGA, é necessário obter o firmware que irá ser executador pelo RISC-V. Para tanto, foi escrito um programa em C que permite a comunicação do RISC-V com o computador em que a FPGA está conectada, via UART. A partir desta conexão via UART, foram enviados números para o RISC-V, que por sua vez envia estes valores para o bloco dedicado de Média Móvel Simples, que realiza a operação e sinaliza o fim da operação por meio de uma flag. Por fim o programa imprime por UART o resultado da operação.

O código em C é compilado via GCC. Para isto foi utilizado uma versão do GCC que compila o código para a arquitetura RISC-V, respeitando as extensões escolhidas para este conjunto de instruções. Para este trabalho foi utilizado especificamente a arquitetura RV32IMC, que suporta as operações básicas, multiplicação e divisão de inteiros e instruções comprimidas.

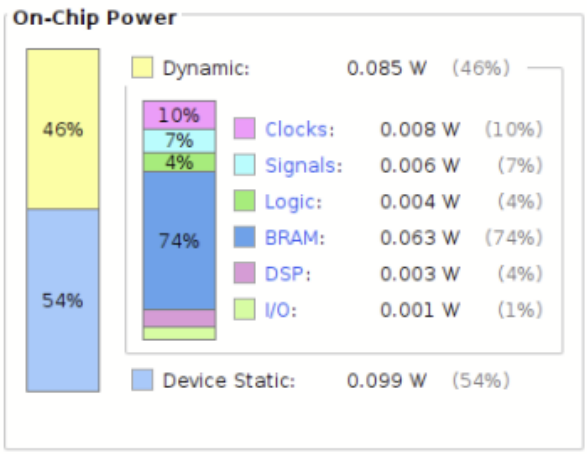


Figura 34 – Consumo de potência do RISC-V + Média Móvel

Resource	Utilization	Available	Utilization %
LUT	2263	63400	3.57
LUTRAM	64	19000	0.34
FF	1375	126800	1.08
BRAM	32	135	23.70
DSP	4	240	1.67
IO	16	210	7.62
BUFG	2	32	6.25

Figura 35 – Consumo de recursos do RISC-V + Média Móvel

Figura 36 – Resultados de timing para o RISC-V + Média Móvel

A Figura 33 mostra o layout do circuito dentro da FPGA. É possível notar pela imagem que o circuito consome uma parte pequena dos recursos da FPGA, sendo possível por exemplo popular a FPGA com diversos núcleos processamento do PicoRV32, caso se quisesse implementar uma arquitetura com vários núcleos de processamento.

A Figura 34 demonstra os resultados de consumo de potência da arquitetura. Neste caso, a distribuição entre potência dinâmica e potência ficou bastante equilibrada, tendo partes quase que iguais em cada um dos tipos de potência. O consumo de potência total se mostrou adequado para um processador, com consumo baixo. Isso é esperado pois o projeto picoRV32 tem o objetivo de ser uma arquitetura de baixo consumo de energia e área, sendo uma arquitetura mais compacta.

Já na Figura 35 é possível ver o consumo de células da FPGA. É possível ver que o consumo de células é baixo em relação aos recursos disponíveis na placa de desenvolvimento.

Finalmente, na Figura 36 é possível ver os resultados de timing para a arquitetura. Nenhum problema de timing foi encontrado, tendo uma folga positiva de timing tanto para setup quanto para hold.