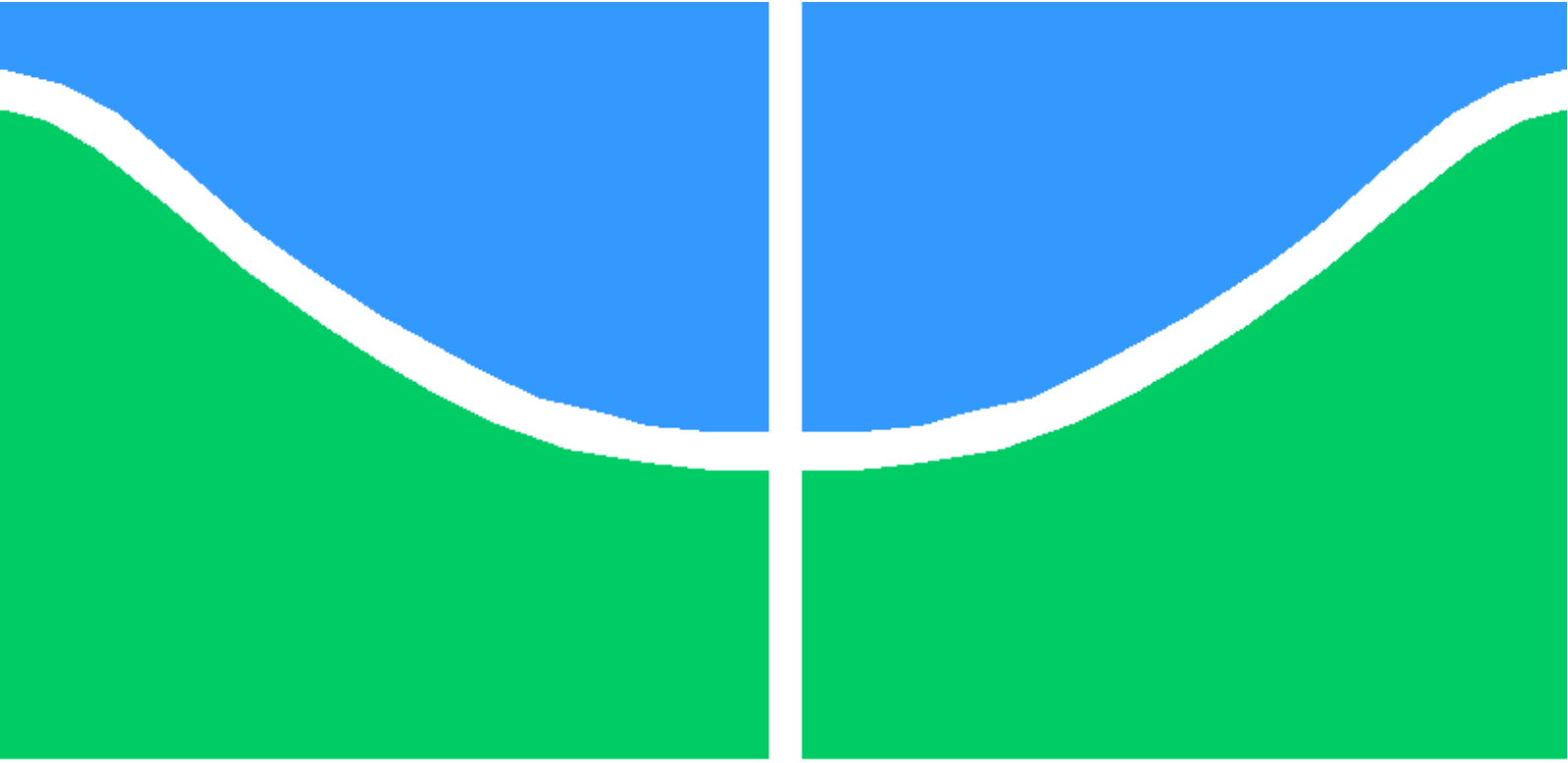

TRABALHO DE GRADUAÇÃO

Modelagem de Sistemas Heterogêneos Utilizando Frameworks Baseados em MoC

Roberta Rosa do Nascimento Hora Guimarães

Brasília, Junho de 2014

UNIVERSIDADE DE BRASÍLIA

FACULDADE GAMA

UNIVERSIDADE DE BRASILIA
Faculdade Gama

TRABALHO DE GRADUAÇÃO

Modelagem de Sistemas Heterogêneos Utilizando Frameworks Baseados em MoC

Roberta Rosa do Nascimento Hora Guimarães

Relatório submetido ao Departamento de Engenharia Eletrônica como requisito parcial para obtenção do grau de Engenheira Eletrônica

Banca Examinadora

Prof. Dr. Gilmar Silva Beserra, FGA-UnB _____
Orientador

Prof. Dr. Wellington Avelino do Amaral, FGA-UnB _____
Examinador interno

Prof. Dr. Sandro Augusto Pavlik Haddad, FGA-UnB _____
Examinador interno

Dedicatória

Dedico este trabalho a minha mãe Valdirene Estevam que sempre me incentivou, me ajudou e que durante toda minha existência se esforçou imensamente para que nunca me faltasse nada essencial e para me ver alcançar uma boa formação acadêmica.

Roberta Rosa do Nascimento Hora Guimarães

Agradecimentos

Venho expressar minha gratidão ao auxílio imensurável recebido pelo meu orientador, o professor Dr. Gilmar Silva Beserra, que desde o início foi capaz de me dar uma visão mais ampla sobre o objetivo deste trabalho e me auxiliou diversas vezes me orientando sobre o que eu deveria fazer e o que eu estava fazendo errado, quando eu me encontrava completamente perdida.

Roberta Rosa do Nascimento Hora Guimarães

RESUMO

O presente texto apresenta uma proposta de trabalho de conclusão de curso que consiste na modelagem em alto nível e na simulação de sistemas heterogêneos utilizando um ambiente baseado na teoria de modelos de computação (MoC). O objetivo principal do trabalho é modelar uma *tag* de RFID como estudo de caso usando a linguagem SystemC e sua extensão SystemC-AMS. Com isso, será possível verificar a funcionalidade da arquitetura proposta através de simulações em alto nível de abstração, bem como disponibilizar protótipos virtuais do sistema, incluindo a funcionalidade das partes analógicas/RF e digital da *tag*. Neste documento são apresentados os conceitos teóricos necessários para o desenvolvimento do trabalho, a modelagem efetuada e os resultados da simulação da *tag* de RFID utilizado como estudo de caso.

ABSTRACT

In this text, we present a proposal for an undergraduate project, which consists on the high level modeling and simulation of heterogeneous systems using one framework based on models of computation (MoC) theory. The main goal of this work is to model an *tag* of RFID system as a case study using SystemC language, its extension SystemC-AMS. With these models, it will be possible to verify the functionality of the proposed architecture using simulations in a high abstraction level. In addition, a virtual prototype of the system, including the analog/RF and digital blocks of *tag*, will be available. In this document, we summarize the basic concepts needed to describe the models and present the modeling and results of the *tag* of the RFID system used as a case study.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SÍMBOLOS

Símbolos Latinos

A	Amplitude	[V]
C_L	Capacitor de Carga	[F]
f	Frequência	[Hz]
P	Potência	[W]
t	tempo	[s]
V_{DD}	Tensão de Alimentação	[V]

Siglas

ADC	Conversor Analógico Digital
AFE	Interface Análogica de Entrada
AMS	Sinais Analógicos e Mistos
ASK	Modulação por Chaveamento de Amplitude
ASIC	Circuito Integrado de Aplicação Específica
BASK	Modulação por Chaveamento de Amplitude Binária
BPSK	Modulação por Chaveamento de Fase Binária
C1G2	Classe 1 Geração 2
CI	Circuito Integrado
CPF	Cadastro de Pessoa Física
CRC	Verificador Cíclico de Redundância
CU	Unidade de Controle
DAC	Conversor Digital Analógico
DAE	Equação Algébrica Diferencial
DC	Corrente Contínua
DDR	Dupla Taxa de Dados
DE	Eventos Discretos
DEC	Decodificador
DEM	Demodulador
DI	Interface de Domínio
DSP	Processador de Sinais Digitais
DSB	Banda Lateral Dupla

Siglas

ELN	Rede Elétrica Linear
EPC	Código Eletrônico do Produto
ESD	Descarga Eletrostática
FGA	Faculdade Gama
FPGA	Matriz de Portas Programáveis em Campo
ForSyDe	Projeto de Sistemas Formais
HDL	Linguagem de Descrição de *Hardware*
HF	Frequência Alta
HW	*Hardware*
IE	Interface para EEPROM
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
IF	Frequência Intermediária
IP	Propriedade Intelectual
LF	Frequência Baixa
LNA	Amplificador de Baixo Ruído
LSF	Fluxo de Sinais Lineares
LSFR	Registrador Linear de Deslocamento
LUT	*Look up Table*
MoC	Modelo de Computação
MOD	Modulador
OCU	Unidade de Controle da Saída
OSC	Oscilador
OSCI	*Open SystemC Initiative*
PA	Amplificador de Potência
PC	Computador Pessoal
PLD	Dispositivo Lógico Programável
PN	Rede de Processos
POR	Liga e Reseta
PR	Reversão de Fase
PWM	Modulação por Comprimento de Pulso
RF	Radiofrequência
RFID	Identificação por Radiofrequência
RG	Registro Geral
RNG	Gerador de Números Aleatórios
ROM	Memória Somente Leitura
RTL	Nível de Transferência de Registradores
SDF	Fluxo de Dados Síncrono
SDRAM	Memória de Acesso Dinâmico Aleatório Síncrona
SoC	Sistema Embarcado

Siglas

SR	Síncrono/Reativo
SSB	Banda Lateral Única
SW	*Software*
TDF	Fluxo de Dados Discretizados
UHF	Frequência Ultra Alta
ULA	Unidade Aritmética e Lógica
UnB	Universidade de Brasília
VHDL	Linguagem de Descrição de *Hardware* de Circuito Integrado de Alta Velocidade

Capítulo 1

Introdução

1.1 Contextualização e Justificativa

Neste capítulo, serão apresentadas a contextualização e justificatica de escolha do tema desta proposta de trabalho de conclusão de curso. O problema a ser resolvido, bem como a abordagem a ser utilizada e o estudo de caso para prova de conceito também são descritos, sendo que em seguida, são definidos os objetivos e, por fim, a estrutura do documento é detalhada.

O projeto de sistemas embarcados tem se tornado uma tarefa de crescente complexidade, considerando-se as diversas interações entre o *hardware*/*software* com o ambiente físico necessárias na maior parte das aplicações. Isso pode ser observado em sistemas cuja funcionalidade HW/SW digital está bastante integrada com blocos análogicos e de sinais mistos, também chamados de AMS *Analogic/Mixed-Signal*. Por exemplo, sistemas de comunicação possuem componentes como antena e transceptores para realizarem uma interface de rádio frequência, sensores, transdutores, osciladores, microprocessadores, etc., englobando módulos que usam sinais analógicos, mistos e digitais [10].

Um sistema com partes que operam com diferentes tipos de sinais é denominado heterogêneo. A principal dificuldade desse tipo de sistema é entender as interações entre os seus componentes analógicos, de sinais mistos e HW/SW no nível de arquitetura. Existe uma grande variedade de sistemas embarcados heterogêneos, tais como radios cognitivos, rede de sensores, sistemas de aquisição de imagem, sistemas de RFID - *Radio Frequency Identification*, dentre outros [10].

Usando o conceito de abstração podemos lidar com os desafios expostos acima, Para isto é necessário desprezar detalhes desnecessários em estágios iniciais do fluxo de projeto o que permite ao projetista se concentrar nos aspectos comportamentais do sistema. Realizar a modelagem e simulação de sistemas heterogêneos no nível funcional oferece inúmeras vantagens, dentre elas o uso de formalismos para incorporar técnicas formais de análise e automação do fluxo de projeto, e o uso de módulos IP para aumento da produtividade. Além disso, modelos em alto nível de abstração permitem validar a funcionalidade do sistema e identificar gargalos na interação entre componentes analógicos, de sinais mistos e de HW/SW em estágios iniciais do fluxo de projeto [10], tais como se o sistema é imune a ruídos, problemas na comunicação HW/SW que devem ser otimizados e se há uma boa compatibilidade entre os blocos analógicos e digitais do sistema.

Alguns ambientes e linguagens de descrição de hardware permitem a modelagem em alto nível de sistemas. Um dos trabalhos pioneiros nessa área é o Ptolemy II, que é implementado em Java e permite a combinação de vários modelos de computação (MoC). O uso de SystemC, uma biblioteca do C++ que permite a descrição de sistemas em vários níveis de abstração, também tem se destacado nessa área, principalmente após a introdução de extensões SystemC-AMS para a descrição de sistemas analógicos e contínuos no tempo. Outra ferramenta bastante versátil é o ForSyDe (*Formal System Design*), que possui uma biblioteca em SystemC que permite a descrição de módulos em diferentes MoC e sua integração em um modelo executável.

1.2 Definição do problema

Considerando que encontra-se em andamento na Faculdade do Gama o projeto e implementação em ASIC de uma *tag* de propósito geral para um sistema de identificação por radio frequência (RFID) como projeto de iniciação científica; que sistemas de RFID podem ser classificados como heterogêneos e, sendo assim, as interações entre os componentes AMS e HW/SW devem ser amplamente entendidas para maximizar o sucesso na sua concepção; que o fluxo de projeto de ASIC deve levar em consideração a modelagem em alto nível para validação da funcionalidade do sistema completo e que neste trabalho, a proposta é realizar a modelagem e simulação de sistemas heterogêneos utilizando *frameworks* baseados em modelos de computação (MoC), tem-se que o sistema de RFID em questão é uma escolha adequada para aplicar o tema proposto e ser utilizado como estudo de caso, abordando os conceitos aqui apresentados.

1.3 Objetivos do projeto

Tendo em vista as considerações acima acerca da importância de modelar e simular sistemas heterogêneos em um alto nível de abstração, este trabalho de graduação se propõe a utilizar a linguagem SystemC e sua extensão SystemC-AMS para modelar e simular uma *tag* de um sistema de RFID tanto na parte analógica/RF como na parte digital. Com isso, será possível avaliar a sua funcionalidade e propor melhorias na sua arquitetura em um estágio inicial do fluxo de projeto, tendo como base a aplicação requerida e outras arquiteturas existentes de sistemas de RFID.

1.4 Apresentação do documento

No capítulo 2, é feita uma revisão bibliográfica apresentando alguns conceitos importantes acerca da modelagem de sistemas e descrevendo brevemente algumas ferramentas utilizadas. O capítulo 3 contém definições de sistemas de RFID uma revisão de arquiteturas encontradas na literatura, incluindo um protocolo de comunicação I2C que será utilizado na comunicação digital. Em seguida, no capítulo 4, são descritas a metodologia utilizada e a modelagem de uma *tag* de RFID como estudo de caso. Os resultados das simulações são discutidos no capítulo 6, e o capítulo 7 resume o que já foi realizado, apresentando também sugestões de implementações em trabalhos futuros.

Capítulo 2

Modelagem de Sistemas

2.1 Níveis de abstração

Este capítulo apresenta alguns conceitos importantes para o entedimento do tema proposto, abordando conceitos de modelagem de sistemas, como níveis de abstração e modelos de computação, além de descrever brevemente algumas ferramentas utilizadas para modelagem de sistemas heterogêneos, bem como suas aplicações e limitações.

Ao se projetar, modelar e analisar um sistema microeletrônico complexo, é importante que se conheça um conceito denominado nível de abstração. Um nível de abstração é basicamento um conjunto de descrições de projeto com o mesmo grau de detalhamento [1].

Ao se avaliar os níveis de abstração de um sistema, é importante saber que, quanto mais alto o nível de abstração, menos detalhes se tem a respeito do sistema, obtendo-se assim uma representação menos complexa do mesmo. De maneira similar, quanto menor o nível de abstração, mais detalhes vão sendo agregados ao sistema.

A análise de um sistema em alto nível de abstração é ideal para se verificar como ocorrem as interações entre os módulos que compõem este sistema, bem como para entender o comportamento e a estrutura geral do mesmo. Um sistema descrito em baixo nível de abstração é mais complexo de se entender como um todo, porém neste caso os blocos individuais do sistema são modelados com mais precisão.

A Fig. 2.1 ilustra os diferentes níveis de abstração existentes através de círculos concêntricos. Quanto maior é o raio do círculo, mais alto é o nível de abstração. Esta representação é conhecida como diagrama Y ou diagrama Gajski-Kuhn.

Podemos notar pela Fig. 2.1 que existem quatro níveis de abstração que são considerados para análise de circuitos eletrônicos, sendo os mesmos classificados em nível sistêmico, arquitetural, lógico e elétrico.

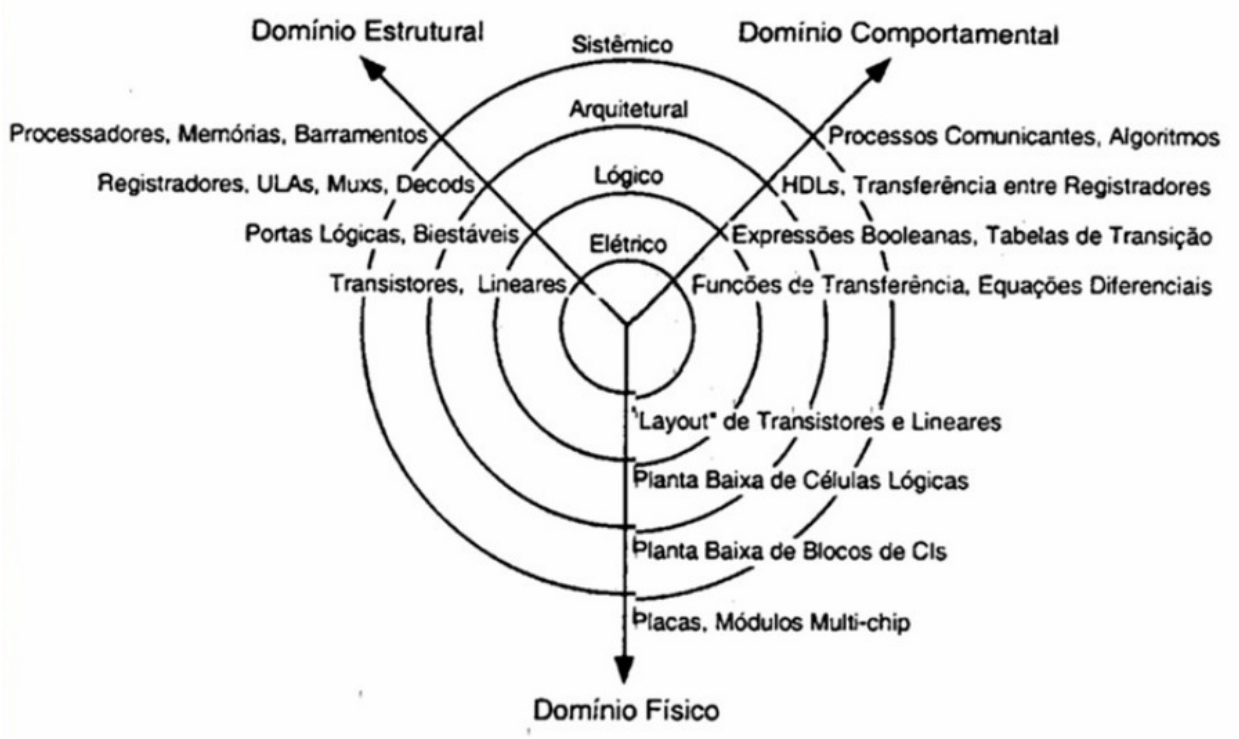

Figura 2.1: Diagrama Y que ilustra o conceito de níveis de abstração [1].

No nível sistêmico, o sistema é descrito através de algoritmos ou módulos. Já no nível arquitetural, o sistema é descrito através de ULAs, registradores, moduladores, conversores D/A e A/D, etc. O nível lógico é descrito através de portas lógicas e flip-flops. Por fim, o nível elétrico possui seus modelos baseados na física de semicondutores e na teoria básica de circuitos elétricos e eletrônicos [1].

Ainda na Fig. 2.1, é possível observar a existência de três domínios de descrição, representados através dos segmentos de reta radiais que compõem a letra Y, sendo os mesmos o domínio estrutural, o domínio comportamental e o domínio físico. Cada domínio corresponde a um conjunto de descrições que compartilham o mesmo tipo de informação.

No domínio físico, tem-se a descrição sobre a geometria dos componentes e módulos, bem como a disposição dos mesmos ao serem fabricados. O domínio estrutural, por sua vez, descreve como são interconectados os módulos que compõem o sistema. O domínio comportamental contém as informações sobre o comportamento do sistema [1].

Cada intersecção de um círculo com um segmento radial possui uma descrição distinta do sistema a ser analisado [1]. O presente trabalho irá abordar a descrição nos domínios comportamental e estrutural em alto nível de abstração utilizando uma *tag* de RFID como estudo de caso, sendo que os mesmos serão modelados e simulados apenas considerando o nível sistêmico com o objetivo de verificar sua funcionalidade e analisar sua arquitetura em um estágio inicial do projeto. Tal abordagem permite que o sistema seja entendido como um todo, e também permite verificar algumas limitações das arquiteturas descritas. Para entender como os sistemas serão descritos, é necessário utilizar conceitos de modelos de computação, apresentados na próxima seção.

2.2 Modelos de Computação - MoC

Um MoC - (*Model of Computation*) é uma representação abstrata que independe de uma linguagem particular de programação e que pode ser utilizado para modelar um sistema através de métodos formais de análise matemática [2].

Basicamente, um MoC descreve uma composição de processos simultâneos,definindo as propriedades dos mesmos, como eles se comunicam entre si e quais são os mecanismos de concorrência presentes nesta comunicação [2].

A Fig.2.2 exemplifica o funcionamento de um modelo de computação, onde os processos simultâneos P1, P2 e P3, interagem entre si através de sinais, onde cada um destes processos possuem propriedades que podem ou não ser diferentes entre si, isto dependerá do modelo de computação em que cada um destes processos é descrito.

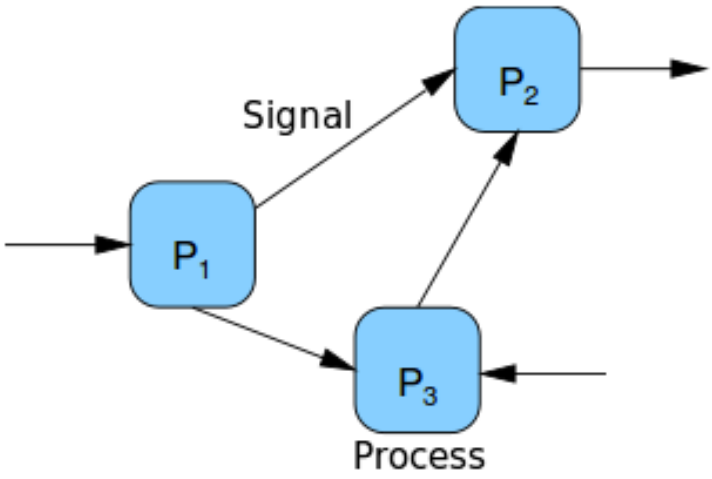

Figura 2.2: Exemplo de processos simultâneos que podem ser modelados por MoC [2].

Por serem bastante abstratos, modelos de computação podem ser utilizados para modelar os mais variados tipos de sistema, sendo portanto uma abordagem bastante poderosa. Diferentes tipos de sistemas requerem diferentes tipos de MoC. A escolha do tipo de MoC utilizado para modelar um sistema deve levar em consideração o quanto este modelo consegue abstrair o problema e qual é a eficiência da implementação deste MoC [11].

Um MoC muito abstrato é ineficiente e impraticável por não possuir detalhes suficientes para analisar formalmente um sistema. No entanto, caso um MoC apresente muitos detalhes e especificações, torna-se difícil de ser implementado e aumenta consideravelmente o tempo de projeto. Logo, para se obter um MoC eficiente, é necessário avaliar com cuidado o nível de detalhamento que é necessário para descrever o sistema [11].

Algumas ferramentas para modelagem de sistemas heterôgeneos em nível sistêmico e arquitetural utilizam modelos de computação. Desta forma, é importante que o conceito apresentado acima seja bem assimilado para entender como utilizar estas ferramentas.

2.3 Ferramentas para Modelagem de Sistemas Heterogêneos

Existem atualmente diversas ferramentas que permitem a modelagem e simulação de sistemas heterogêneos. Dentre elas, pode-se citar o Simulink, Ptolemy II, linguagens de descrição de hardware (HDL's), tais como VHDL/VHDL-AMS e Verilog/Verilog-AMS/Verilog-A, SystemC/SystemC-AMS, e ForSyDe - (*Formal System Design*).

Cada uma destas ferramentas apresenta vantagens e desvantagens em relação às outras, sendo mais adequada para um determinado tipo de aplicação. Essas vantagens e desvantagens serão brevemente descritas nas subseções a seguir.

2.3.1 Simulink

O Simulink é uma ferramenta para modelagem, simulação e análise desenvolvida pela companhia *TheMathWorks*. Essa ferramenta é acessada ao digitar no *MatLab* o comando »simulink. Por ser uma ferramenta disponível no *MatLab*, a mesma não é gratuita, o que é uma desvantagem em comparação às outras que serão analisadas.

O Simulink é muito utilizado para realizar modelagem funcional e simulação, podendo inclusive verificar o comportamento do sistema em tempo contínuo. Porém, o alvo dessa ferramenta não é o projeto de sistemas embarcados AMS em nível de arquitetura [10]. Sendo assim, essa ferramenta é mais apropriada para a descrição de outros tipos de sistemas, como sistemas dinâmicos na abordagem de controle por exemplo.

2.3.2 Ptolemy II

O Ptolemy II vem sendo desenvolvido desde 1996 e consiste em uma coleção de classes e pacotes em Java capazes de oferecer recursos cada vez mais específicos. Essa ferramenta permite uma sintaxe abstrata utilizando uma estrutura hierárquica de entidades com portas e interconexões que pode inclusive ser modelada através de uma interface gráfica [12].

Várias ferramentas especializadas foram criadas com o Ptolemy II, como HyVisual, para modelagem de sistemas híbridos; VirtualSense, para modelagem e simulação de redes sem fio; Viptos, para projeto de redes de sensores, dentre outras [12].

O Ptolemy II é um *software open-source* que permite o projeto orientado por atores. Os atores são componentes de *software* que são executados simultaneamente, sendo capazes de descrever sistemas com processos simultâneos. O Ptolemy II permite também que estes atores sejam interconectados de forma hierárquica [12].

Nessa ferramenta, a semântica é determinada por um componente de *software* denominado diretor, que implementa um modelo de computação. Os diretores do projeto Ptolemy suportam os modelos de computação de rede de processos - PN; eventos discretos - DE; fluxo de dados - SDF; síncrono/reativo - SR; visualização 3-D; e modelos de tempo contínuo. Cada nível de hierarquia pode possuir seu próprio diretor [12]. Basicamente, o Ptolemy II busca obter uma compreensão das combinações heterogêneas de modelos de computação realizadas pelos diretores.

Essa ferramenta, assim como o Simulink, é muito utilizada para realizar modelagem funcional e simulação em nível sistêmico, possuindo ainda a vantagem de ser uma ferramenta gratuita. Porém, o Ptolemy II, assim como o Simulink, não tem como alvo o projeto de sistemas embarcados AMS em nível de arquitetura, sendo mais útil para outros tipos de aplicação [10].

2.3.3 Linguagens de Descrição de *Hardware*

As linguagens de descrição de *hardware* (HDL's) são extremamente úteis para descrever o comportamento de um circuito eletrônico ou sistema de forma que o mesmo possa ser sintetizado e implementado fisicamente [13].

As principais linguagens de descrição de *hardware* existentes são o VHDL e Verilog, sendo que a principal motivação para o uso dessas linguagens é o fato de ambas serem extremamente padronizadas, ou seja, possuem a mesma sintaxe, independente da tecnologia e do fornecedor. Essa característica faz com que os códigos elaborados em VHDL ou Verilog sejam bastante portáveis e reutilizáveis [13].

Por serem linguagens que se dispõem a descrever o comportamento de *hardware*, tanto o VHDL como o Verilog possuem suas declarações funcionando de forma paralela. Caso se deseje um comportamento sequencial de algum bloco, o mesmo deve ser descrito dentro de um processo [13].

As principais aplicações para HDL's encontram-se no campo de dispositivos lógicos programáveis, como PLD's e FPGA's, e no campo de ASIC's. Atualmente, diversos chips comerciais utilizam essa abordagem [13].

As linguagens VHDL e Verilog conseguem englobar somente aplicações no domínio digital, sendo inadequadas para descrever sistemas analógicos. Devido a essa limitação, foram desenvolvidas extensões de forma a permitir a modelagem de sistemas mistos. Essas extensões foram denominadas VHDL-AMS, Verilog-AMS e Verilog-A.

Apesar das extensões para modelagem de sistemas heterogêneos, as HDL's ainda apresentam limitações para modelar de forma eficiente a comunicação HW/SW de sistemas heterogêneos em nível sistêmico, sendo mais adequadas para modelagem em nível de implementação [10].

Devido a essa limitação, as linguagens VHDL/VHDL-AMS e Verilog/Verilog-AMS/Verilog-A não serão utilizadas neste trabalho. A linguagem escolhida foi SystemC e sua extensão SystemC-AMS, descritas a seguir.

2.3.4 SystemC e SystemC-AMS

O SystemC foi construído utilizando um padrão C++ estendido por um conjunto de bibliotecas de classes criadas com objetivo de projeto e verificação. Essa linguagem é usada para modelagem em nível de sistema, exploração de arquitetura, modelagem abstrada com sinais mistos, modelagem de performance, desenvolvimento de *software*, verificação funcional e síntese de alto nível. Basicamente, o SystemC aborda o projeto e verificação *hardware/software* de um sistema [14].

Essa linguagem foi definida por *Open SystemC Iniciative* (OSCI) e ratificada pelo IEEE *Standart 1666*^{TM}-2011, sendo amplamente utilizada por companhias líderes em propriedade intelectual (IP), automação de projetos eletrônicos, semicondutores, sistemas eletrônicos e *software* embarcado. Essas companhias usam o SystemC para exploração de arquiteturas, desenvolvimento de blocos de *hardware* de alta performance em vários níveis de abstração e desenvolvimento de plataformas virtuais para projetos que envolvem integração *hardware/software* [14].

Um sistema em chip SoC - (*System on Chip*) é contistuído de silício e um *software* embarcado. O projeto desse tipo de sistemas envolve algoritmos complexos, desenvolvimento de arquitetura e análise de desempenho, envolvendo processos de *trade-off* que definem pontos críticos do projeto, tais como funcionalidade, consumo de potência e desempenho [14].

Desta forma, as ferramentas utilizadas para esses tipo de sistema devem garantir uma melhoria significativa na produtividade buscando aprimorar em diversas ordens de magnitude tanto o projeto da arquitetura como a implementação do SoC. Também é necessário que as ferramentas permitam que o desenvolvimento do *software* seja realizado antes que o desenvolvimento do *hardware* esteja finalizado, de forma a garantir que os custos do projeto sejam menores e que não se perca janelas do mercado [14].

Por ser uma linguagem com o código fonte aberto para projeto e verificação de arquitetura e outros atributos em nível de sistema, o SystemC permite que seja realizado o desenvolvimento do projeto de *hardware* e *software* de forma concorrente em nível de sistema. Além disso, por possuir um alto nível de abstração, permite que a verificação de arquitetura e do desempenho seja feita bem mais rapidamente que no nível de transferência de registradores (RTL), o que torna esta linguagem extremamente adequada para o desenvolvimento de SoC [14].

O SystemC-AMS é uma extensão da linguagem SystemC que permite o projeto e modelagem em vários níveis de abstração de sistemas embarcados que utilizam sinais mistos, analógicos e digitais. Sendo assim, essa linguagem é utilizada para realizar modelagem funcional, exploração de arquitetura e prototipagem vitual em sistemas embarcados de sinais mistos [14].

O SystemC-AMS tem sua principal aplicação voltada para modelagem em alto nível de abstração de sistemas heterogêneos. Desta forma, trata-se de uma ferramenta que se adequa ao desenvolvimento do tema proposto neste trabalho.

O SystemC-AMS fornece suporte para os modelos de computação de fluxo de sinais - SDF, fluxo de dados - TDF e redes elétricas - ELN. As redes elétricas e fluxos de sinais são modelados através de um módulo DAE - *Differential Algebraic Equation*, capaz de resolver equações diferenciais algébricas lineares. O DAE é restrito a modelos lineares de redes elétricas e fluxo de sinais, proporcionando assim um alto desempenho de simulação [10].

Já a simulação para fluxo de dados utiliza um escalonador estático computado antes do início da simulação. Desta forma, o escalonador é ativado em um intervalo discreto de tempo e sincronizado pela semântica de tempo do SystemC. Esse componente é chamado de fluxo de dados temporarizado, ou TDF - *Timed Data Flow* [10].

Podemos notar pela Fig. 2.3 que o SystemC-AMS é completamente compátivel com o padrão SystemC.

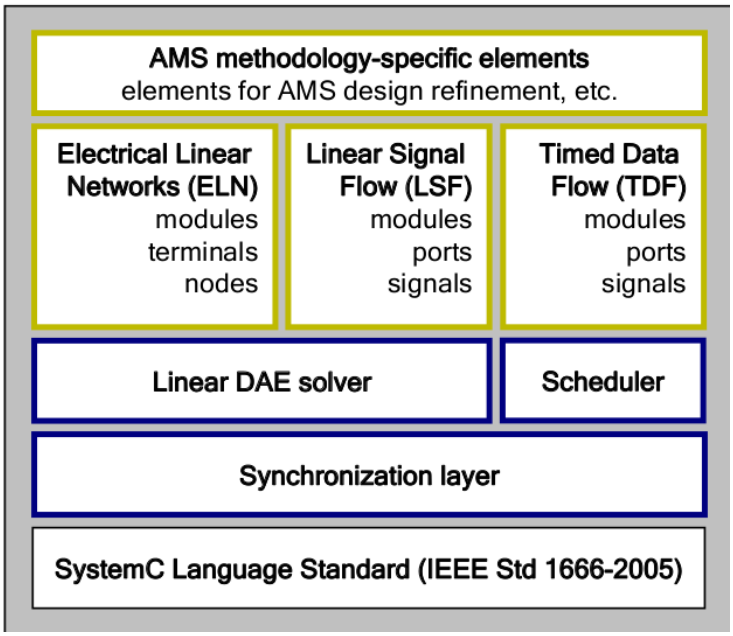

Figura 2.3: Extensão AMS para o padrão da liguagem SystemC [3].

Assim como o SystemC, o SystemC-AMS utiliza em sua estrutura a declaração de módulos, com cada módulo possuindo um construtor associado. Porém, nesta nova construção de linguagem é utilizado o prefixo *sca_*, sendo utilizado *sca_tdf*, *sca_eln* e *sca_lsf*, de acordo com o modelo requerido, para indicar portas, interfaces, sinais e módulos [3].

Ao utilizar o SystemC-AMS, é importante definir inicialmente qual modelo que representa mais eficientemente o módulo que se deseja modelar. Para realizar essa escolha, é importante saber que sistemas análogicos não podem ser resolvidos pela comunicação e sincronização de processos, sendo modelados através de equações diferenciais. Desta forma, os mesmos devem ser modelados utilizando ELN - *Electrical Linear Networks* ou LSF - *Linear Signal Flow* que são os modelos de comunicação para redes elétricas e fluxo de sinais citados acima [3].

O ELN é utilizado quando se requer uma abordagem conservativa, ou seja, a representação da estrutura topológica do sistema modelado. Neste modelo, os nós são caracterizados com duas quantidades - a tensão e a corrente - e o sistema é modelado ao aplicar a lei de Kirchhoff da corrente e da tensão [3].

Já o modelo LSF utiliza uma abordagem não conservativa, onde tem-se uma representação abstrada do comportamento analógico do sistema e obtém-se uma função de transferência que descreve o comportamento da saída em relação à entrada [3].

Por fim, o modelo TDF é utilizado para trabalhar com fluxo de dados em tempo discreto, sendo este modelo possível de ser tratado por comunicação e sincronização de processos [3].

O tempo de simulação do SystemC-AMS varia dependendo do Modelo de Computação utilizado. Sendo que o Modelo de computação que apresenta a simulação mais rápida, na ordem de alguns segundos, é o TDF - *Timed Data Flow*, devido a este ser implementado através de um escalonador.

Em seguida, temos os modelos de computação implementados pelo DAE - *Differential Algebraic Equation* que são o LSF e ELN. O LSF - *Linear Signal Flow* que tem um tempo de simulação na ordem de segundos a poucos minutos, é mais rápido que o ELN - *Electrical Linear Networks* devido ao mesmo possuir uma abordagem não conservativa, descrevendo os sinais através de funções de transferência no lugar de descrevê-los pelas leis de Kirchhoff da tensão e da corrente conforme o modelo ELN efetua. Porém, mesmo os sistemas descritos em ELN que possuem tempo de simulação na ordem de minutos ainda são mais rápidos que os sistemas descritos em VHDL-AMS ou Verilog-A/Verilog-AMS, sendo esta outra vantagem que torna essa linguagem mais apropriada para modelagem em nível sistêmico de um sistema heterogêneo [3].

A Fig. 2.4 apresenta diferentes tempos de simulação para as liguagens SystemC-AMS, VHDL-AMS, Verilog-AMS e Verilog-A [3].

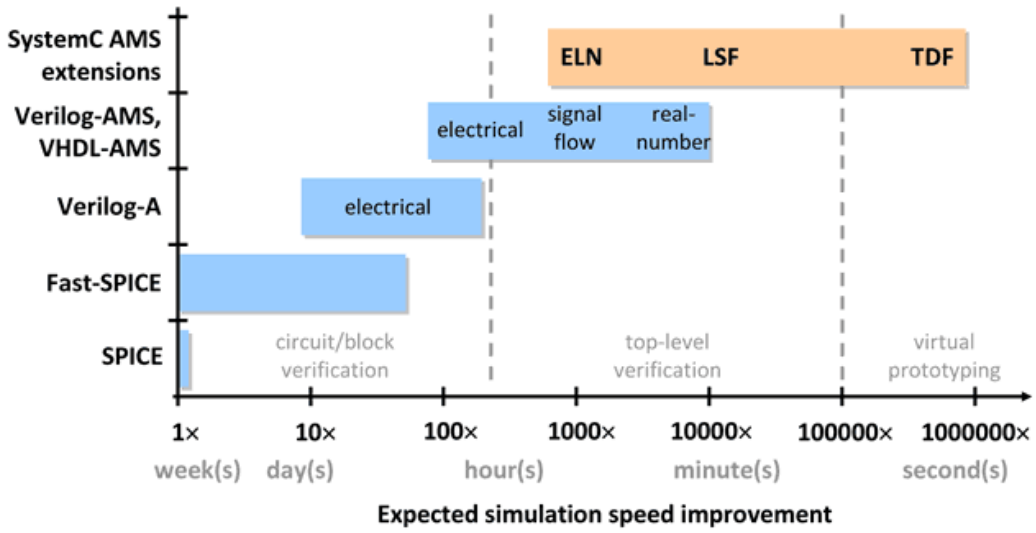

Figura 2.4: Tempo de simulação dos MoC's do SystemC-AMS comparado a *linguagens de descrição de hardware* [3].

Após definir o modelo de computação que melhor representa o módulo que se deseja modelar, basta realizar a implementação do mesmo, instanciando depois os blocos individuais para obter uma modelagem hierárquica de um sistema heterogêneo. Durante este trabalho, essa será a abordagem adotada para modelar a *tag* de RFID.

2.3.5 ForSyDe - *Formal System Design*

Por fim, temos a última ferramenta apresentada neste trabalho. O ForSyDe é uma metodologia com base formal para concepção de sistemas embarcados heterogêneos. Essa metodologia utiliza a teoria de modelos de computação citada anteriormente para capturar as especificações de um sistema heterogêneo, sendo suportada por um conjunto de ferramentas, bibliotecas de modelos e documentações relacionadas. O ForSyDe foi desenvolvido buscando mudar o desenvolvimento de projetos de sistema para um alto nível de abstração, servindo para tornar possível fazer refinamentos e transformações de projetos no mais alto nível de abstração [2].

Os fundamentos formais nos quais o ForSyDe se baseou foram cuidadosamente selecionados de forma a permitir que primeiramente o projetista forneça um modelo de especificação inicial que pode envolver diferentes tipos de modelos de computação que serão modelados por uma rede de processos interconectados por sinais.

Cada processo é gerado por um construtor de processos que permite que a comunicação seja separada da funcionalidade. Desta forma, no ForSyDe, um sistema pode ser modelado utilizando uma rede de processos concorrentes hierárquica, tais como P1, P2, P3, P4 e P5 da Fig. 2.5, que se comunicam entre si através de sinais. Esses processos podem ser de diferentes modelos de computação, indicados na Fig. 2.5 como MoC A e MoC B.

Os processos de modelos de computação diferentes comunicam-se entre si através de uma interface de domínio (DI), responsável por converter o sinal de um modelo de computação para outro [2].

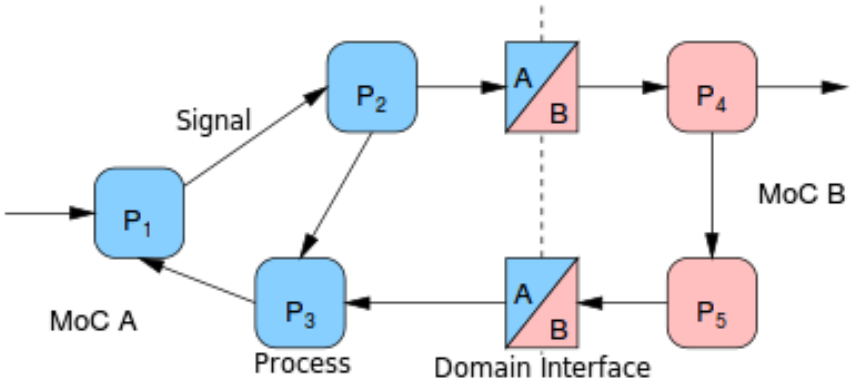

Figura 2.5: Exemplo de um projeto de um sistema baseado em MoC pelo ForSyDe [2].

Desta forma, é possível obter um modelo abstrato de especificações, que por sua vez, são refinados usando transformações de projetos para modelos de implementação detalhados, que por fim, são traduzidos para linguagem que será utilizada na implementação [2].

O ForSyDe está sendo utilizado principalmente para realizar a modelagem de sistemas heterogêneos usando modelos de computação e o refinamento e implementação de modelos de especificação. Sendo assim, o ForSyDe também é apropriado para descrever e simular sistemas heterogêneos baseando-se na teoria de modelos de computação que é o tema proposto neste trabalho, sendo proposto como uma metodologia alternativa para efetuar a modelagem destes sistemas. Neste trabalho o ForSyDe não será utilizado, porém o mesmo é uma opção interessante para um trabalho futuro nesta área.

2.3.6 Simulação Mista

Existe ainda a possibilidade de se efetuar a modelagem de um sistema heterogêneo utilizando para tal simulações mistas. Neste tipo de abordagem alguns blocos do sistemas podem ser modelados em nível de transistores enquanto outros são modelados através do *Simulink* ou de uma linguagem de descrição de *Hardware* ou SystemC/SystemC-AMS.

Resumidamente esta abordagem permite um refinamento do sistema, visto que ao modelar estes blocos em nível de transistores temos uma descrição mais próxima da realidade. Isto permite que os efeitos não lineares de partes mais críticas do sistema tais como filtros, amplificadores, sejam modelados de forma mais exata e com isso tornar mais claro os efeitos que estes efeitos não lineares ocasionam no sistema como um todo.

Esta abordagem é bastante interessante para refinar ainda mais o modelo que está sendo levantado. Apesar disto, a mesma não será abordada neste trabalho, por utilizar uma modelagem em nível de transistores o que foge do tema proposto que é explorar a modelagem de sistemas heterogêneos utilizando *frameworks* baseados em Modelos de Computação, porém esta abordagem também é interessante para um trabalho futuro nesta área.

Capítulo 3

Sistemas de Identificação por Radio Frequência

3.1 Introdução

Este capítulo apresenta conceitos sobre sistemas de RFID e descreve algumas arquiteturas de sistemas de RFID encontradas na literatura. Com base nessas arquiteturas, foi definida a arquitetura a ser modelada neste trabalho de graduação como estudo de caso, cujos detalhes serão apresentados no capítulo 4.

A identificação por rádio frequência é baseada em uma tecnologia que utiliza ondas eletromagnéticas para realizar a comunicação entre um leitor (*reader*) e uma etiqueta (*tag*) de RFID. Sistemas de RFID possuem diversas aplicações devido à sua característica de permitir a transmissão de sinais através de um sinal de RF, sem a necessidade de contato entre o *reader* e a *tag*.

Sistemas típicos de RFID são compostos de um leitor, da etiqueta de RFID e de um banco de dados. O leitor envia um sinal de RF para a *tag*, em geral passiva, e a *tag* responde ao sinal de RF enviado pelo leitor, utilizando para tal a energia coletada pelo sinal recebido pelo leitor [15].

A comunicação entre os componentes típicos em um sistema de RFID se dá conforme é explicado a seguir. O leitor envia sinais de RF para a *tag* através de uma antena, sendo que esses sinais contêm a informação que o leitor está requerindo. A *tag*, por sua vez, coleta o sinal enviado pelo leitor, também através uma antena, utilizando parte do sinal de RF para prover sua alimentação e demodulando a outra parte do sinal para ter acesso a informação que o leitor está requisitando. Depois, a *tag* acessa na memória a informação que o leitor deseja, modula essa informação, transformando-a em um sinal de RF, e a envia através da antena de volta para o leitor. Cabe ressaltar que esta comunicação entre leitor e *tag* pode ser realizada através de diferentes protocolos de comunicação.

Um sistema típico de RFID pode ser observado mais claramente na Fig. 3.1.

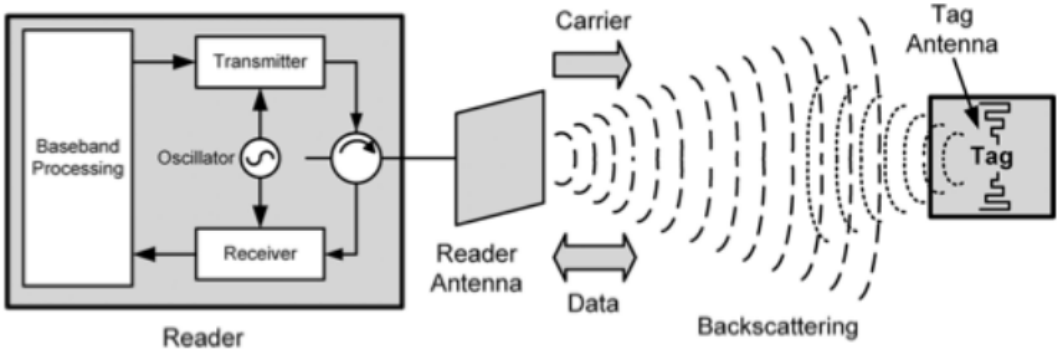

Figura 3.1: Sistema Típico de RFID [4].

3.1.1 Tipos de *tags* de RFID

As *tags* de RFID podem ser classificadas de acordo com o seu tipo de alimentação em:

Ativas: *Tags* de RFID que utilizam baterias internas tanto para prover a alimentação de seus circuitos como para enviar as ondas de rádio frequência para o leitor.

Semi-passiva: Nestas *tags*, a bateria interna é utilizada apenas para prover a alimentação de seus circuitos, sendo a transmissão das ondas de RF dependente da energia coletada do leitor.

Passivas: *Tags* que dependem totalmente do leitor para obter a energia utilizada tanto para alimentação dos seus circuitos como para transmissão de ondas de RF.

As *tags* ativas e semi-passivas possuem um preço mais elevado por possuirem mais *hardware* em seus circuitos que as *tags* passivas. Este fato torna as *tags* passivas mais atraentes para aplicações que envolvam uma grande quantidade de mercadorias [4].

Já quanto à funcionalidade, as *tags* de RFID podem ser classificadas como:

Propósito único: *Tags* que possuem apenas um único conjunto de dados utilizados para uma aplicação específica.

Propósito geral: *Tags* que possuem um conjunto de informações armazenadas na memória. De acordo com a requisição do leitor, será buscado na memória o conjunto de dados que deverá ser enviado ao leitor. Este tipo de *tag* é utilizado para englobar uma faixa maior de aplicações.

Cabe ressaltar que as etiquetas de propósito geral são capazes de interagir com vários leitores, enviando a cada um deles a informação que o mesmo está requerindo [4].

3.1.2 Aplicações dos sistemas de RFID

Existem diversas aplicações para um sistema de RFID. Dentre elas, pode-se citar:

Identificação de mercadorias, com cada mercadoria possuindo uma *tag* de RFID passiva de propósito único com seu código de produto. Esta aplicação é semelhante a do código de barras, porém apresenta a facilidade de permitir que a leitura dos códigos do produto seja feita à distância,

eliminando as incômodas filas na hora de realizar o pagamento de uma ou mais mercadorias. Neste caso, basta que os produtos caiam no campo de um leitor para o mesmo detectar o que foi comprado, debitar este valor na conta de quem está efetuando a compra e ainda controlar o estoque.

Sistema de identificação universal, *tags* de RFID passivas de propósito geral que armazenam todas as informações de uma pessoa física ou jurídica, tais como, número do RG, carteira de trabalho, CPF, passaporte, cartão de crédito, etc., permitindo que apenas um documento apresente todas as informações relevantes de uma pessoa, seja ela física ou jurídica.

Rastreamento de cargas, onde diversos leitores espalhados durante o percurso conseguem localizar as mercadorias presentes em um transporte de carga permitindo que o cliente seja informado sobre em que parte do percurso está o seu produto.

Só pelas aplicações citadas acima, pode-se perceber que esta tecnologia é bastante interessante. No Brasil, por exemplo, *tags* de RFID são amplamente utilizadas na agropecuária, onde uma *tag* colocada na orelha do animal contém todo histórico da vida deste animal.

3.2 Arquiteturas de Sistemas de RFID

Conforme mencionado no item anterior, um sistema de RFID possui como principais componentes o leitor, a *tag* e um banco de dados [15]. Logo, antes de iniciar a modelagem, é necessário conhecer algumas arquiteturas para esse tipo de sistema, tanto para a *tag* como para o leitor. Cabe ressaltar que as arquiteturas de *tags* que serão modeladas neste trabalho são passivas e de próposito geral.

É extremamente importante que uma *tag* de RFID passiva seja de baixo consumo, principalmente porque a *tag* deve ser capaz de usar a energia do sinal de RF recebido para prover sua energia e enviar novamente os sinais de RF para o leitor. Desta forma, se a *tag* for de baixo consumo, é possível maximinizar o tempo de operação, incrementar a comunicação à distância e reduzir os custos [5]. Tal fato direcionou a busca por arquiteturas de *tag* de RFID em trabalhos de envolvam a preocupação com baixo consumo.

3.2.1 *Tag* de RFID Passiva de Propósito Geral

Em [4], foi realizada a prototipação de uma *tag* de RFID em uma FPGA utilizando a linguagem de descrição de *hardware* VHDL. Esse trabalho baseou-se na arquitetura de Harvard (Fig. 3.2), já que a mesma é um bom ponto de partida para otimizar o desempenho e velocidade.

Porém, em [4], a preocupação maior estava em verificar a funcionalidade de uma *tag* de RFID de propósito geral. Por isso, nesse trabalho foi realizada uma simplificação da arquitetura Harvard buscando obter uma arquitetura inovadora que otimizasse a comunicação de uma *tag* de propósito geral de baixo consumo com o leitor, sendo sugerido o apresentado na Fig. 3.3.

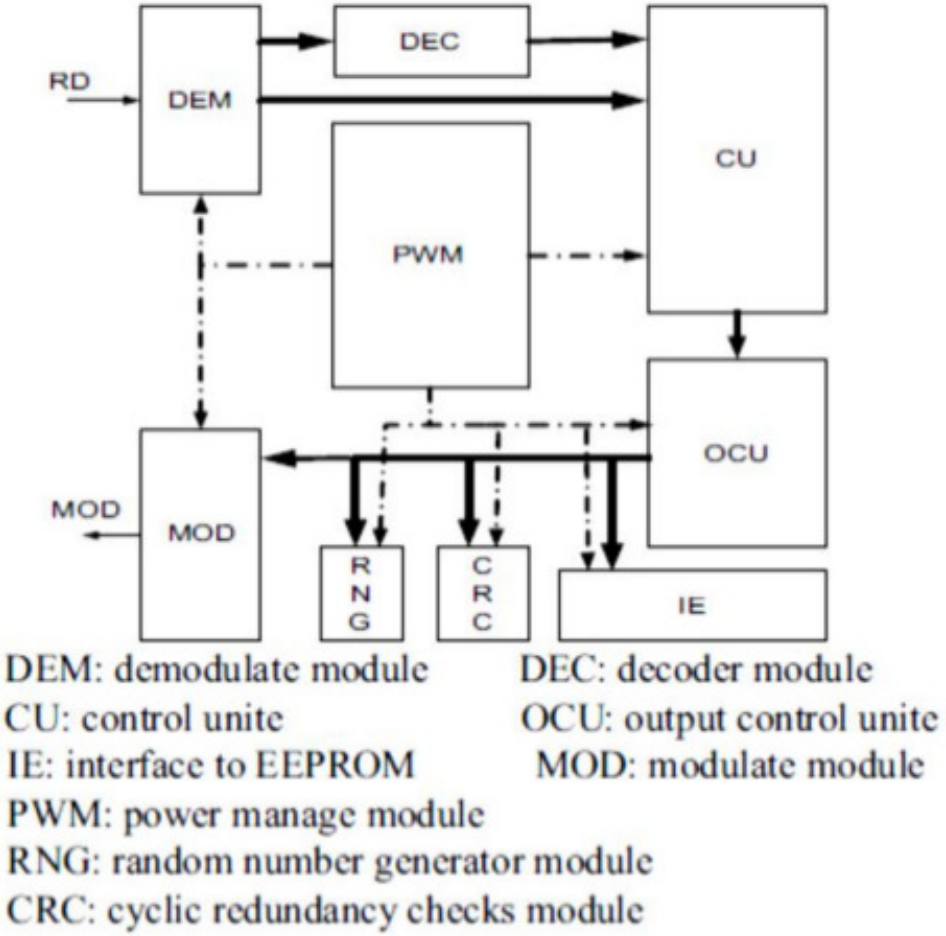

Figura 3.2: Arquitetura Harvard [4].

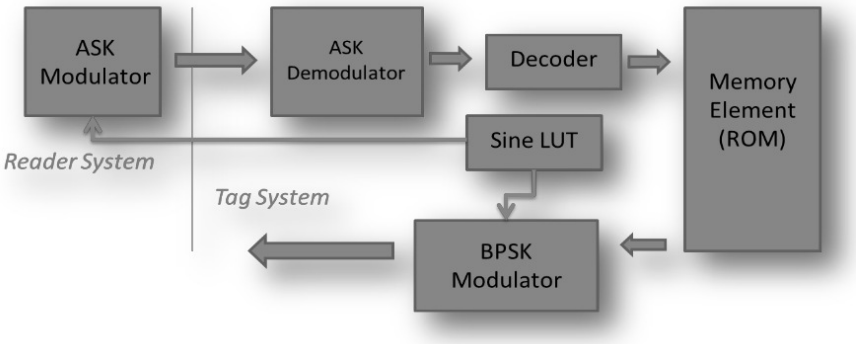

Figura 3.3: Arquitetura implementada em [4].

Esta arquitetura adotada em [4] dá ênfase ao transceptor, que é o bloco composto do receptor e transmissor de RF da *tag*, e no acesso à memória. Basicamente, A *tag* irá receber um sinal modulado do leitor, demodular esse sinal de RF e enviá-lo a um decodificador. O decodificador, por sua vez, irá referenciar o endereço da memória na qual se está requisitando leitura, a memória irá acessar esta informação e enviá-la para o modulador que, por sua vez, irá realizar a modulação, de maneira que a informação seja enviada de volta ao leitor através de uma antena.

Essa arquitetura é um bom ponto de partida para um entendimento inicial do funcionamento de *tags* de RFID. Porém, faltam blocos importantes para compor de fato uma *tag* de RFID de propósito geral, conforme ficará mais evidente a seguir.

Cabe ressaltar que em [4] foi utilizado um demodulador ASK e um modulador BPSK. A redução do consumo nesse trabalho se dá pela reduzida quantidade de blocos utilizados em um sistema de RFID.

3.2.2 Demodulador ASK de Baixo Consumo

No trabalho desenvolvido em [5], o foco foi direcionado à realização de um demodulador ASK de baixo consumo para uma *tag* de RFID que opera nas frequências UHF - *Ultra-High Frequency*.

O demodulador ASK, também utilizado na arquitetura anterior em [4], tem sua utilização justificada por ser a mais simples forma de demodulação[5]. Como é importante que a *tag* possua baixo consumo, quanto mais simples for a operação de seus componentes, melhor será para minimizar o consumo de potência da *tag*.

Já a operação em UHF, comparada à operação em LF - *Low Frequency* e HF - *High Frequency* para sistemas de RFID, é mais vantajosa, pois ao se aumentar a frequência do sinal de RF, diminui-se o comprimento de onda, já que o mesmo é inversamente proporcional à frequência. Como o tamanho da antena é uma fração do comprimento de onda, temos que, ao diminuirmos o comprimento de onda, diminuímos também o tamanho da antena. A diminuição do tamanho da antena impacta em uma redução na área da *tag* no chip, o que reduz bastante o custo. Além disso, com comprimentos de onda menores, é possível ter uma operação em distâncias longas, bem como uma alta taxa de transmissão de dados.

O trabalho em [5] usou como base a arquitetura apresentada na Fig. 3.4.

Pode-se observar que essa arquitetura aborda alguns aspectos ignorados pela arquitetura em [4], como o casamento de impedância entre antena e *tag* em *maching network*, que em geral é realizado fora do chip; a proteção da *tag* em relação à eletricidade estática pelo ESD - *Electrostatic Discharge*; o circuito analógico, que produz a alimentação DC da *tag* e é composto por retificador, limitador e regulador; o transceptor composto pelo modulador e demodulador; o DSP - *Digital Signal Processor*, que realiza o processamento digital dos dados e se comunica com a memória; o gerador de números aleatórios RNG - *Random Number Generator*, utilizado para o algoritmo de anticolisão; por fim, o POR - *Power On Reset*, utilizado para ligar e resetar o DSP da *tag*, e o oscilador - OSC, que controlará o DSP.

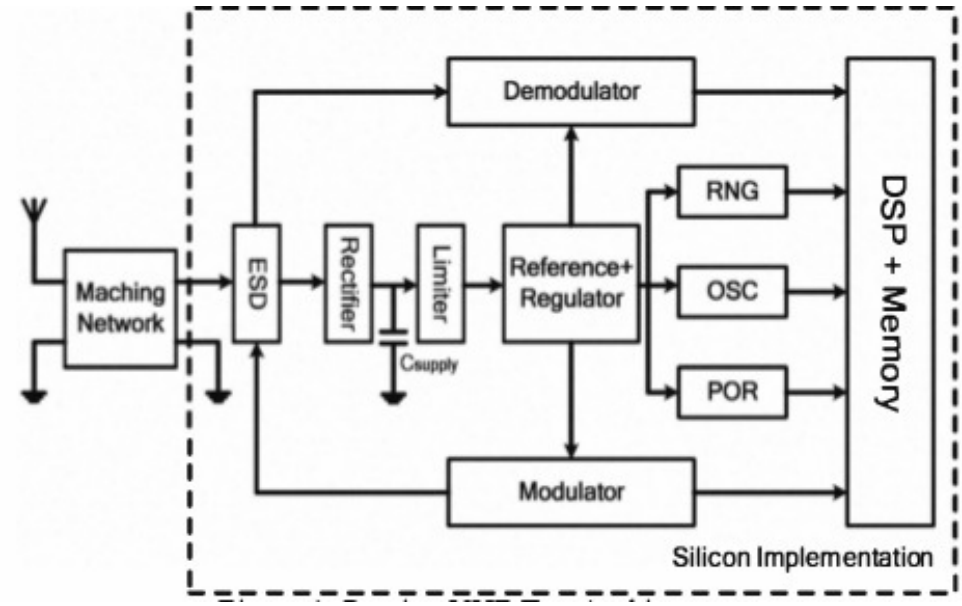

Figura 3.4: Arquitetura de uma *tag* de RFID UHF passiva utilizada em [5].

Os componentes a mais em relação ao trabalho [4] tornam possível entender melhor o funcionamento típico de uma *tag* de RFID. Ela deve ser capaz de produzir sua própria alimentação, utilizando para tal o retificador, limitador e regulador; ter um circuito de proteção para estar protegida de eletricidade estática; fazer a modulação e demodulação do sinal de RF para fornecer uma informação binária ao DSP, e converter a informação enviada pelo DSP em um sinal de RF, que será enviado pela antena; seguir um protocolo de comunicação que exija em algum momento a geração de um número aleatório para evitar colisão; e, por fim, realizar o processamento digital dos dados, sendo necessário em alguns momentos requisitar um acesso à memória.

3.2.3 Processador Banda Base de uma *tag* de RFID de Baixo Consumo

No trabalho apresentado em [6], foi implementado um processador banda base de uma *tag* de RFID de baixo consumo. Esse trabalho também utilizou como base a arquitetura apresentada em 3.2, porém não foi desprezado nenhum bloco desta arquitetura, sendo os mesmos somente adaptados para minimizar o consumo da *tag*.

Antes da implementação do processador banda base da *tag*, [6] apresentou também os principais módulos de um sistema de RFID, conforme mostra a Fig. 3.5.

Ao avaliar a Fig. 3.5, é possível observar o circuito que realiza o casamento de impedâncias tanto no leitor como na *tag*, que é composto por um capacitor e indutor em paralelo, e que a *tag* é composta por um circuito RF, um circuito analógico, um processador banda base e uma memória, envolvendo assim áreas multidisciplinares. Também é possível verificar na Fig. 3.5 como ocorre o fluxo de comunicação entre os circuitos analógicos e de RF com o processador banda base e a memória EEPROM.

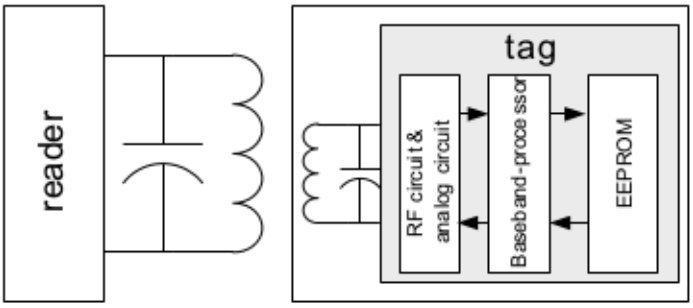

Figura 3.5: Módulos de um sistema de RFID [6].

Como citado anteriormente, em [6] buscou-se adaptar a arquitetura em Fig. 3.2 de forma a torná-la de baixo consumo. As principais alterações realizadas tiveram como objetivo reduzir a tensão de alimentação V_{DD} da *tag*, reduzir a frequência de operação f da *tag*, a quantidade de tarefas realizadas simultaneamente α e a capacitância de carga C_L da *tag*. Estas reduções foram buscadas devido à potência dinâmica consumida ser dada por:

$$P = 0.5 \cdot \alpha \cdot C_L \cdot V_{DD}^2 \cdot f \tag{3.1}$$

Inicialmente, [6] fez uma avaliação da potência consumida por cada um dos blocos da Fig. 3.2 e avaliou quando cada um desses blocos é utilizado, conforme pode ser observado nas Figs. 3.6 e 3.7.

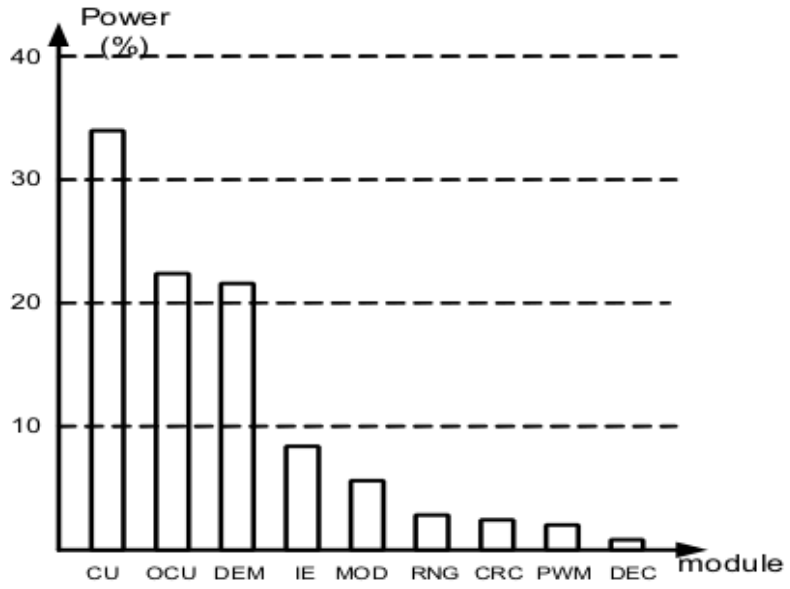

Figura 3.6: Consumo de potência dos blocos em Fig. 3.2 realizado por [6].

A primeira alteração na arquitetura proposta com base nos estudos apresentados nas Figs. 3.6 e 3.7 foi adicionar um módulo de gerenciamente de potência, que desabilita os blocos que não estão sendo utilizados em determinado procedimento. Desta forma, foi adicionada uma variável *enable* em cada unidade funcional, onde essa unidade somente funcionará se a variável *enable* estiver habilitada. Este procedimento reduz consideravelmente a tensão de alimentação da *tag* [6].

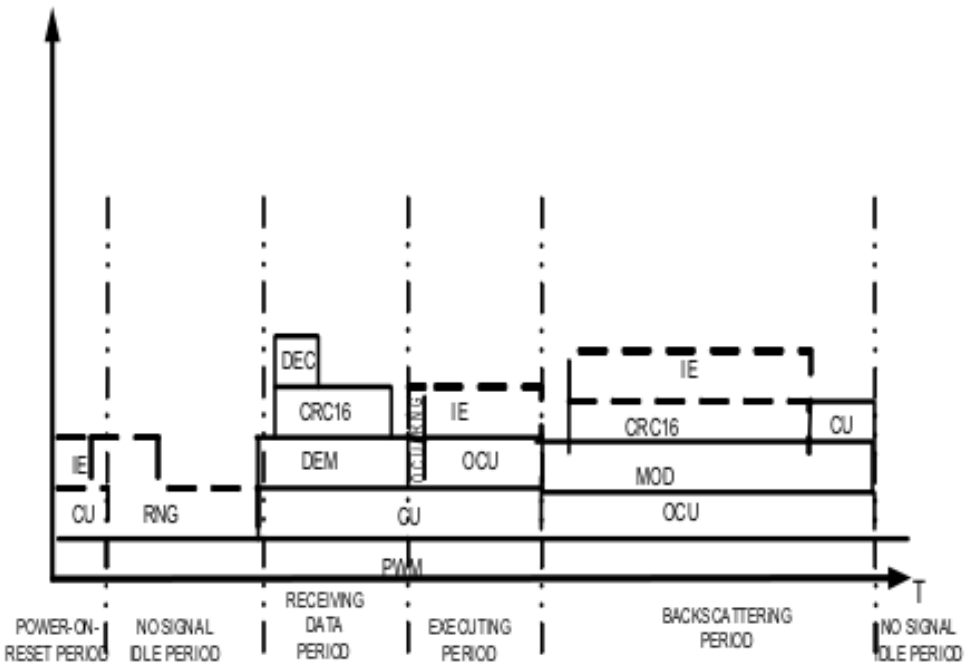

Figura 3.7: Distribuição do trabalho de cada módulo de acordo com o procedimento, estudo realizado por [6].

A segunda alteração consistiu em fazer com que os dados processados na recepção e transmissão dos sinais fossem feitos bit a bit, de modo a evitar o uso de registradores que armazenam todo o dado para depois o mesmo ser processado. Esse procedimento reduziu a área e, consequentemente, a dissipação de potência da *tag*, além de melhorar a eficiência da mesma [6].

Em seguida, foi proposto um método de decodificação parcial para comandos de comprimentos diferentes. Esse método foi baseado no protocolo EPC C1G2, onde os comandos possuem comprimentos variáveis de 2 a 8 bits e são transmitidos serialmente para a *tag* [16]. Desta forma, esses comandos foram decodificados em 1 até 3 passos, conforme pode ser visualizado na Fig. 3.8, utilizando para tal multiplexadores de 2 entradas. Essa alteração reduziu a área da *tag* ao utilizar estruturas mais simples que um decodificador de várias entradas, além de melhorar a velocidade da decodificação, pois evita a utilização de registradores que armazenam grandes quantidades de dados para serem enviados paralelamente para o decodificador [6].

A próxima alteração consistiu em um método alternativo de geração de números aleatórios. O protocolo de anticolisão exige que seja gerado um RN16, número aleatório de 16 bits, durante a operação da *tag*. A qualidade desse número aleatório gerado é essencial para o algoritmo de anticolisão, que garantirá que a comunicação entre leitor e *tag* será realizada corretamente. Em geral, o RNG é realizado usando um LSFR - *Linear Shift Feedback Register*, porém, como citado anteriormente em [6], deseja-se evitar a utilização de registradores para diminuir a potência consumida.

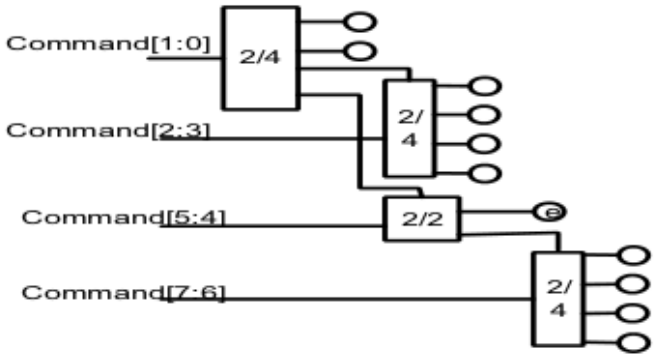

Figura 3.8: Arquitetura proposta para o decodificador em [6].

Desta forma, o novo método para gerar um número aleatório de 16 bits proposto consistiu na operação descrita a seguir. Quando a *tag* cair no campo do leitor, a mesma iniciará no estado reinicialização de energia. Neste momento, o CRC-16 do código eletrônico do produto - EPC - é buscado na memória EEPROM e colocado no gerador de números aleatórios como semente. A partir de então, a detecção de um sinal de baixo nível significará que um novo comando está chegando do leitor. A probabilidade de se obter o mesmo número aleatório entre diferentes *tags* é praticamente desprezível, principalmente devido aos diferentes tipos de semente e diferentes períodos de reinicialização de energia, o que torna esse método um bom gerador de números aleatórios[6]. Isso elimina a necessidade de um LSFR, economizando área e, consequentemente, consumo de potência na *tag*.

Por fim, o último método proposto em [6] consistiu em reduzir a lógica de operações da *tag*, equilibrando com isso o atraso de cada caminho de dados. Tal abordagem foi utilizada para minimizar o número de transistores extras utilizados pela *tag*.

3.2.4 Leitor de RFID UHF

Em [7], foi descrito um projeto de um sistema de RFID compatível com o padrão EPC C1G2, com operação na banda de 915 MHz. O leitor descrito possui um circuito analógico para interface RF, um processador banda base e um controle de clock, conforme pode ser visualizado na Fig. 3.9.

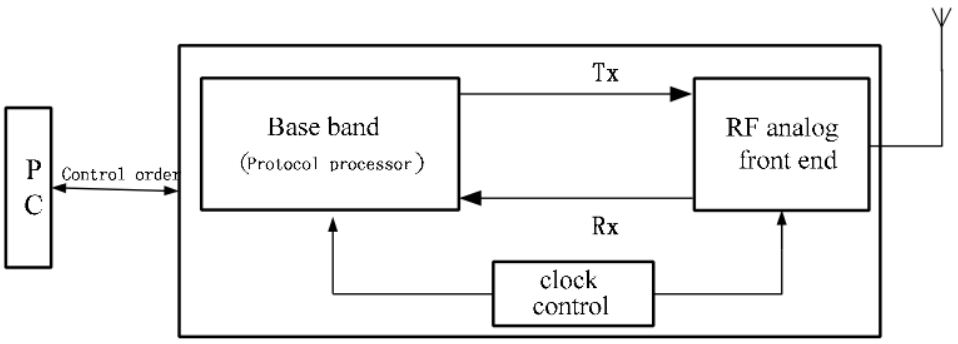

Figura 3.9: Arquitetura proposta para um leitor de RFID [7].

O primeiro componente desse leitor é a interface RF de entrada, também conhecida como AFE - *Analogic Front End*. Esse componente contém um receptor e um transmissor de sinal de RF, um sintetizador de frequência e um circulador, conforme podemos observar na Fig. 3.10 a seguir [7].

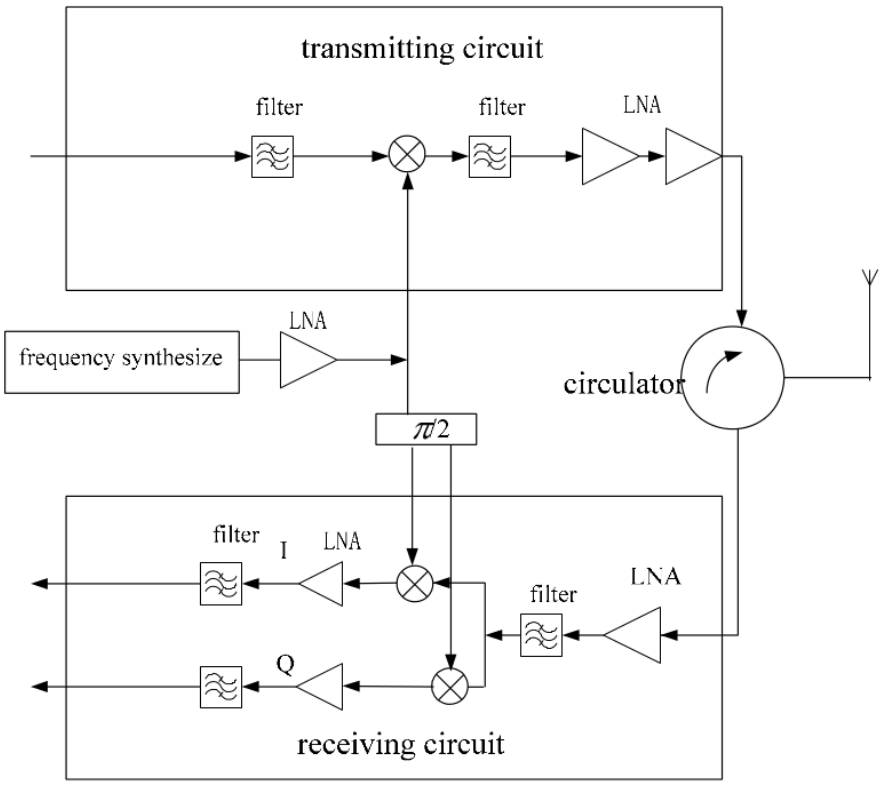

Figura 3.10: Interface RF do leitor em [7].

O circulador é responsável por fazer a isolação do transceptor. O leitor usa a mesma antena tanto para recepção como transmissão do sinal de RF, porém a recepção e transmissão do sinal de RF não podem ocorrer ao mesmo tempo, já que isso comprometeria o desempenho do leitor [7]. Basicamente, o circulador alterna entre esses dois estágios, permitindo em alguns momentos o uso da antena como transmissor e em outros como receptor [7].

O transceptor do leitor de RFID é composto do circuito transmissor e do circuito receptor. Como é necessário que o sinal transmitido seja capaz de ativar as *tags* de RFID em uma longa distância, a potência do sinal do circuito transmissor será muito maior que a do circuito receptor. Esse fato faz com que seja necessário um circulador que consiga um bom isolamento para a potência de transmissão para que a mesma não interfira na recepção dos sinais de RF [7].

O transmissor recebe um sinal de frequência intermédiaria - IF - vindo do processador banda base e inicialmente faz uma filtragem utilizando um filtro LC para eliminar ruídos espúrios e de quantização. Então, esse sinal IF filtrado é multiplicado por uma portadora, para que assim seja gerado um sinal de UHF. O sinal de UHF gerado passa por um filtro RF para reduzir interferências, tais como os sinais de vazamento do oscilador e sinais de frequências espelhados, e é amplificado por um amplificador de potência para que adquira potência suficiente para ativar a *tag* em uma longa distância. Após esses processos, o sinal modulado é enviado ao circulador e deste para a antena, que irá transmitir o sinal para o ar [7].

Já o receptor recebe um sinal de RF da *tag*, filtra esse sinal para eliminar interferências adquiridas durante a transmissão do mesmo, divide esse sinal em I e Q e, por fim, passa esses sinais em um amplificador de baixo ruído LNA - *Low Noise Amplifier*, filtrando em seguida para transformá-los em um sinal IF [7]. Esse processo torna o sinal recebido pela *tag* apto para ser tratado no processador banda base.

O sintetizador de frequência, por sua vez, é o responsável por gerar a portadora utilizada nos blocos descritos anteriormente.

O processador banda base apresentado na Fig. 3.11 possui um modulador, demodulador, conversores DAC e ADC, unidade de controle, memórias, Ethernet para comunicação com o PC, etc. Esse processador tem como principal função realizar o protocolo de comunicação entre a *tag* e o leitor. O leitor projetado em [7] seguiu o protocolo de comunicação EPC C1G2.

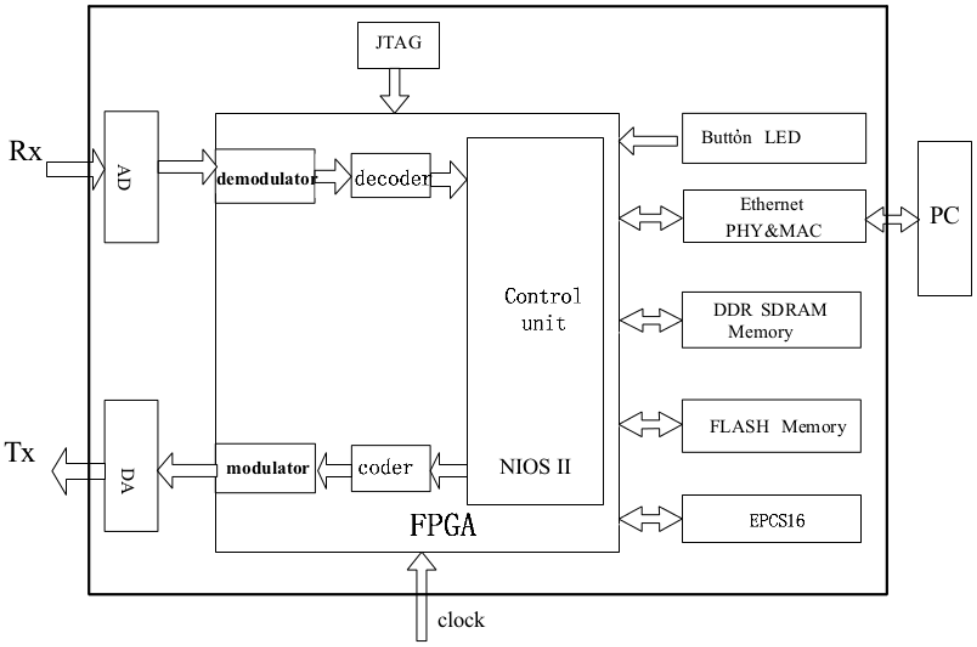

Figura 3.11: Processador banda base do leitor descrito em [7].

O protocolo EPC C1G2, já citado anteriormente nas arquiteturas anteriores, define as características de um sistema RFID passivo. Nesse protocolo, o leitor envia uma informação para uma ou mais *tags* através da modulação de um sinal de RF utilizando um modulador que pode ser DSB-ASK (*Double-Sideband Amplitude Shift Keying*), SSB-ASK (*Single-Sideband Amplitude Shift Keying*) ou PR-ASK (*Phase-Reversal Amplitude Shift Keying*) [16].

Quando o sinal é enviado para a *tag*, é aproveitado pela mesma para prover sua alimentação. O leitor detecta a presença da *tag* através do envio do sinal de RF da portadora não modulada. Esse sinal é enviado continuamente, já que o leitor é quem fornece a fonte de alimentação da *tag*. Ao receber esse sinal, a *tag* responde enviando uma resposta, que é obtida modulando a portadora na amplitude e/ou fase [16].

A arquitetura apresentada na Fig. 3.11 pode ser implementada para outros padrões de sistemas de RFID, não sendo restrita ao padrão EPC C1G2. O processador banda base é dividido em diversos módulos. O conversor A/D converte o sinal IF gerado pelo AFE em um sinal digital, que é demodulado e decodificado através do demodulador, e decodificado. Em seguida, esse sinal vai para unidade de controle, que realiza os comandos e controles que permitem o processo de comunicação entre leitor e *tag*. Enquanto o sinal permanece sendo processado pela unidade de controle, o acesso e controle da memória realizados pelo módulo DDR SDRAM podem ser requeridos, bem como a execução de comandos e controles enviados pelo PC através do módulo de ethernet. Por fim, o sinal digital é enviado a um codificador PIE - *Pulse-Interval Encoding* e, posteriormente, para um modulador. Esse sinal é então convertido para um sinal de IF através do conversor D/A, sendo enviado ao AFE [7].

Por fim, tem-se o módulo que controla o *clock* responsável por gerar os sinas adequados para cada um dos módulos e sub-módulos do sistema, como, por exemplo, o amostrador do conversor A/D, a unidade de controle, etc.

A arquitetura do sistema de RFID descrito no capítulo 4 que foi modelado neste trabalho de graduação, foi obtida através do estudo e análise das arquiteturas apresentadas acima. Conforme ficará mais claro no capítulo 4, optou-se por utilizar uma interface de memória o que permite a utilização de uma memória externa ao sistema, deixando o sistema mais adaptável a diferentes tipos de aplicações. A comunicação entre a interface de memória e a memória será baseada na comunicação I2C. Cabe ressaltar que este tipo de protocolo de comunicação é um dos mais simples existentes no mercado e um dos mais difundidos devido a suas inúmeras vantagens.

Apesar deste protocolo não ser específico para uma arquitetura de RFID, ele também pode ser utilizado neste tipo de sistema, é como este será o caso neste trabalho, o protocolo de comunicação I2C será explicado na seção a seguir.

3.2.5 Protocolo de comunicação I2C

O protocolo de comunicação I2C foi desenvolvido em 1966 pela *Philips Semiconductor* e ultimamente está extremamente difundido no mercado, servindo para interconectar uma gama extremamente ampla de dispositivos eletrônicos, tais como microcontroladores e microprocessadores, drivers de LCD, portas I/O, memórias RAM e EEPROM, conversores de dados, etc [8].

Este protocolo de comunicação é executado utilizando apenas duas vias de comunicação bidirecionais: um barramento de dados serial (SDA) e um barramento de clock (SCL). Ambos os barramentos são conectados a uma fonte de alimentação através de um resistor de *pull-up*, de forma que, quando o barramento estiver livre, as linhas ficarão no nível lógico alto [8]. Este barramento está representado na Fig. 3.12, que inclusive mostra que este protocolo de comunicação serve para diversos valores de alimentação, maximizando assim a quantidade de dispositivos que podem ser conectados ao mesmo.

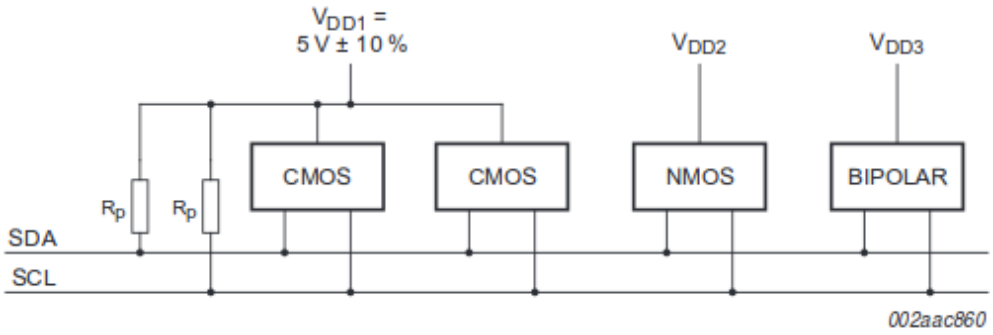

Figura 3.12: Barramento utilizado para comunicação I2C [8].

Este protocolo pode ser utilizado para comunicação com diversos tipos de dispositivos, sendo que a quantidade de dispositivos é limitada apenas pela capacitância máxima do barramento, que é de aproximadamente 400 pF. Cada dispositivo interligado nestes dois barramentos possui um endereçamento próprio e pode atuar como receptor ou transmissor do sinal. Isso pode variar dependendo da natureza do dispositivo. Por exemplo, um driver de LCD atua somente como receptor [8].

Para que a comunicação I2C seja realizada, são necessárioa um mestre, que é um dispositivo que inicia a comunicação, gera um sinal de clock e encerra a comunicação, e um escravo, que é o dispositivo que pode agir como receptor, obtendo dados do barramento, ou como transmissor enviando dados para o barramento [8]. Neste trabalho, a interface com a memória será o mestre e a memória será o escravo, que em alguns momentos agirá como receptor e em outros, como transmissor.

O dado no barramento SDA tem que ser estável no nível alto do barramento de clock, podendo ser alterado somente no nível baixo, conforme pode ser verificado na Fig. 3.13. As condições que indicam o início e final do processo de comunicação são os sinais de *start* e *stop*. O sinal *start* é caracterizado por uma variação do nível alto para baixo no barramento SDA quando o barramento SCL está em nível alto. O sinal *stop* caracteriza-se por uma variação em SDA de nível lógico baixo para alto quando o barramento SCL está em nível baixo. Os comandos de *start* e *stop* podem ser observados na Fig. 3.14. Quando o barramento detecta o sinal de *start*, ele fica ocupado, e quando detecta o sinal de *stop*, fica livre [8].

Após o comando de *start*, o endereço que o barramento I2C irá acessar é enviado seguido de 1 bit, que indica se a operação realizada será de leitura ou de escrita (1 no caso de leitura e 0 no caso de escrita). Em seguida, é enviado o sinal de ACK (*Acknowledge*) do dispositivo que foi endereçado. Esse sinal é um bit em nível baixo, já que o barramento na condição inicial está em nível alto. Então, o dado é enviado e, a cada pacote de dado transmitido, um sinal de ACK ou NACK (*Not Acknowledge*) é enviado, indicando se o dado foi recebido ou enviado com sucesso, caso o sinal seja ACK, ou que houve falha na comunicação, caso o sinal seja NACK. Quando se desejar parar a comunicação, será enviado o sinal *stop* [8].

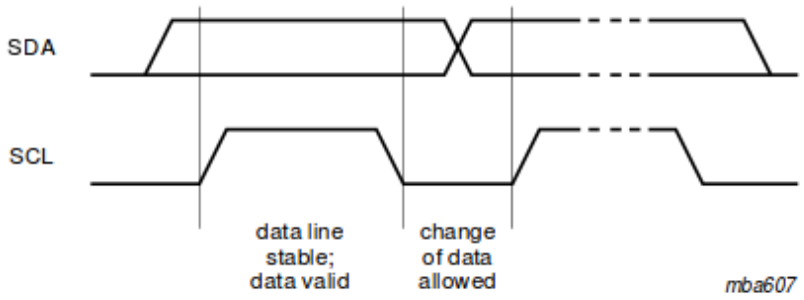

Figura 3.13: Forma que deve ocorrer o envio de dado no protocolo I2C [8].

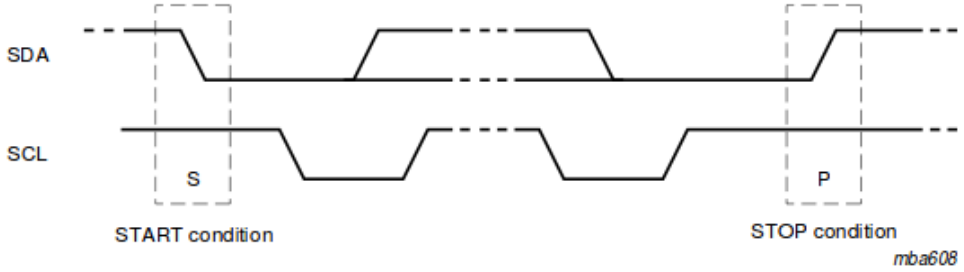

Figura 3.14: Comando de *start* e *stop* no protocolo I2C [8].

Conforme dito anteriormente, o barramento é bidirecional. Porém, o mestre e o escravo não podem enviar dados para o barramento de forma simultânea. Quando o mestre está enviando um sinal pelo barramento, o escravo apenas recebe esse sinal, e quando o escravo envia um sinal para o barramento, o mestre apenas recebe o sinal. As operações de leitura e escrita definem os momentos em que o dado será enviado pelo mestre ou pelo escravo. As Figs. 3.15 e 3.16 apresentam a distinção entre os comandos de leitura e escrita no protocolo I2C [8].

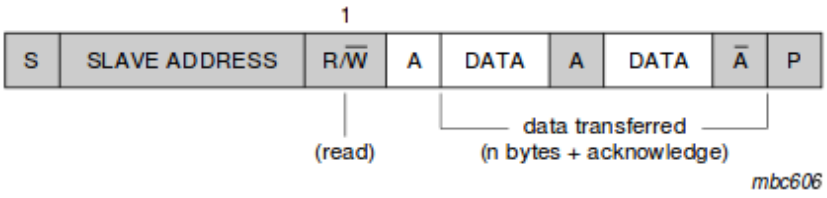

Figura 3.15: Comando de leitura no protocolo I2C [8].

Neste ponto, já é possível perceber que o I2C é um protocolo simples, rápido e eficiente. Desta forma, o mesmo pode ser facilmente ser utilizado para realizar procedimentos de leitura e escrita na memória, sendo este o principal motivo desse protocolo ter sido escolhido neste trabalho.

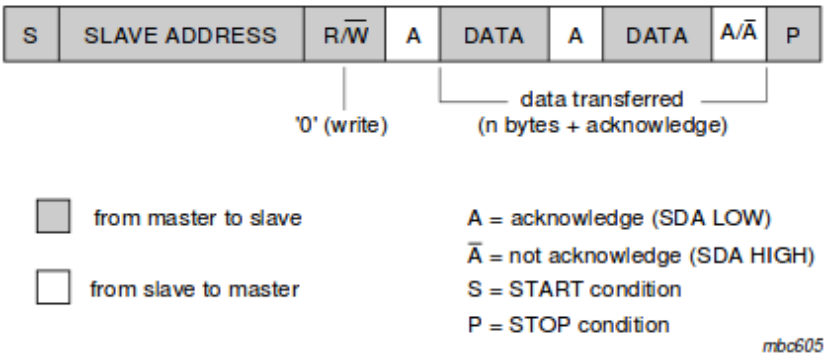

Figura 3.16: Comando de escrita no protocolo I2C [8].

A proposta deste trabalho é a modelagem e simulação de uma *tag* de RFID passiva de propósito geral como estudo de caso utilizando a ferramenta SystemC/SystemC-AMS, considerada apropriada para modelagem de um sistema heterogêneo em alto nível de abstração. Com isso, será possível validar a funcionalidade do sistema e identificar possíveis limitações para tornar viável a proposta de modificações que tornem essas arquiteturas mais robustas ainda em um estágio inicial do fluxo de projeto.

Capítulo 4

Metodologia para Modelagem de um sistema de RFID

4.1 Introdução

O presente capítulo irá apresentar a modelagem em nível sistêmico de arquiteturas de RFID baseadas nas arquiteturas levantadas no capítulo 3. Conforme citado no capítulo 2, a modelagem será realizada utilizando a ferramenta SystemC/SystemC-AMS, que é baseada em modelos de computação.

Essas modelagens serão realizadas de forma hierárquica, onde primeiramente serão descritas as funcionalidades dos blocos básicos que compõem o sistema e, posteriormente, será realizada a instanciação e interconexão dos mesmos.

Ao todo, serão apresentadas duas versões de arquitetura, onde a primeira é completamente baseada na arquitetura de *tag* apresentada em [4], sendo a mesma implementada utilizando apenas um modelo de computação e a segunda consiste na adição de um protocolo de comunicação, simulação de ruído e acesso a memória via uma interface de memória, sendo que nesta *tag* a implementação utiliza diferentes modelos de computação.

Todas as duas versões foram desenvolvidas em SystemC-AMS. Os resultados dessas implementações são apresentados no capítulo 6, e a versão completa dos códigos utilizados estará presente nos anexos.

4.2 Arquiteturas modeladas

4.2.1 Primeira Versão

A primeira versão modelada consiste na arquitetura de uma *tag* simples que possui blocos funcionais semelhantes aos modelados em [4], mostrados na Fig. 4.1 e descritos a seguir.

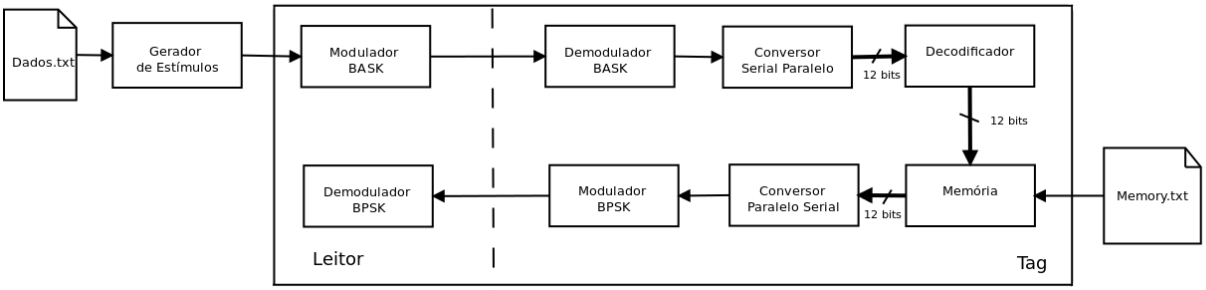

Figura 4.1: Arquitetura completa da primeira versão desenhado com o auxílio do *software* Dia [9].

O gerador de bits seriais é o bloco responsável por gerar os estímulos de teste da *tag*. Ele lê sequências de caracteres hexadecimais de um arquivo de texto, converte-os para binário e os envia serialmente.

O gerador de onda senoidal é responsável por gerar os sinais de portadora, ou seja, as ondas senoidais que serão utilizados na modulação.

O modulador BASK - *Binary Amplitude Shift Keying* converte a saída binária obtida do gerador binário para uma saída de RF modulada na amplitude. O modulador e o gerador senoidal são partes presentes no leitor que será modelado nessa versão devido a necessidade da entrada na *tag* ser uma onda de radio frequência modulada. Este modulador funciona como um multiplexador, onde a saída será uma onda senoidal de amplitude de 5V, caso a entrada binária seja igual a 1, e uma onda senoidal de amplitude 1.8V, caso a entrada binária seja 0, conforme pode ser visualizado na Fig. 4.2. Cabe ressaltar que optou-se por representar o bit 0 com uma onda de amplitude menor devido a alimentação da *tag* ser obtida através do sinal enviado pelo leitor, ou seja, se este sinal fosse zero poderia ocorrer uma queda na alimentação da *tag* que prejudicaria o funcionamento correto da mesma.

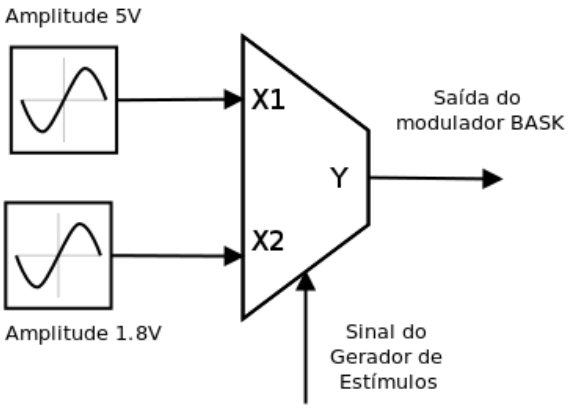

Figura 4.2: Modulador BASK desenhado com o auxílio do *software* Dia [9].

Em seguida, na *tag*, temos um demodulador BASK, que consiste em um retificador para refletir a parte negativa do sinal de entrada modulado para parte positiva deste sinal, um filtro passa-baixa, para eliminar ruído de alta frequência e tornar mais suaves as variações de amplitude do sinal, e um amostrador com histerese e com região de indesição, responsável por amostrar o sinal

a uma determinada taxa e compará-lo com uma tensão limiar, de forma a obter o sinal binário demodulado. Cabe ressaltar que este amostrador deve possuir histerese de forma a evitar as regiões de mudança de sinal e deve possuir região de indesição visando melhorar a imunidade a ruído. Para melhor visualização, esse demodulador é apresentado na Fig. 4.3.

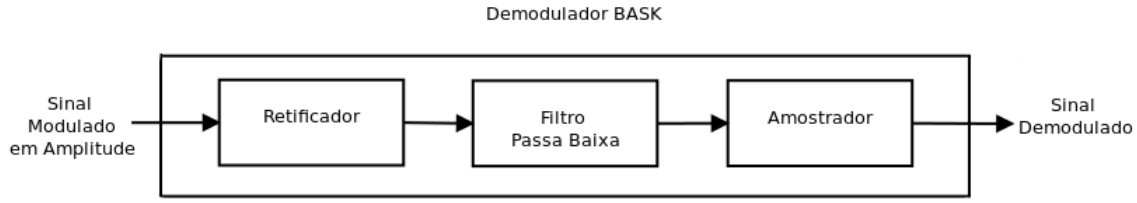

Figura 4.3: Demodulador BASK desenhado com o auxílio do *software* Dia [9].

O próximo bloco é um conversor serial-paralelo responsável por transformar a entrada serial em saídas digitais paralelas de 12 bits, correspondentes ao tamanho escolhido para enviar algum comando a *tag*.

A saída desse conversor está conectada na entrada do decodificador. Caso esse bloco binário seja igual a alguma das sequências pré estabelecidas no decodificador, a *tag* efetuará uma ação de leitura, com cada comando consistindo na leitura de um endereço diferente de memória.

Desta forma, será enviado para a memória o endereço no qual será efetuada a leitura. A memória, que nesta versão foi implementada como um arquivo de texto com informações hexadecimais, será lida sequencialmente, principalmente devido as limitações de leitura e escrita em arquivos em C++, até que se atinja o endereço que contém a informação que deve ser enviada.

A informação sai da memória de forma paralela e é enviada para o leitor. Porém, para que isso ocorra, a mesma precisa passar em um conversor paralelo-serial para que a mesma possa ser enviada de forma serial para o modulador BPSK - *Binary Phase Shift Keying*.

O modulador BPSK, por sua vez, recebe esse dado é modula o sinal utilizando duas ondas senoidais defasadas entre si em 180°, sendo que uma delas representa o sinal binário 1 e a outra, o sinal binário 0, conforme pode ser visto na Fig. 4.4.

Por fim, tem-se o demodulador BPSK, que se encontra presente no leitor e foi modelado nesta versão para facilitar a conferência dos dados que a *tag* está enviando ao leitor.

O demodulador consiste em um multiplicador que irá multiplicar a portadora recebida pelo leitor por uma onda senoidal de fase 0°. Desta forma, quando o sinal recebido pelo leitor estiver com fase 0°, o sinal de saída do multiplicador será $sin^2(x)$. Porém, quando o sinal estiver defasado em 180°, o que corresponde a $-sin(x)$, a saída da multiplicação será de $-sin^2(x)$. Como qualquer número real elevado ao quadrado é sempre positivo, temos que, para uma onda senoidal de fase 0°, a saída do multiplicador será maior ou igual a zero, e que, para uma onda senoidal de fase 180°, a saída do multiplicador será menor ou igual a zero. Em seguida, é realizada uma integração por período, de forma a afastar a saída do multiplicador do zero. Por fim, o sinal de saída do integrador é amostrado a uma certa taxa, sendo em seguida comparado com o limiar 0. Caso o resultado seja maior que zero, o sinal recebido é 1, e caso seja menor que zero, o sinal recebido é 0.

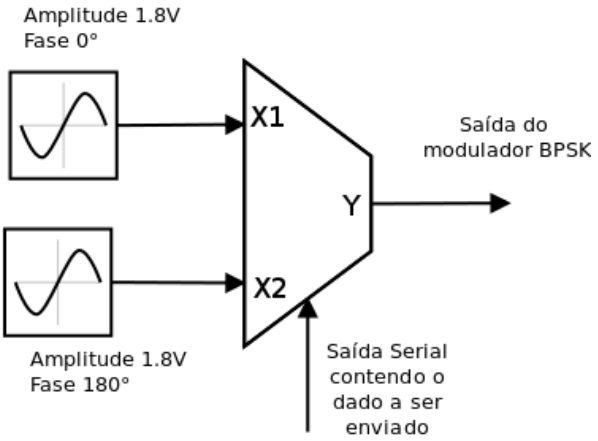

Figura 4.4: Modulador BPSK desenhado com o auxílio do *software* Dia [9].

Para melhor visualização, os blocos que efetuam o processo de demodulação BPSK estão apresentados na Fig. 4.5.

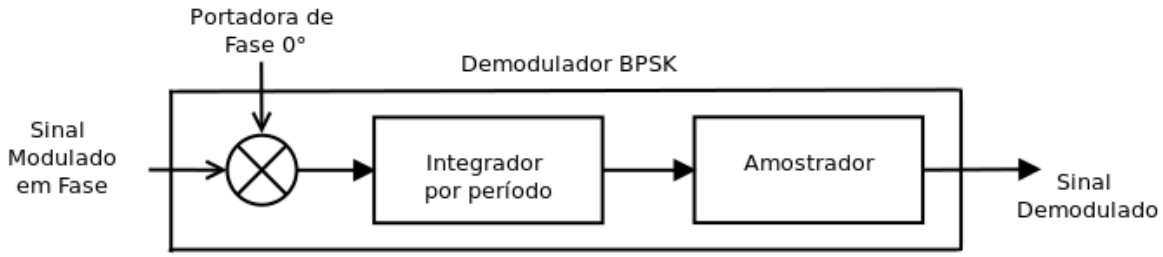

Figura 4.5: Demodulador BPSK desenhado com o auxílio do *software* Dia [9].

4.2.2 Segunda Versão

A segunda versão refina a primeira através da inserção de ruído entre a comunicação entre a *tag* e o leitor para simular os ruídos adquiridos pelo sinal de RF quando o mesmo passa pelo ar. Além disso, o acesso a memória é realizado através de uma interface, permitindo um acesso de forma direta, ou seja, sem ser sequencial. Existe também um refinamento na bloco amostrador do sinal, na qual é adicionado uma região de indefinição para que este bloco se assemelhe mais ao modelo real. E, por fim, é feita uma completa alteração na comunicação digital da *tag*, de modo a permitir que as operações de leitura e escrita na *tag* sejam feitas com maior controle do endereço que se deseja ler ou escrever, bem como da *tag* na qual a operação deve ser executada.

No quesito implementação, a segunda versão difere quase totalmente da primeira. Somente o gerador de estímulos permanece o mesmo. Apesar do conceito de como os moduladores e demoduladores BASK e BPSK foram implementados ser idêntico ao da primeira versão, o modelo de computação utilizado para descrever estes modulos foi alterado. Desta forma, essas partes não serão explicadas novamente nesta seção, ficando subentendido que o funcionamento destes blocos é igual ao da primeira versão.

A única alteração relevante nos blocos analógicos que compõem esta versão foi a adição do ruído gaussiano branco e de uma atenuação no sinal que sai dos moduladores e chega no demodulador. Para tal, multiplicou-se o sinal de saída do modulador por uma constante maior que 0 e menor que 1 para simular a atenuação do sinal e utilizou-se a função randômica para obtenção de um ruído gaussiano com a variância indicada na instanciação, conforme pode ser visualizado na Fig. 4.6. A forma com que o ruído gaussiano branco é gerado será explicada detalhadamente na seção 5.3.2.

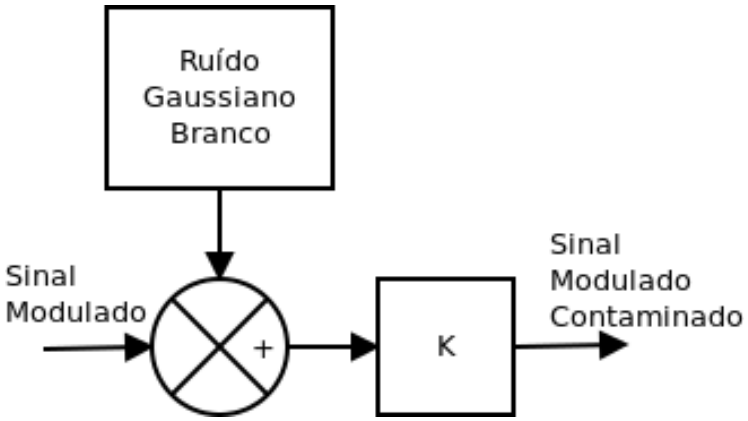

Figura 4.6: Diagrama que ilustra a inserção de ruído e atenuação, onde K é uma constante maior que zero e menor que um, utilizada para atenuar o sinal de entrada, este diagrama foi desenhado com o auxílio do *software* Dia [9].

Os blocos digitais, por sua vez, foram completamente remodelados, sendo necessário entender como irá ocorrer a comunicação entre o leitor e a *tag* antes de detalhar a forma com que eles funcionam nesta versão. Cabe ressaltar que todas as informações consideradas abaixo possuem 4 bits, sendo esta escolha realizada apenas para facilitar a verificação e validação do modelo. Esses dados podem ser ampliados posteriormente, visando a adaptação de outras funcionalidades, se necessário.

Inicialmente, o leitor enviará um comando de ativação para *tag*, seguido de diversos 1's. Quando a *tag* receber este comando de ativação e decodificar o mesmo, ela irá imediatamente ativar um gerador de números aleatórios, que só será desativado quando receber o primeiro 0. Durante esse processo, nenhum comando de leitura ou escrita será requisitado da memória, e nenhum dado será enviado para o leitor.

Após essa etapa, a máquina de estados da *tag* escreve o número aleatório que foi gerado no endereço 2 da memória e envia para o leitor um comando de 16 bits, que contém informação do comando ao qual a *tag* está respondendo, o endereço que a *tag* está manipulando, o número aleatório gerado pela *tag* e a confirmação de que o dado foi escrito.

Este comando de 16 bits enviado para o leitor sempre possuíra a estrutura indicada acima. A única diferença é que, caso seja requisitada uma leitura na memória, ele irá enviar o dado que foi lido no lugar da confirmação de que o dado foi escrito.

Em seguida, ainda no comando de ativação, a máquina de estados da *tag* irá requisitar a leitura do endereço 1 da memória, enviando o conteúdo desse endereço para o leitor. Cabe ressaltar que esse é o único endereço de memória que só tem permissão de leitura. Essa particularidade se deve a esse endereço ser destinado a armazenar o identificador da *tag*, que é uma informação que não pode ser sobrescrita.

Após receber o comando de ativação e concluir o envio das informações citadas acima, a *tag* continuará ativa a espera de um novo comando, que pode ser de leitura, escrita ou um comando que irá matar a *tag* encerrando a execução da mesma. Caso a *tag* não receba o comando *kill*, ela só irá parar a execução quando a alimentação for cortada.

Os comandos de leitura, escrita ou *kill* só serão executados caso o número aleatório enviado pelo leitor coincida com o número aleatório da *tag*. Isso permite que o leitor consiga se comunicar com diversas *tags*, tendo total controle de qual deverá ser a *tag* que responderá ao comando.

O comando de leitura enviado pelo leitor consiste em 4 bits reservados para o comando, 4 bits reservados para o endereço de memória que deve ser lido e 4 bits com o número aleatório da *tag* na qual se está requisitando a leitura da memória. Esse comando recebe como resposta da *tag* o comando padrão de leitura.

Já o comando de escrita difere do comando de leitura apenas no final, onde são enviados mais 4 bits com o dado que deve ser escrito no endereço e a resposta da *tag* a este comando, que é igual a resposta padrão de escrita.

O comando *kill* é composto apenas do comando que simboliza o *kill* e de mais 4 bits que correspondem ao número aleatório da *tag*. Assim que detecta esse comando, caso o número aleatório seja igual ao da *tag* em questão, a mesma entra em um estado de *stand by*, no qual aguarda novamente o recebimento de um comando de inicialização. Caso contrário, a *tag* não faz nada e espera a chegada de um novo comando.

A comunicação entre o leitor e a *tag* pode ser visualizada mais claramente através da Fig. 4.7.

Para executar a comunicação acima, são necessários os seguintes blocos digitais: um decodificador, um gerador de números aleatórios, uma unidade de controle, uma unidade de controle de saída, uma interface com a memória e a memória. A comunicação entre esses blocos exposta na Fig. 4.9 e bastante semelhante com a arquitetura *Harvard* apresentada na Fig. 3.2.

Antes de explicar o funcionamento dos blocos citados acima, será apresentada a forma com a qual será realizada a leitura e escrita da memória. A comunicaçao adotada entre a interface de memória e a memória será baseada no protocolo I2C, porém neste caso não haverá o barramento bidirecional de clock, já que o sistema será modelado em alto nível. Desta forma, os comandos de *start* e *stop* na memória serão simulados através de uma *flag* que irá habilitar ou desabilitar a memória.

Outra mudança é que o barramento bidirecional de dados será dividido em dois barramentos, um que contém apenas os dados de saída da interface e outro com apenas os dados de saída da memória.

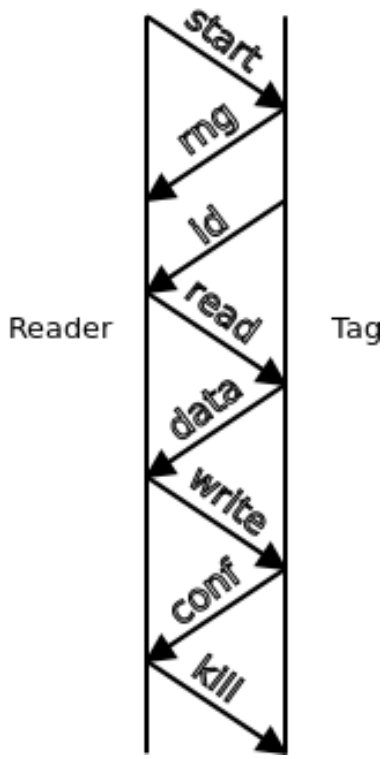

Figura 4.7: Comunicação entre o Leitor e Tag desenhado com o auxílio do *software* Dia [9].

Essa divisão no barramento bidirecional de dados facilita o entedimento da interação entre a memória e a interface, já que com eles separados é mais fácil visualizar qual foi a resposta enviada e recebida por cada bloco. Porém, se os dois barramentos forem mesclados, eles se comportarão da mesma forma que o barramento bidirecional.

Conforme citado no capítulo 3, a interface enviará o comando que será realizado na memória depois de receber o comando de *start* no protocolo I2C. A memória responde a esse comando com um sinal de um bit que indicará se o mesmo foi ou não foi recebido. Caso o comando tenha sido recebido, a interface envia o endereço que deseja acessar, recebendo novamente um dado de confirmação de que o endereço foi recebido da memória.

Por fim, caso o comando requisitado seja uma escrita, a interface enviará o dado que deve ser escrito na memória e a memória irá responder com o sinal indicando se a escrita foi realizada ou se apresentou alguma falha. Caso o comando enviado pela interface seja uma leitura, a memória irá iniciar o envio do dado contido no endereço após confirmar o recebimento do mesmo e, assim que receber esse dado, a interface irá enviar um sinal de confirmação para memória. A Fig. 4.8 é apresenta como esta comunicação é realizada.

Agora que já está elucidado como ocorrerá a comunicação entre a *tag* e leitor e entre a interface e a memória, será explicado o funcionamento dos blocos que compõem esta arquitetura, conforme pode ser visto na Fig. 4.9.

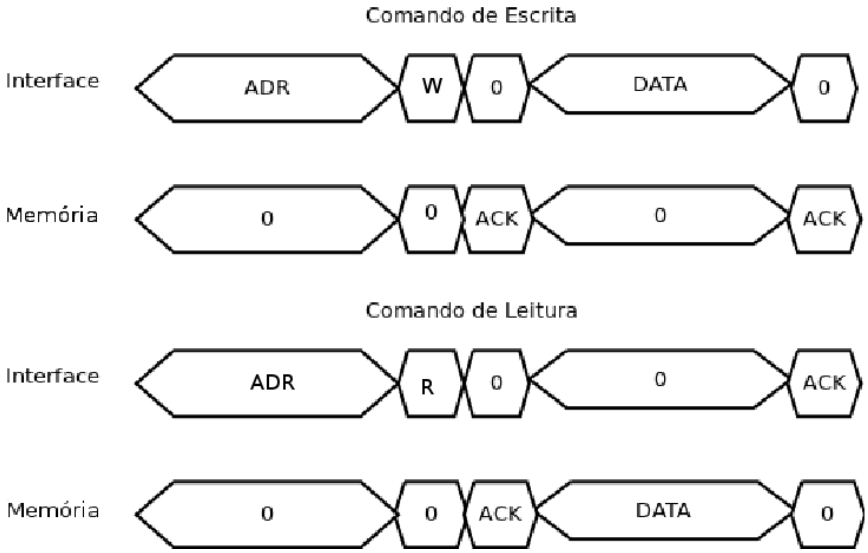

Figura 4.8: Comunicação entre interface e memória da segunda versão da *tag* desenhado com o auxílio do *software* Dia [9].

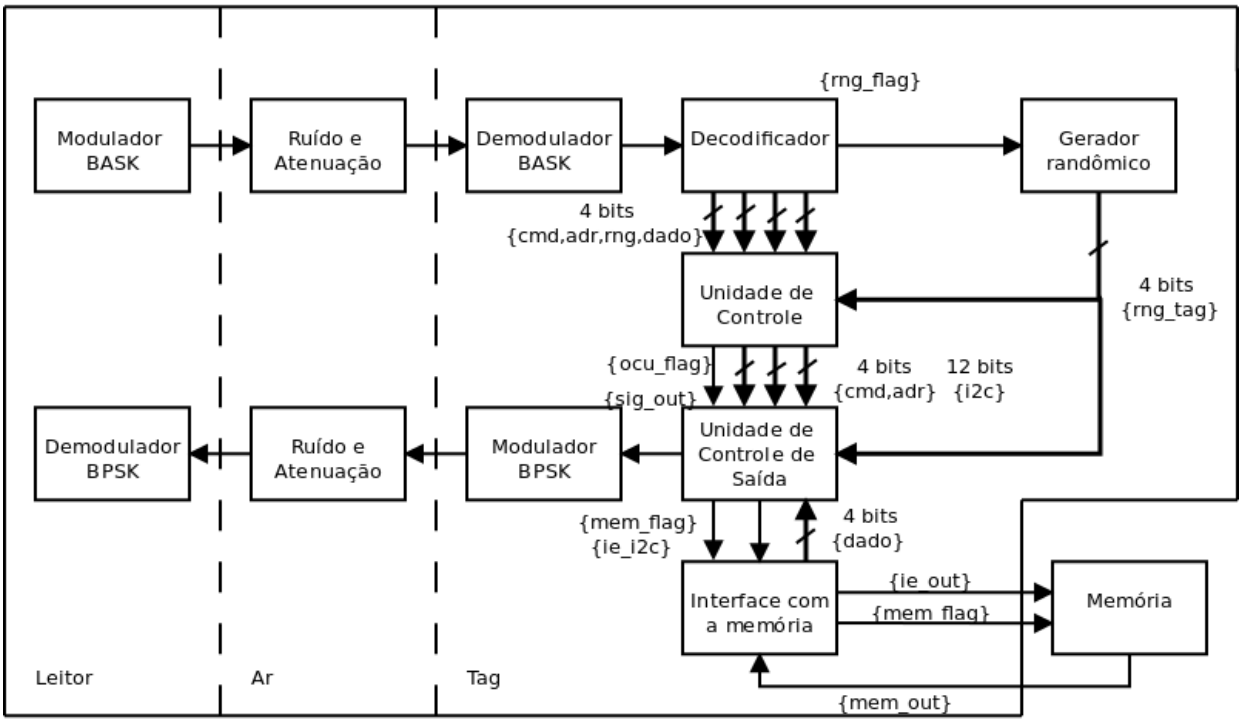

Figura 4.9: Módulos que compõem a segunda versão da *tag* desenhado com o auxílio do *software* Dia [9].

O sinal de saída do demodulador BASK entrará no decodificador, que será responsável por paralelizar a informação recebida separando os bits em saídas de comando, endereço, número aleatório e dado, todos contendo 4 bits. Além dessas saídas, haverá também como saída a *flag* de ativação do gerador de números aleatórios.

Enquanto esta *flag* de ativação estiver em 1, o gerador de números aleatórios estará funcionando. Quando a *flag* for desabilitada, o número aleatório gerado pelo gerador será enviado para a unidade de controle e para a unidade de controle de saída.

Em seguida, o sinal separado pelo decodificador irá entrar na unidade de controle, que é uma máquina de estados que inicialmente está em *stand by*, e, caso receba um comando de inicialização, inicia a comunicação descrita acima. Essa máquina de estados possui os estados, *stand by*, *start*, *init*, *write*, *read* e *kill*, sendo que os três últimos estados são executados somente quando o número aleatório recebido pela *tag* é igual ao número aleatório gerado pela *tag*.

O diagrama que explica a transição dos estados baseando-se na comunicação descrita acima pode ser visualizado na Fig. 4.10.

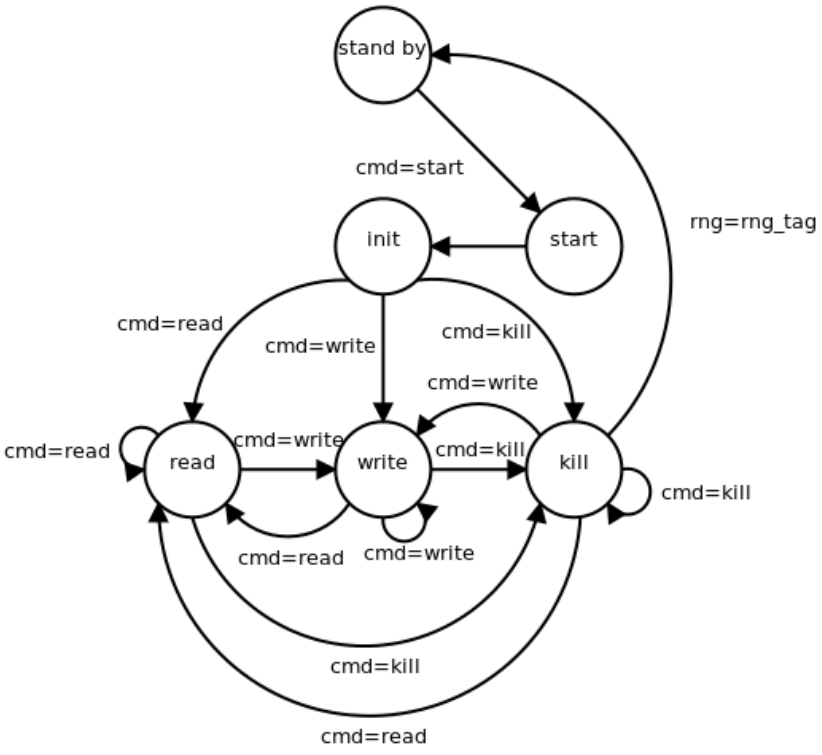

Figura 4.10: Máquina de estados que descreve o comportamento da segunda versão da *tag* desenhado com o auxílio do *software* Dia [9].

A saída da máquina de estados será um comando paralelo que durará 16 pulsos de clock e será composto de 4 bits para comando de leitura ou escrita na memória, 4 bits para o endereço e, caso o comando requisitado solicite escrita na memória, mais 4 bits contendo o dado a ser escrito. Além dessa saída, existe uma flag que irá ativar a unidade de controle de saída visando garantir o sincronismo no tratamento de dados da mesma.

A unidade de controle da saída recebe como entradas a saída da unidade de controle, a saída do gerador de números aleatórios, as saídas de comando e endereço do decodificador e a saída de dado da interface de memória. Esse bloco é responsável por organizar os dados que serão enviados para a interface de memória e para o modulador BPSK. Cabe ressaltar que este bloco só realiza alguma operação caso a sua *flag* de ativação esteja em nível lógico alto e também possui como saída deste bloco uma *flag* responsável por ativar a interface de memória.

A saída que será enviada para a interface de memória é simplesmente uma saída serial obtida através da conversão paralelo serial da saída da unidade de controle. Já a saída que será enviada para o modulador será o formato padrão de resposta da *tag* citada acima, que consiste no comando enviado pelo leitor, no endereço de memória manipulado, no número aleatório da *tag* e, por fim, na confirmação de que o dado foi escrito, no caso de comando de escrita, ou no dado lido pela *tag*, no caso de comando de leitura.

A interface de memória possui como entradas a saída da unidade de controle de saída e a saída da memória. Esse bloco é responsável por organizar o comando serial oriundo da unidade de controle de saída de forma a se adaptar ao protocolo de comunicação baseado no I2C que está sendo utilizado neste trabalho.O comando de comunicação gerado é enviado serialmente para memória, que responde de acordo com o protocolo de comunicação I2C.

Da saída da memória, a interface obtém a confirmação de escrita do dado ou qual é o dado que foi lido da memória e envia essa informação para a unidade de controle de saída de forma paralela. Nesse bloco, há também uma saída que consiste na *flag* de ativação da memória, visando garantir o sincronismo entre esses dois blocos.

Por fim, tem-se a memória que recebe o comando baseado no protocolo I2C da interface com a memória e gera como saída uma resposta também baseada no protocolo I2C. Essa memória foi implementada utilizando um vetor com um número fixo de posições, neste caso, 16 posições de memória, onde o endereço equivale ao índice do vetor. Essa implementação tornou mais direto o processo de leitura e escrita da memória.

Após voltar para a unidade de controle de saída, a resposta da *tag* está pronta para ser modulada pelo modulador BPSK e enviada para o leitor. A segunda versão termina com a demodulação do comando no leitor.

Capítulo 5

Modelagem de um Sistema de RFID em SystemC/SystemC-AMS

5.1 Introdução

Para se modelar as versões descritas no capítulo anterior em SystemC-AMS, duas referências bibliográficas foram consultadas continuamente, sendo elas o manual do usuário do SystemC-AMS encontrado em [17], que apresenta diversas explicações a respeito da lógica de programação do SystemC-AMS, inclusive apresentando diversos exemplos de aplicações, e o manual de referência da linguagem, também abreviado para LRM - *Language Reference Manual* [18], que, por sua vez, possui documentadas todas as classes presentes nos MoC's do SystemC-AMS, apresentando a relação entre a entrada e saída destas classes. Logo, se o leitor desejar aprender mais a respeito destas linguagem, a leitura das referências apresentadas acima serão de inestimável ajuda.

5.2 Primeira Versão

5.2.1 Gerador de Estímulos

Antes de iniciar a modelagem dos blocos que compõem a *tag* da primeira versão, foi necessário modelar um bloco que gera sinais binários seriais para que, com estes sinais, fosse possível simular a requisição de um dado pelo leitor. Para tal, adotou-se que o sinal enviado pelo leitor seria escrito em um arquivo denominado data.txt utilizando a notação hexadecimal. Logo, o código para o gerador de estímulos deve ler esse arquivo em hexadecimal, fazer a conversão de hexadecimal para binário e enviar serialmente a saída binária obtida.

Para o processo de leitura do arquivo, foram utilizadas as funções de C++ básicas para manipulação de arquivos. Logo, primeiramente temos que declarar uma variável do tipo *fstream*, que nesse caso foi chamada de *data*, e em seguida abrir o arquivo data.txt nessa variável através do comando *data.open*. Por fim, basta realizar a leitura do arquivo linha a linha através do comando *getline*, sendo que essa operação é feita enquanto o arquivo estiver aberto e a leitura do mesmo for possível.

A cada linha do arquivo que foi lida através do *getline*, é preciso fazer a conversão de hexadecimal para binário. Essa conversão foi realizada caracter a caracter através de um *switch case*, sendo que cada resultado da conversão foi indexado a uma *string* denominada *binData*. Em seguida, a conversão de cada linha presente em *binData* foi indexada na *string bin*.

Independente do modelo de computação utilizado no SystemC-AMS, a primeira parte do código a ser executada sempre é o construtor. Era desejado que a operação de leitura do arquivo fosse realizada somente uma vez, ou seja, que o arquivo fosse lido completamente e a indexação da conversão hexadecimal em binário de todo o arquivo estaria na *string bin*. Desta forma, optou-se por descrever esta operação dentro do construtor para que a mesma fosse realizada antes de qualquer outra operação, conforme pode ser visualizado no trecho apresentado abaixo.

```
1  SCA_CTOR(bit_src) :out("out") { //Construtor.
2    data.open("data.txt");//Abre o arquivo com os dados que devem ser gerados.
3    if(data.is_open()) {    //Se o arquivo esta aberto faca.
4      while(data.good()) {  //Enquanto for possivel le o arquivo.
5        getline(data,sHex); //Le uma string e armazena em sHex.
6        binData = "";       //Converte a string para binario.
7        for (i=0;i<sHex.length();i++) {
8          switch (sHex[i]) {
9            case '0': binData.append ("0000"); break;
10           case '1': binData.append ("0001"); break;
11           case '2': binData.append ("0010"); break;
12           case '3': binData.append ("0011"); break;
13           case '4': binData.append ("0100"); break;
14           case '5': binData.append ("0101"); break;
15           case '6': binData.append ("0110"); break;
16           case '7': binData.append ("0111"); break;
17           case '8': binData.append ("1000"); break;
18           case '9': binData.append ("1001"); break;
19           case 'A': case 'a': binData.append ("1010"); break;
20           case 'B': case 'b': binData.append ("1011"); break;
21           case 'C': case 'c': binData.append ("1100"); break;
22           case 'D': case 'd': binData.append ("1101"); break;
23           case 'E': case 'e': binData.append ("1110"); break;
24           case 'F': case 'f': binData.append ("1111"); break;
25         }
26       }
27       bin.append (binData); //Armazena as palavras numa string.
28     }
29     data.close(); //Quando encontrar EOF o arquivo fecha.
30     i = 0;        //Inicializa i.
31   }
32 }
```

Em seguida é necessário converter o valor presente na *string bin* para a saída a ser enviada ao modulador BASK. Esse bloco do sistema é melhor descrito pelo MoC TDF - *Timed Data Flow* por gerar como saída um valor binário. No MoC TDF do SystemC-AMS existem três funções de retorno *void* que serão utilizadas com demasiada frequência neste trabalho, sendo elas *void initialize*, *void processing* e *void set_ attributes*.

A função *void set_ attributes* é utilizada para indicar as características dos sinais, entradas e saídas utilizadas no código, tais como taxa de amostragem, tempo entre uma amostra e outra, atraso entre a leitura ou escrita de um sinal, etc. A função *void initialize* é utilizada quando se

deseja inicializar o valor de uma variável que será usada em outras funções. Por fim, a função *void processing* funciona como um loop infinito executando todo seu processamento a cada amostra de tempo definida anteriormente em *void set_ attributes*. Existem ainda mais funções que podem ser utilizadas na linguagem SystemC-AMS na modelagem com o modelo de computação TDF, sendo as descritas anteriormente apenas as funções mais básicas dessa linguagem. Devido as outras funções não serem usadas neste trabalho, as mesmas não serão explicadas. Caso o leitor se interesse em conhecê-las, elas estão bem documentadas no manual do usuário [17].

O primeiro passo para converter essa *string bin* em um valor binário de saída é inicialmente determinar qual é o tempo que separa duas amostras de tempo ou a taxa de amostragem do sistema. Isso é necessário porque o MoC TDF é de tempo discreto e, portanto, precisa ter como informação o tempo entre duas amostras ou a taxa de amostragem do sistema.

Essa informação, conforme apresentado acima, é realizada na função *void set_ attributes*. O tempo escolhido entre as amostras da saída foi de 1 μs devido a frequência do gerador senoidal apresentado a seguir ser de 1 MHz, ou seja, será enviado para a saíŋda 1 bit para cada períŋodo completo da onda senoidal. Cabe ressaltar que, no caso de uma implementação hierárquica, caso algum dos módulos do sistema não possua a função *void set_ attributes*, a mesma irá utilizar as configurações oriundas do sinal de entrada na qual este bloco está conectado.

```
1 void set_attributes() {
2   out.set_timestep(1, sc_core::SC_US); //Tempo entre duas amostras.
3 }
```

Nessa classe, nenhuma variável precisou ser inicializada. Logo, a função *void initialize* não precisou ser utilizada. Já a função *void processing* é necessária em toda classe implementada através do modelo de computação TDF, sendo nessa função descrito o processamento executado pela classe.

Uma classe definida no modelo de computação TDF é reconhecida através do seu módulo denominado *SCA_ TDF_ MODULE* e pode possuir entradas e saídas do modelo de computação TDF ou DE - (*Discrete Event*), que é semelhante ao TDF com os sinais variando apenas em tempos pré estabelidos, porém com o sinal em tempo contínuo. Uma entrada ou saída denominada como DE costuma ser utilizada somente quando o bloco que está sendo modelado precisa interagir com o bloco de outro modelo de computação, tal como o LSF.

Desta forma, o gerador de estímulos foi implementado usando um módulo TDF, tendo uma saída do tipo booleana TDF, já que na primeira versão todos os blocos foram implementados neste modelo de computação, sendo que o código presente no construtor realiza a leitura e conversão do dado hexadecimal do arquivo para uma *string* e o código presente na função *void processing* executa a geração de uma saída booleana de acordo com o dado presente na *string*.

Na função *void processing*, temos que, enquanto uma variável inteira i for menor que o tamanho da *string*, deve-se verificar se a posição i da string é '0' ou '1'. Caso seja '0', será enviado para saída o sinal *false* e, caso seja '1', é enviado para saída o sinal *true*. Quando i for maior que o tamanho da string, será sempre enviado *false* para saída. O código que executa o gerador de estímulos apresentado acima pode ser visualizado abaixo.

```
void processing() {
  if (i<bin.length()) {
    if(bin[i]=='0')
      out.write(false);
    else
      out.write(true);
    i++;
  }
  else
    out.write(false);
}
```

5.2.2 Modulador BASK

Antes de modelar o modulador BASK, é preciso lembrar que ele utiliza dois sinais de portadoras e um multiplexador, além do gerador de estímulos citado acima, conforme pode ser visualizado na Fig. 4.2.

Então, antes desse bloco ser modelado, é necessário explicar como é modelado o multiplexador e o gerador senoidal. Ambos os blocos foram modelados nesta versão utilizando o MoC TDF, apesar do modelo que melhor os descreve ser o LSF, devido a esses dois blocos serem compostos com sinais analógicos.

A modelagem em LSF dos blocos analógicos deste sistema, apesar de não ter sido realizada nesta versão, foi realizada em uma segunda versão, o que inclusive renderá comentários posteriores comparando essas duas abordagens.

O gerador senoidal possui apenas uma saída do tipo *double* TDF, assim como o gerador de estímulos. Porém, nesse caso, a declaração do construtor é feita de uma forma diferente da declaração do gerador de estímulos, que informava apenas o nome da classe a ser construída.

O construtor do gerador senoidal permite que sejam passados para esta classe outros parâmetros, além da classe a ser construída. Nesse caso, os parâmetros que foram passados são amplitude e frequência do sinal senoidal, além do tempo entre as amostras deste sinal. Esses parâmetros possuem um valor *default* para o caso de não serem informados durante a instanciação. Os valores *default* são amplitude de 1V, frequência de 1 kHz e tempo entre as amostras de 10 ns.

O tempo decorrido entre duas amostras é definido para todo esse módulo através da função *set_ attributes*, utilizando *set_ timestep*. Então, a cada amostra citada acima, a função *void processing* é executada, onde primeiramente ela obtém o tempo atual através de *get_ time().to_ seconds()*, armazena esse valor em uma variável t e em seguida calcula a saída dada através da equação $A \cdot \sin(2 \cdot \pi \cdot f \cdot t)$.

A declaração do construtor e o trecho onde são setados os atributos e saídas deste módulo são apresentados abaixo. Este código é um dos exemplos encontrados no manual do usuário [17], que possui um exemplo completo de um modulador e demodulador BASK, tendo sido realizadas apenas algumas pequenas alterações para se adaptar melhor a aplicação requerida neste trabalho.

```
sca_tdf::sca_out<double> out;

sin_src(sc_core::sc_module_name nm, double ampl_ = 1.0, double freq_ = 1.0e3,
sca_core::sca_time Tm_ = sca_core::sca_time(10, sc_core::SC_NS))
: out("out"), ampl(ampl_), freq(freq_), Tm(Tm_) {}

void set_attributes() {
  set_timestep(Tm);
}

void processing() {
  t = get_time().to_seconds();
  out.write(ampl * std::sin(2.0 * M_PI * freq * t));
}
```

Em seguida, temos o multiplexador, que aqui será chamado de *mixer*. O *mixer* tem como entradas duas portadoras, que são duas entradas do tipo *double* TDF, o sinal do gerador de estímulos, do tipo booleano TDF, e possui apenas uma saída do tipo *double* TDF. Esta saída será igual a portadora de maior amplitude, no caso da entrada do gerador de estímulos ser '1', e igual a portadora de menor amplitude, no caso dessa entrada ser '0'.

No caso do *mixer*, o construtor foi descrito da forma padrão, ou seja, que contém apenas o nome da classe a ser construída. Também foi necessário definir a taxa de amostragem no mixer, sendo que esta foi definida para todas as entradas e saídas do tipo *double* com o valor de 100. Esse valor foi escolhido devido ao período da onda senoidal ser de 10 ns e do tempo entre cada amostra do sinal obtida do gerador de estímulos ser de 1 μs. Logo, para que esses tempos tenham um tamanho igual, de forma com que o SystemC-AMS consiga simular os mesmos sem problemas de sincronização, é necessário que os sinais oriundos do gerador senoidal sejam amostrados a cada 100 amostras, ou seja, 1 μs. Essa configuração foi realizada na função *void set_ attributes* através de *set_ rate*.

Logo, na função *void processing*, é necessário apenas indicar que, se a entrada do gerador de estímulos for '1', a saída será uma das ondas senoidais. Caso contrário, a saída será a outra portadora. O código que apresenta o funcionamento do *mixer* está apresentado abaixo.

```
SCA_CTOR(mixer)
: in_bin("in_bin"), in_wav1("in_wav1"), in_wav2("in_wav2"), out("out"), rate(100) {}

void set_attributes() {
  in_wav1.set_rate(rate);
  in_wav2.set_rate(rate);
  out.set_rate(rate);
}

void processing() {
    for(i=0;i<rate;i++)
    {
      if (in_bin.read())
        out.write(in_wav1.read(i),i);
      else
        out.write(in_wav2.read(i),i);
    }
}
```

Por fim, é necessário que os blocos anteriores sejam instanciados para formar a estrutura apresentada na Fig. 4.2. Essa instanciação é realizada utilizando um módulo do SystemC onde os sinais decladados como *private* interconectam as entradas e saídas de cada bloco. Ela é realizada dentro do construtor e, para que a mesma seja efetuada com sucesso, as classes definidas anteriormente precisam ser incluídas e cada bloco deve possuir seu respectivo nome. O código que apresenta a instanciação dos blocos mencionados acima está apresentado abaixo.

No código abaixo, também é possível observar como são setados os parâmetros de entrada do gerador senoidal quando o mesmo é declarado.

```
1  #include "sin_src.h"
2  #include "mixer.h"
3
4  SC_MODULE(bask_mod) {
5    sca_tdf::sca_in<bool>    in;
6    sca_tdf::sca_out<double> out;
7
8    sin_src sine1,sine2;
9    mixer   mix;
10
11   SC_CTOR(bask_mod)
12   : in("in"), out("out"),
13   sine1("sine1", 5.0, 1.0e7, sca_core::sca_time(10.0, sc_core::SC_NS)),
14   sine2("sine2", 1.8, 1.0e7, sca_core::sca_time(10.0, sc_core::SC_NS)), mix("mix") {
15     sine1.out(carrier1);
16     sine2.out(carrier2);
17     mix.in_wav1(carrier1);
18     mix.in_wav2(carrier2);
19     mix.in_bin(in);
20     mix.out(out);
21   }
22 private:
23   sca_tdf::sca_signal<double> carrier1, carrier2;
24 };
```

5.2.3 Demodulador BASK

O demodulador BASK, por sua vez, é composto de um retificador, um filtro e um amostrador, conforme apresentado anteriormente na Fig. 4.3. Logo, para modelar esse bloco, é necessário que os três blocos citados acima sejam modelados.

O retificador é uma classe que escreve na saída o valor absoluto da entrada, ou seja, quando a entrada for positiva, ela continuará a mesma, e quando for negativa, será multiplicada por -1. Assim, essa classe é extremamente simples, tendo somente uma entrada e saída do tipo *double* TDF, onde a saída recebe o sinal absoluto da entrada, conforme é apresentado no trecho abaixo.

```
1  void processing()
2  {
3    out.write( std::abs(in.read()) );
4  }
```

Em seguida, tem-se um filtro passa-baixa descrito pela equação 4.1. Este filtro, como mencionado anteriormente, é utilizado para filtrar ruídos de alta frequência e tornar mais suaves as variações de tensões de forma a garantir um resultado mais preciso na amostragem.

Para implementar este filtro no MoC TDF, é necessário utilizar a função de transferência de Laplace que existe no SystemC-AMS, sendo que existem dois modelos para esta função. Um é baseado em polos e zeros e o outro, no numerador e denominador. As funções *sca_ltf_nd* e *sca_ltf_zp* estão documentadas no LRM [18].

$$H(s) = \frac{1}{1 + \frac{s}{2 \cdot \pi \cdot f}} \tag{5.1}$$

Essa implementação foi realizada utilizando a função *sca_ltf_nd*, onde os numeradores e denominadores do filtro são inicializados na função *void initialize*, de acordo com os valores apresentados na equação 4.1. Em seguida, é necessário apenas escrever na saída a função *sca_ltf_nd* atuando sobre a entrada.

Cabe ressaltar que o módulo que descreve o filtro também foi construído de forma a permitir a inserção de parâmentros externos a função, que são o nome da classe a ser construída, a frequência de corte onde o sinal atenua 3 dB do filtro e a amplitude do sinal filtrado, definida como *default* com o valor 1.

```
1  sca_tdf::sca_in<double> in;
2  sca_tdf::sca_out<double> out;
3
4  ltf_nd_filter( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0)
5  : in("in"), out("out"), fc(fc_), h0(h0_) {}
6
7  void initialize()
8  {
9    num(0) = 1.0;
10   den(0) = 1.0;
11   den(1) = 1.0 / (2.0 * M_PI * fc);
12 }
13
14 void processing()
15 {
16   out.write( ltf_nd(num, den, in.read(), h0) );
17 }
```

Por fim, tem-se o amostrador, que também é uma classe onde o construtor permite o envio de parâmetros para o código, sendo os parâmetros de entrada o nome do amostrador e a tensão de *threshold*, que também será chamada de tensão de limiar.

Basicamente, o amostrador serve como um comparador onde, caso o sinal de entrada seja maior que a tensão de limiar, a saída será igual a 1, e caso contrário, a saída será igual a 0. Esse bloco também possui um comportamento bem simplificado e é mostrado no código a seguir.

```
sampler(sc_core::sc_module_name nm, double th): in("in"), out("out"),
rate(100), threshold(th) {}

void set_attributes()
{
  in.set_rate(rate);
  sample_pos = (unsigned long)std::ceil(2.0 * (double)rate/3.0);
}

void processing()
{
  if (in.read(sample_pos) > threshold)
    out.write(true);
  else
    out.write(false);
}
```

É valido reparar que, nesse caso, também foi necessário setar a taxa de amostragem do sinal de entrada para 1 μs, já que o desejado é que a função *void processing* seja executada somente após um período completo da onda senoidal. Outra observação interessante é que a amostragem não é realizada nos extremos do sinal, e sim em um ponto intermediário, podendo obter assim uma saída mais estável e, portanto, mais confiável. Isso é verificado no código quando se observa que o sinal de entrada que é comparado com a tensão de limiar é obtido no período *sample_pos*.

Agora, para obter o demodulador BASK apresentado na Fig. 4.3, basta instanciar todos os blocos citados acima da mesma forma que foi efetuada a instanciação do modulador BASK. Porém, como neste trabalho deseja-se verificar os sinais internos do demodulador BASK, essa instanciação não foi realizada, sendo que os blocos foram conectados entre si somente na função principal.

5.2.4 Conversor Serial Paralelo

O conversor serial paralelo tem como função converter o sinal serial obtido no demodulador BASK para um bloco de dados paralelo que será enviado para o decodificador. Nessa versão, a quantidade de bits da saída paralela foi definida como 12 bits.

Para implementar essa funcionalidade, foi utilizado um vetor auxiliar do tipo booleano e um inteiro i para ser utilizado como um contador. Esse vetor auxiliar foi inicializado com zeros e o i, com o tamanho do bloco paralelo que deverá ser obtido no final, ou seja, 12. Ambas as inicializações ocorreram na função *void initialize*. Após essa inicialização, a classe irá constantemente executar a função *void processing*, onde a cada amostra de tempo a variável auxiliar na posição i irá obter o valor da entrada e a posição i será incrementada. Quando i atingir o tamanho máximo do sinal paralelo, essa variável auxiliar será escrita na saída e a i será atribuído o valor 0. Estas funções encontram-se descritas abaixo.

Resumidamente, a cada período de amostragem, a entrada serial irá ser armazenada no índice i do vetor auxiliar, e quando o índice i do vetor atingir o tamanho máximo do dado paralelo de saída, este índice deverá ser zerado e todo o vetor auxiliar deverá ser escrito na saída.

```
void initialize()
{
  for(i=0;i<TAM;i++)
    aux[i]=false;
  i=TAM;
}

void processing()
{
  if(i==TAM) {
    for(i=0;i<TAM;i++)
      out[i].write(aux[i]);
    i=0;
  }
  aux[i]=in.read();
  i++;
}
```

5.2.5 Decodificador

O decoficador, por sua vez, obtem o dado paralelo oriundo do conversor serial paralelo e o transforma em um valor inteiro. Em seguida, é verificado se o valor que foi recebido equivale a um endereço de memória válido. Para isso, tem-se um *switch case* com alguns valores pré estabelecidos para acessar algum endereço de memória. Caso seja um comando válido, a resposta do decodificador será diferente de zero e esse valor inteiro irá corresponder ao enderaço de memória que deve ser acessado.

No decodificador, nenhuma inicialização ou atributo precisou ser setado. Logo, esse bloco foi descrito totalmente dentro da função *void processing*, conforme é apresentado a seguir no trecho abaixo.

```
void processing ()
{
  aux=0;
  for(i=0;i<TAM;i++)
    aux+=(in[i].read())*((int)pow(2.0,i));

  switch(aux) {
    case 1320: out.write(1);  break;
    case 3528: out.write(2);  break;
    case 1088: out.write(3);  break;
    case 4088: out.write(4);  break;
    default  : out.write(0);  break;
  }
}
```

5.2.6 Memória

A memória funcionará de forma semelhante ao gerador de estímulos, já que a mesma nessa versão foi simulada utilizando o acesso em um arquivo com valores hexadecimais, onde cada linha corresponde a um endereço de memória.

Porém, ao contrário do gerador de estímulos, nesse caso é necessário que a memória seja acessada sempre que houver uma requisição realizada pelo decodificador.

Assim, o processo de leitura do arquivo foi deslocado para a função *void processing*, sendo realizado cada vez que uma entrada é detectada. Outra alteração do código de leitura da memória é que a mesma não precisa memorizar todos os dados que foram lidos, como ocorria no gerador de estímulos. Na leitura da memória, apenas o endereço atual é relevante. Logo, a *string bin* existente no gerador de estímulos não é mais necessária.

Basicamente a memória recebe um endereço do tipo inteiro do decodificador e lê a quantidade de linhas necessárias para atingir esse endereço. Quando atinge o endereço alvo, a leitura do arquivo é interrompida e esse endereço alvo é convertido de hexadecimal para binário, sendo indexado na *string binData* e esta *string* posteriormente convertida para uma saída paralela, que será enviada ao conversor paralelo serial.

O trecho que realiza o acesso a memória descrito acima está apresentado abaixo.

```
1  void processing() {
2    j=in.read();
3    data.open("memory.txt"); //Abre o arquivo com os dados da memoria.
4
5    if(data.is_open()) {     //Se o arquivo esta aberto faca.
6      for(i=0;i<j+1;i++)
7        getline(data,sHex); //Le uma string e armazena em sHex.
8
9      binData = "";         //Converte a string para binario.
10     for (i=0;i<(int)sHex.length();i++) {
11       switch (sHex[i]) {
12         case '0': binData.append ("0000"); break;
13         case '1': binData.append ("0001"); break;
14         case '2': binData.append ("0010"); break;
15         case '3': binData.append ("0011"); break;
16         case '4': binData.append ("0100"); break;
17         case '5': binData.append ("0101"); break;
18         case '6': binData.append ("0110"); break;
19         case '7': binData.append ("0111"); break;
20         case '8': binData.append ("1000"); break;
21         case '9': binData.append ("1001"); break;
22         case 'a': case 'A': binData.append ("1010"); break;
23         case 'b': case 'B': binData.append ("1011"); break;
24         case 'c': case 'C': binData.append ("1100"); break;
25         case 'd': case 'D': binData.append ("1101"); break;
26         case 'e': case 'E': binData.append ("1110"); break;
27         case 'f': case 'F': binData.append ("1111"); break;
28       }
29     }
30
31     for (i=0;i<(int)binData.length();i++) {
32       if(binData[i]=='0')
33         out[i].write(false);
34       else
35         out[i].write(true);
36     }
37     data.close();
38   }
39 }
```

5.2.7 Conversor Paralelo-Serial

O conversor paralelo-serial é responsável por converter a saída paralela oriunda da memória para a saída serial que será modulada e enviada ao leitor. Essa conversão é até mais fácil de ser implementada que a conversão paralelo-serial, onde é necessário somente incrementar um contador i enquanto a saída lê o valor da entrada na posição i.

O contador i mencionado acima assim precisa ser inicializado com a quantidade de bits que a entrada paralela possui e sempre que o mesmo atingir o valor dessa quantidade máxima de bits, que neste caso é 12, precisa ser zerado. O trecho que executa a conversão paralelo-serial pode ser visualizado abaixo.

```
1  void initialize() {
2    i=TAM;
3  }
4
5  void processing() {
6    if(i==TAM)
7      i=0;
8    out.write(in[i].read());
9    i++;
10 }
```

5.2.8 Modulador BPSK

O modulador BPSK é implementado da mesma forma que o modulador BASK. Conforme podemos perceber nas Figs. 4.2 e 4.4, a estrutura dos dois moduladores são idênticas, alterando somente o valor da amplitude da portadora. Dessa forma, a instanciação do modulador BPSK foi muito semelhante a do modulador BASK, alterando somente a amplitude das portadoras utilizadas para 1.8V e -1.8V.

5.2.9 Demodulador BPSK

O demodulador BPSK é implementado através de um multiplicador, um integrador no período e um amostrador. Da mesma forma que o modulador BASK, esse demodulador é composto de três blocos simples de serem implementados.

O multiplicador, tal como o retificador, é implementado em apenas uma linha de código, onde a saída recebe o sinal das duas entradas multiplicadas, conforme é apresentado no trecho a seguir.

```
1  void processing() {
2    out.write( (in1.read()) * (in2.read()));
3  }
```

Cabe ressaltar que as entradas do multiplicador serão a saída do modulador BPSK enviada para o leitor e uma portadora de amplitude 1.8V.

Já o integrador no período necessita de uma variável auxiliar que irá se auto incrementar durante um período completo da onda senoidal. Ao atingir o final do período, essa variável auxiliar será resetada. A saída desse integrador receberá constantemente esta variável auxiliar.

```
1  void initialize() {
2    aux=0;
3    i=0;
4  }
5
6  void processing() {
7    if(i==100) {
8      aux=0;
9      i=0;
10   }
11
12   aux+=in.read();
13   out.write(aux);
14   i++;
15 }
```

Cabe ressaltar que o indicador utilizado para informar que um período foi encerrado é o contador i. O período é encerrado quando esse contador atinge o valor de 100 porque o período completo da onda senoidal é 1μs, e como o período entre duas amostras do sinal foi setado anteriormente em 10 ns, temos que apenas na centésima amostra do sinal a senoide irá iniciar um novo período.

Por fim, tem-se o amostrador, que é exatamente igual ao amostrador do demodulador BASK, sendo que a única coisa a ser alterada é o parâmetro de tensão de limiar que será enviado a este amostrador.

Da mesma forma que o demodulador BASK, desejava-se nesta versão que os sinais internos fossem apresentados ao leitor, de forma que a instanciação foi realizada apenas na função principal.

5.2.10 Instanciação da primeira versão

Após descrever o funcionamento de cada bloco individual da primeira versão, é necessário que estes blocos sejam conectados entre si para que o sistema possa ser simulado como um todo. Essa instanciação final, assim como a simulação do sistema, é obtida na função principal denominada *main.cpp*.

Nessa função principal, todas as classes descritas acima devem ser incluídas. Em seguida, dentro da função principal do systemC *sc_main()* é criado um arquivo de simulação denominado *trace.vcd* com uma resolução de 10ns, que é a menor resolução entre todos os sinais utilizados nesta versão. O próximo passo é realizar as interconexões entre todos os blocos descritos acima. Por fim, todos os sinais que serão simulados devem ser enviados ao arquivo de simulação pelo comando *sca_util::sca_trace()* sendo inicializada a simulação através do comando *sc_core::sc_start()*, no qual é setado o tempo total da simulação.

Como esta instanciação engloba muitos módulos o código que a descreve é um pouco grande, então por questão de legibilidade, o mesmo se encontra no anexo I, junto com as versões completas dos códigos apresentados acima.

5.3 Segunda Versão

Conforme mencionado anteriormente, a segunda versão apresenta uma melhoria significativa no protocolo de comunicação e na descrição dos blocos analógicos do sistema, utilizando para tal o MoC LSF. Também nessa versão é adicionado um ruído gaussiano branco e uma atenuação entre os sinais de saída dos moduladores e entradas dos demoduladores visando simular a interferência do meio externo no sistema. É válido lembrar que da primeira para a segunda versão o único bloco que não foi alterado foi o gerador de estímulos, de forma que o mesmo não será novamente descrito.

5.3.1 Modulador BASK

Na segunda versão, optou-se por realizar a modelagem dos blocos analógicos do sistema através do MoC LSF, que é mais apropriado para descrever sinais analógicos. A implementação com esse modelo de computação é obtida em sua maior parte pela instanciação das classes LSF já presentes na biblioteca do SystemC-AMS. Todas as classes LSF disponíveis em SystemC-AMS podem ser facilmente encontradas no manual de referência da linguagem [18] e são muito bem descritas no guia do usuário [17], que inclusive apresenta qual é a interação entre as entradas e saídas dessas classe, bem como o tipo de todas as entradas, saídas e parâmetros que devem ser passados para a classes.

Para implementar o modulador BASK, inicialmente foi preciso definir a classe no MoC LSF responsável por gerar o sinal das portadoras. A classe escolhida neste caso foi *sca_lsf::sca_source*, que apresenta apenas uma saída denominada y do tipo *sca_lsf::sca_out*, na qual é escrita uma onda senoidal.

Essa classe possui como parâmetros da onda senoidal o nome da classe, o valor no instante de tempo t=0, o *offset*, a amplitude, a frequência, a fase, o *delay*, a amplitude ac do sinal, fase ac do sinal e amplitude do ruído ac do sinal. Ou seja, conforme podemos observar, a implementação nesse modelo de computação permite que em um instante futuro diversas situações possam ser simuladas, bastando para isso alterar o parâmetro adequado da classe utilizada.

Como a princípio é desejado apenas alterar a amplitude e frequência do sinal senoidal, foram obtidas as portadoras que serão utilizadas nessa versão através do código apresentado abaixo. Cabe ressaltar que também é necessário configurar o tempo entre as amostras do sinal e que o trecho para geração da portadora apresentado abaixo é um dos exemplos de utilização do MoC LSF apresentados no guia do usuário [17].

```
1  //Define a saida e o MOC utilizado para descrever a fonte.
2  sca_lsf::sca_out    out;
3  sca_lsf::sca_source src;
4
5  //Construtor que seta a amplitude e frequencia de forma externa.
6  sine_src(sc_core::sc_module_name nm, double A, double f): out("out"),
7  src("src",0.0,0.0,A,f)  {
8    //Instanciacao do modulo que descreve a fonte.
9    src.set_timestep(10,sc_core::SC_NS);
10   src.y(out);
11 }
```

Em seguida, é necessário definir a classe que melhor irá descrever o modulador BASK. Nesse caso, optou-se por utilizar a classe *sca_lsf::sca_tdf_mux*, que nada mais é que um multiplexador que possui duas entradas do tipo *sca_lsf::sca_in* denominadas x1 e x2, uma entrada do tipo *sca_tdf::sca_in <bool>*, denominada ctrl, e uma saída do tipo *sca_lsf::sca_out*, denominada y. Se ctrl é igual a true, a saída y receberá x1. Caso contrário, a saída y receberá x2. O código que executa o modulador BASK está descrito abaixo.

```
1  SC_MODULE(mod) {
2    //Define as entradas e saidas do MOC utilizado para descrever o modulador.
3    sca_tdf::sca_in<bool>  in;
4    sca_lsf::sca_in        sine1,sine0;
5    sca_lsf::sca_out       out;
6
7    //Utiliza-se o MOC mux que e equivalente a um multiplexador de
8    //entradas e saida analogica e controle digital.
9    sca_lsf::sca_tdf_mux   mux;
10
11   SC_CTOR(mod):in("in"),sine1("sine1"),sine0("sine0"),out("out"), mux("mux")
12   {
13     //Instanciacao das entradas e saidas do multiplexador.
14     mux.x1(sine0);
15     mux.x2(sine1);
16     mux.ctrl(in);
17     mux.y(out);
18   }
19 };
```

É possível perceber que, com o MoC LSF, temos uma implementação mais direta do modulador BASK, bastando herdar as propriedades de uma classe já definida no SystemC-AMS, que possui uma implementação mais abrangente. Nesta versão é válido ressaltar que optou-se por realizar as instanciações apenas na função principal *main.cpp*. Logo diferente da primeira versão não temos uma classe para efetuar a instanciação das portadoras com o multiplexador com a mesma sendo efetuada em *main.cpp*.

5.3.2 Ruído e atenuação

O ruído gaussiano branco precisou ser modelado em uma função utilizando os recursos disponíves do C++, tais como raiz quadrada, a função de potenciação e a função que retorna um valor aleatório, respectivamente *sqrt, pow e rand*. Para descrição desse ruído, foi utilizado o método polar de Marsaglia.

A função utilizada para gerar esta distribuição aleatória com base na variância inserida de parâmetro foi obtida no guia do usuário [17], sendo que o código apresentado a seguir no MoC TDF apenas compatibiliza a saída desta distribuição randômica para que a mesma possa ser utilizada posteriormente na classe onde serão definidos os efeitos da interface com o ar. O trecho que apresenta uma saída TDF com um ruído gaussiano branco de variância Q está apresentado abaixo.

```
// A função gauss_rand() retorna uma distribuição gaussiana
// O número randômico de variância "variance" é centrado em 0 e utiliza o
// método polar de Marsaglia
double gauss_rand(double variance) {
  double rnd1, rnd2, Q, Q1, Q2;
  do {
    rnd1 = ((double)rand())/((double)RAND_MAX) ;
    rnd2 = ((double)rand())/((double)RAND_MAX) ;
    Q1 = 2.0*rnd1-1.0 ;
    Q2 = 2.0*rnd2-1.0 ;
    Q  = Q1*Q1+Q2*Q2 ;
  } while (Q > 1.0) ;
  return (sqrt(variance)*(sqrt(-2.0*log(Q)/Q)*Q1));
}

SCA_TDF_MODULE(gauss_noise) {
  //Ruido gaussiano branco.
  sca_tdf::sca_out<double> noise;

  gauss_noise(sc_core::sc_module_name nm, double variancia):
  noise("noise"), Q(variancia){}

  void processing() {
    //O ruido e a saida da funcao gauss_rand para variancia escolhida.
    noise.write(gauss_rand(Q));
  }
private:
  double Q;
};
```

Após implementada uma classe que gera uma saída do tipo *sca_tdf::sca_out <double>* contendo o ruído gaussiano branco com variância Q, podemos implementar a classe responsável por contaminar a saída do modulador com ruído e atenuar o sinal de saída dos moduladores.

Para isso, optou-se por herdar três diferentes tipos de classes do MoC LSF, *sca_lsf::sca_add, sca_lsf::sca_gain e sca_lsf::sca_tdf_source*, além de ser necessário utilizar a classe apresentada anteriormente que gera um ruído gaussiano branco.

A classe *sca_lsf::sca_tdf_source* possui uma entrada do tipo *sca_tdf::sca_in<double>* denominada inp e uma saída do tipo *sca_lsf::sca_out* denominada y. Essa classe é utilizada quando se deseja converter um sinal do MoC TDF para o MoC LSF, nesse caso sendo utilizada para converter o ruído gaussiano branco que possui uma saída TDF para o MoC LSF. Desta forma esta classe será utilizada apenas para adequar a saída da classe apresentada acima para o modelo LSF.

A classe *sca_lsf::sca_gain* possui uma entrada LSF denominada x, uma saída LSF denominada y e um parâmetro que corresponde ao ganho. Basicamente, essa classe multiplica a entrada LSF pelo ganho setado no parâmetro que, caso seja menor do que 1, como deverá ser na modelagem deste sistema, causará um efeito de atenuação no sinal de entrada. Essa classe é utilizada para atenuar a saída do modulador.

Por fim, temos a classe *sca_lsf::sca_add*, que possui duas entradas LSF x1 e x2 e uma saída LSF y, que recebe a soma dos sinais x1 e x2. Essa classe é utilizada para somar o sinal atenuado ao ruído LSF, obtendo assim um sinal completamente contaminado do modulador.

O trecho que executa a interface com o ar é apresentado a seguir.

```
SC_MODULE(air) {
  //Entrada e saida da interface.
  sca_lsf::sca_in  in;
  sca_lsf::sca_out out;

  //Mocs utilizados na simulacao de ruido e atenuacao.
  sca_lsf::sca_add        add;
  sca_lsf::sca_gain       gain;
  sca_lsf::sca_tdf_source sink;
  gauss_noise             sig_wn;

  air(sc_core::sc_module_name nm,double atenuacao, double variancia)
  : in("in"), out("out"), add("add",1.0,1.0), gain("gain",atenuacao),
  sink("sink",1.0), sig_wn("sig_wn",variancia) {
  //Produz a entrada atenuada.
  gain.x(in);
  gain.y(sig_at);

  //Ruido gaussiano branco em tdf.
  sig_wn.noise(noise_tdf);

  //Converte o ruido em lsf.
  sink.inp(noise_tdf);
  sink.y(noise_lsf);

  //Contamina o sinal atenuado com ruido gaussiano branco.
  add.x1(sig_at);
  add.x2(noise_lsf);
  add.y(out);
  }

private:
  sca_lsf::sca_signal sig_at, noise_lsf;
  sca_tdf::sca_signal<double> noise_tdf;
};
```

5.3.3 Demodulador BASK

O demodulador BASK é formado por um retificador, um filtro e um amostrador. Na modelagem desse bloco em LSF, foi encontrada uma grande dificuldade devido a nenhuma classe em LSF implementar de forma direta um retificador.

Desta forma, para implementar um retificador utilizando o MoC LSF, optou-se por implementar a estrutura apresentada na Fig. 5.1, sendo necessário instanciar quatro classes do MoC LSF, além de um amostrador.

Como a utilização do amostrador foi requerida no retificador, é válido mencionar que o mesmo apresenta um funcionamento bem simplificado e inclusive semelhante ao apresentado na primeira versão. A diferença está no fato da entrada do amostrador ser do MoC DE e de termos como parâmetros a tensão de limiar e o tempo que irá decorrer entre duas amostras, utilizado neste caso devido a saída ser do tipo TDF. O trecho que apresenta o comportamento do amostrador é apresentado abaixo.

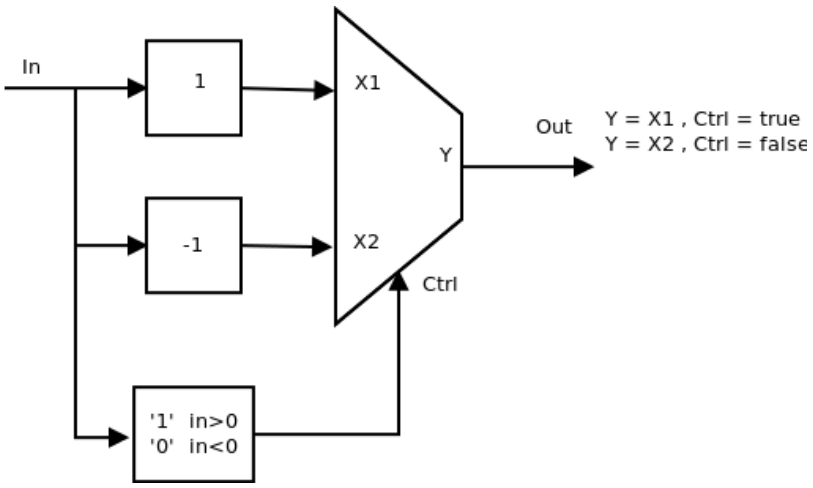

Figura 5.1: Implementação do decodificador no MoC LSF desenhado com o auxílio do *software* Dia [9].

```
1  //Declara as entradas e saidas do amostrador.
2  sca_tdf::sca_de::sca_in<double>  in;
3  sca_tdf::sca_out<bool>  out;
4
5  sampler(sc_core::sc_module_name nm, double th, sca_core::sca_time Tm)
6  : in("in"), out("out"), threshold(th), time(Tm) {}
7
8  void initialize() {
9    aux=false;
10 }
11
12 void set_attributes() {
13   out.set_timestep(time);
14 }
15
16 void processing() {
17   aux = (in.read()>threshold)? true : false;
18   out.write(aux);
19 }
```

Dessa forma, podemos instanciá-lo no retificador, sendo necessário apenas indicar qual é a tensão de limiar e o tempo entre as amostras deste retificador. A implementação do retificador seguindo o apresentado na Fig. 5.1 é mostrada a seguir.

```
1  SC_MODULE(retifier) {
2    //Entradas e saidas do retificador.
3    sca_lsf::sca_in   in;
4    sca_lsf::sca_out  out;
5
6    //MOC's utilizados para retificação.
7    sca_lsf::sca_gain    mul1, mul2;
8    sca_lsf::sca_de_sink sink;
9    sca_lsf::sca_tdf_mux mux;
10   sampler              sp;
11
12   SC_CTOR(retifier): in("in"), out("out"), mul1("mul1",-1.0),
13   mul2("mul2",1.0), sink("sink", 1.0), mux("mux"),
14   sp("sp",0.0,sca_core::sca_time(10,sc_core::SC_NS)) {
```

```
15      //Multiplica o sinal por -1.
16      mul1.x(in);
17      mul1.y(x1);
18
19      //Multiplica o sinal por 1.
20      mul2.x(in);
21      mul2.y(x2);
22
23      //Converte entrada para discrete event.
24      sink.x(in);
25      sink.outp(in_de);
26
27      //Se a entrada for maior que zero ctrl=true, caso contrario ctrl=false.
28      sp.in(in_de);
29      sp.out(ctrl);
30
31      //Retifica o sinal.
32      mux.x1(x1);
33      mux.x2(x2);
34      mux.ctrl(ctrl);
35      mux.y(out);
36    }
37  private:
38    //Sinais internos.
39    sca_lsf::sca_signal        x1, x2;
40    sc_core::sc_signal<double> in_de;
41    sca_tdf::sca_signal<bool>  ctrl;
42  };
```

Por fim, tem-se o filtro passa-baixa implementado através da classe *sca_lsf::sca_ltf_nd*. Novamente, o mesmo é implementado de forma semelhante ao da primeira versão, com a saída sendo herdada de uma classe pré definida no SystemC-AMS. A diferença principal no caso dessa implementação é que a forma com a qual o numerador e denominador são definidos apresenta uma sintaxe bastante diferente da primeira versão, além de que, nessa implementação, é necessário que a saída do filtro seja convertida para o MoC DE, de forma a se tornar compatível com a entrada do amostrador já apresentado acima que também é utilizado no demodulador. O trecho principal do filtro passa baixa está apresentado abaixo.

```
1  sca_lsf::sca_in         in; //Entrada e saiyda do filtro.
2  sc_core::sc_out<double> out;
3
4  sca_lsf::sca_ltf_nd  lp;    //MOC's utilizados para o filtro.
5  sca_lsf::sca_de_sink sink;
6
7  filter(sc_core::sc_module_name nm, double fc, double h0): in("in"), out("out"),
8  lp("lp",sca_util::sca_create_vector(1.0),
9  sca_util::sca_create_vector(1.0,1.0/(2.0*M_PI*fc)),sc_core::SC_ZERO_TIME,h0),
10 sink("sink",1.0) {
11   //Filtro passa baixo.
12   lp.x(in);
13   lp.y(lp_out);
14
15   //Converte a saida do filtro para DE.
16   sink.x(lp_out);
17   sink.outp(out);
18 }
```

5.3.4 Decodificador

O decodificador primeiramente é responsável por verificar se o comando que chega na *tag* é válido. Nesse caso, todos os comandos enviados pelo leitor iniciam com a sequência de bits 10. Logo, o comando obtido pela *tag* é considerado válido caso o mesmo seja iniciado com 10. Caso o comando seja válido, ele permite que todo ele seja decodificado. Caso contrário, o decodificador não executa o processo de decodificação.

A decodificação do comando consiste em ler completamente um comando serial recebido pelo modulador e convertê-lo em pacotes paralelos que serão enviados para a unidade de controle. Haverá quatro pacotes paralelos na saída do decodificador, que são o comando recebido pela *tag*, o endereço que será manipulado, o número aleatório da *tag* que deve responder ao comando enviado e, por fim, o dado que será escrito caso o comando recebido anteriormente solicite escrita na memória.

Outra funcionalidade do decodificador é ativar o gerador de números aleatórios quando receber um comando de inicialização, desativando esse gerador assim que for chegado ao decodificador o primeiro comando em baixo. O código que executa a função do decodificador é extenso é por questão de legibilidade será apresentado apenas no anexo II.

5.3.5 Gerador aleatório

O gerador aleatório possui uma implementação bem simplificada. Quando sua *flag* de ativação está em nível lógico alto, o mesmo produz uma saída randômica igual a zero ou um para cada bit que compõe o número aleatório produzido pela *tag*. O trecho do gerador aleatório é apresentado a seguir.

```
1  SCA_TDF_MODULE(rdm) {
2    sca_tdf::sca_in <bool> flag;
3    sca_tdf::sca_out<bool> rng[RNG];
4
5    SCA_CTOR(rdm) : flag("flag") {}
6
7    void processing() {
8        if(flag==true) {
9      for(i=0;i<RNG;i++)
10         rng[i].write((bool)(rand()%2));
11       }
12   }
13
14 private:
15     int i;
16 };
```

5.3.6 Unidade de Controle

A unidade de controle por sua vez, é responsável por implementar uma máquina de estados que possui a transição de seus estados apresentado na Fig. 4.10.

Da mesma forma que o decodificador, o código que executa esta máquina de estados é extenso, visto que o mesmo deve tratar as pecularieda des de cada estado para que a comunicação *tag* e leitor ocorra de forma eficiente. Desta forma, assim como no decodificador o código da unidade de controle estará presente apenas no anexo II.

Esta máquina de estados inicialmente se encontra no estado *stand by* na qual permanece até que receba um comando de *start*. Após receber este comando o estado da máquina de estados muda para *start*, que é responsável por enviar o comando i2c para armazenar o número aleatório gerado no estado anterior no segundo endereço de memória. Depois a máquina de estados vai para o estado *init* o qual tem como saída o comando i2c para leitura do identificador da *tag* na memória. Após estas etapas, a saída esperada para o leitor será receber o valor do número aleatório gerado pela *tag* e seu respectivo identificador, estas saídas apesar de não serem implementadas nesta classe são importantes de mencionar, pois só será permitida a transição para os estados de leitura, escrita e *kill*, caso o número aleatório enviado pelo leitor coincida com o gerado pela *tag* de forma a permitir que o leitor selecione de forma mais segura qual *tag* deve responder a solicitação.

Os estados de leitura e escrita enviam o código i2c informando em qual endereço de memória deve ser realizada a manipulação na memória, o tipo da operação, leitura ou escrita, e no caso de escrita, o dado que será escrito. O estado que mata a *tag* apenas a envia a máquina de estados novamente para o estado de *stand by*.

5.3.7 Unidade de Controle da Saída

Esta classe é responsável por organizar e enviar a saída esperada da *tag* para o modulador e posteriormente para o leitor e também realizar um pré processamento dos sinal i2c enviado pela unidade de controle de forma a adequá-la para o utilizado na interface de memória.

Resumidamente o que ocorre é uma conversão paralelo serial do comando i2c enviado pela unidade de controle visando enviá-lo a interface de memória, onde tem-se também uma espera para que a interface informe se o dado foi escrito ou qual é o dado que foi lido. Por fim, é organizado os sinais oriundos do decodificador e da memória para eviar serialmente ao modulador a resposta que a *tag* enviará ao leitor. O trecho com as operações principais desta classe encontra-se apresentado abaixo.

```
1  void processing()
2  {
3    if(flag==true)
4    {
5      //Ativa a interface com a memoria.
6      ie_ativo.write(true);
7
8      //Caso i seja igual ao tamanho do comando I2C o mesmo e inicializado.
9      if(i==I2CT)
10       i=0;
11
12     //Reinicia quando necessario o contator do dado de memoria.
13     if(k<0)
14       k=3;
15
```

```
16      //Quando o dado oriundo da memoria estiver pronto, armazena numa variavel auxiliar.
17      if(i==I2CT-1) {
18        r=0;
19        for(j=DAT-1;j>=0;j--) {
20          aux[r]=data[j].read();
21          r++;
22      }
23
24      //Serializa comando i2c oriundo da unidade de controle.
25      if(i<I2C)
26        i2c_ie.write(uc[i]);
27      else
28        i2c_ie.write(false);
29
30      //Organiza sinal obtido da interface de memoria para enviar ao modulador BPSK.
31      if      (i<WORD)
32        out.write(aux[i]);
33      else if(i>=WORD && i<2*WORD)
34        out.write(cmd[k].read());
35      else if(i>=2*WORD && i<3*WORD)
36        out.write(adr[k].read());
37      else
38        out.write(rng[k].read());
39
40      k--;
41      i++;
42    }
43    else {
44      //Se a flag de ativação estiver em baixo, todas as saidas ficam em nivel baixo.
45      ie_ativo.write(false);
46      out.write(false);
47      i2c_ie.write(false);
48    }
49  }
```

5.3.8 Interface com a memória

A interface de memória por sua vez é responsável por identificar se o comando requerido é de leitura ou escrita e implementar o código i2c para se comunicar diretamente com a memória.

Basicamente a cada pulso de clock o código i2c serial oriundo da unidade de controle de saída é armazenado em uma variável auxiliar, o que provou-se necessário devido ao protocolo padrão de comunicação i2c exigir um sinal de ACK para o caso da operação ter sido realizada com sucesso e de NACK caso haja alguma falha na mesma e devido estes sinais de controle não estarem apresentados no código oriundo da unidade de controle de saıda.

Assim que o bloco referente ao comando chega a interface, a mesma seleciona se é uma operação de leitura ou escrita e então segue o protocolo padrão i2c de comunicação já mencionado nos capítulos anteriores. O código que apresenta a execução das operações descritas acima também é extenso e por isso estará presente somente no anexo II.

5.3.9 Memória

A memória por sua vez, assim como a interface de memória armazena o comando i2c recebido pela interface em uma variável auxiliar e o decodifica para saber se é uma operação de leitura e escrita.

Caso seja uma operação de escrita a memória pega o dado enviado pela interface de controle e o armazena no endereço de memória solicitado na interface enviando de resposta para interface uma confirmação que o dado foi escrito, caso a operação seja de leitura, a memória acessa o endereço solicitado na interface e envia o conteúdo deste endereço para interface de memória.

Da mesma forma que o código da interface de memória, o código que executa as operações acima é extenso e por isto será encontrado apenas no anexo II.

5.3.10 Modulador BPSK

O modulador BPSK teve sua modelagem de forma idêntica ao do modulador BASK, sendo inclusive reaproveitada a classe do modulador BASK com diferentes portadoras instanciadas na entrada.

5.3.11 Demodulador BPSK

O demodulador BPSK é composto de um multiplicador, um integrador no período e um amostrador. Cabe ressaltar que o amostrador utilizado para este demodulador é o mesmo utilizado no demodulador BASK, logo este bloco não será explicado novamente.

O multiplicador utiliza as classes *sca_lsf::sca_de_gain e sca_lsf::sca_de_sink*. Basicamente, a classe *sca_lsf::sca_de_gain* permite multiplicar um sinal de entrada com um ganho variável, desde que este ganho esteja no MoC DE. Por isso, uma das entradas que será multiplicada é convertida inicialmente para o MoC DE através da classe *sca_lsf::sca_de_sink* para que com isso seja possível obter a multiplicação destes dois sinais. O trecho que apresenta o citado acima encontra-se abaixo.

```
1  sca_lsf::sca_in   in1,in2; //Entradas e saiydas do multiplicador.
2  sca_lsf::sca_out  out;
3
4  sca_lsf::sca_de_sink sink; //Classes utilizadas para descrever o multiplicador.
5  sca_lsf::sca_de_gain mul;
6
7  //Construtor que seta as entradas e saÍydas do multiplicador.
8  SCA_CTOR(mult): in1("in1"), in2("in2"), out("out"), sink("sink",1.0), mul("mul",1.0) {
9    //Instanciacao do conversor lsf to de.
10   sink.x(in2);
11   sink.outp(gain);
12
13   //Instanciacao do modulo que descreve o multiplicador.
14   mul.x(in1);
15   mul.inp(gain);
16   mul.y(out);
17 }
```

Em seguida, tem-se o integrador no período. Entre as classes definidas no MoC LSF do SystemC-AMS, temos um integrador, porém o mesmo funciona de forma contínua, sem que seja possível inicializá-lo com 0 depois de um certo período ou mesmo delimitar os seus limites de integração. Devido ao mencionado acima, este foi outro bloco no qual houve uma dificuldade em se implementar utilizando o MoC LSF.

Para modelar o integrador foi necessário obter um sinal de entrada atrasado em um período completo, o qual foi obtido através da classe *sca_lsf::sca_delay*. Em seguida, a entrada atrasada e a entrada normal foram integradas por intermédio de duas classes do tipo *sca_lsf::sca_integ*. Com isso, é possível através da subtração da integração da classe normal pela integração da classe atrasada obter a integração por período do sinal.

Adicionalmente, essa integração foi convertida para o MoC DE, de forma a tornar a saída desse bloco compatível com a saída do amostrador. O trecho que apresenta a integração por período se encontra descrito abaixo.

```
1  sca_lsf::sca_in          in; //Entrada e saida do integrador.
2  sc_core::sc_out<double>  out;
3
4  sca_lsf::sca_integ   sum1;   //Declara os MOCs usados na integracao por periodo.
5  sca_lsf::sca_integ   sum2;
6  sca_lsf::sca_delay   delay;
7  sca_lsf::sca_sub     sub;
8  sca_lsf::sca_de_sink sink;
9
10 SCA_CTOR(integ): in("in"), out("out"), sum1("sum1",1.0,1.0), sum2("sum2",1.0,1.0),
11 delay("delay",sca_core::sca_time(1,SC_US),1.0,0.0), sub("sub",1.0,1.0), sink("sink",1.0) {
12   //Atrasa em um periodo a entrada.
13   delay.x(in);
14   delay.y(aux1);
15
16   //Integra a entrada.
17   sum1.x(in);
18   sum1.y(aux2);
19
20   //Integra a entrada atrasada.
21   sum2.x(aux1);
22   sum2.y(aux3);
23
24   //Subtrai a integral da entrada pela integral da entrada atrasada.
25   sub.x1(aux2);
26   sub.x2(aux3);
27   sub.y(aux4);
28
29   //Converte a saiyda de TDF para DE para amostrar.
30   sink.x(aux4);
31   sink.outp(out);
32 }
```

5.3.12 Instanciação da Segunda Versão

A função principal da segunda versão foi feita de forma equivalente a instanciação da primeira versão. Sendo assim os pormenores desta implementação não serão novamente explicados e o código que descreve esta instanciação se encontra presente no anexo II.

Capítulo 6

Resultados

Os códigos descritos no capítulo 4 foram compilados com auxílio de um *Makefile*. Desta forma, foi necessário apenas utilizar o comando *make* no terminal para gerar os arquivos objetos e executáveis. O executável gerado possui o nome run.x e foi executado através do comando ./run.x digitado no terminal. Os resultados obtidos destas compilações são apresentados nos tópicos a seguir. Para reproduzir estes resultados com os códigos apresentados nos anexos I e II, é necessário ter devidamente instalado no computador o SystemC-AMS-1.0 e SystemC-2.3.0, e rodar o *Makefile* apresentado abaixo.

```
1  CC      = g++
2  OPT     = -O3
3  DEBUG   = -g
4  OTHER   = -Wall
5
6  #no debug
7  CFLAGS = $(OPT) $(OTHER) -Wno-deprecated
8
9  #uncomment this line for debug build
10 # CFLAGS = $(DEBUG) $(OTHER)
11
12 MODULE = run
13 # sources files separated by spaces
14 SRCS = main.cpp
15
16 OBJS = $(SRCS:.cpp=.o)
17
18 # Set these variables according to your configuration
19 SYSTEMC = /usr/local/systemc-2.3.0
20 SYSTEMCAMS = /usr/local/systemc-ams-1.0
21
22 # Target architecture eg. linux or linux64
23 TARGET_ARCH = linux64
24
25 INCDIR = -I. -I$(SYSTEMC)/include -I$(SYSTEMCAMS)/include
26 LIBDIR = -L$(SYSTEMC)/lib-$(TARGET_ARCH) -L$(SYSTEMCAMS)/lib-$(TARGET_ARCH)
27
28 LIBS    = -lpthread -lsystemc -lsystemc-ams  $(EXTRA_LIBS)
29
30 EXE     = $(MODULE).x
31
```

```
32 .SUFFIXES: .cpp .o .x
33
34 $(EXE): $(OBJS) $(SYSTEMC)/lib-$(TARGET_ARCH)/libsystemc.a
35   $(SYSTEMCAMS)/lib-$(TARGET_ARCH)/libsystemc-ams.a
36   $(CC) $(CFLAGS) $(LIBDIR) -o $@ $(OBJS) $(LIBS)
37
38 all: $(EXE)
39
40 .cpp.o:
41   $(CC) $(CFLAGS) $(INCDIR) -c $<
42
43 clean::
44   rm -f $(OBJS) *~ $(EXE) *.vcd
```

6.1 Simulação da Primeira Versão

A simulação em SystemC/SystemC-AMS obtida ao executar o arquivo run.x gerou um arquivo com informações de todos os sinais internos da *tag* para ser plotado. Esse arquivo denominado trace.vcd, foi visualizado no *software* GTKWave [19]. As formas de onda estão apresentadas nas Figs. 6.1, 6.3, 6.4 e 6.5 apresentadas a seguir.

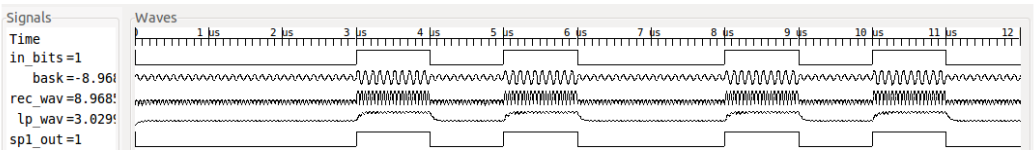

Figura 6.1: Simulação da primeira versão BASK em SystemC-AMS

A figura 6.1 apresenta a simulação do transceptor BASK. É possível perceber que tanto a modulação na amplitude como a demodulação estão sendo efetuadas com sucesso de forma que o sinal recuperado na saída, sp1_out, é idêntico ao sinal que está entrando no modulador.

Nesta versão nenhum atraso foi observado na demodulação. O que é motivado principalmente a forma diferenciada que foi utilizada na amostragem do sinal, ao amostrar em um ponto intermédiario do sinal longe dos períodos de transição.

Por fim, tem-se que nesta demodulação o sinal obtido pelo filtro, mesmo no nível baixo de entrada na modulação, é bem diferente de zero, estando na faixa de 1V conforme pode ser visualizado na Fig. 6.2 a seguir. Desta forma a *tag* só estará com a alimentação igual a zero quando estiver sem nenhuma comunicação com o leitor, mantendo-se sempre com pelo menos 1V de tensão DC.

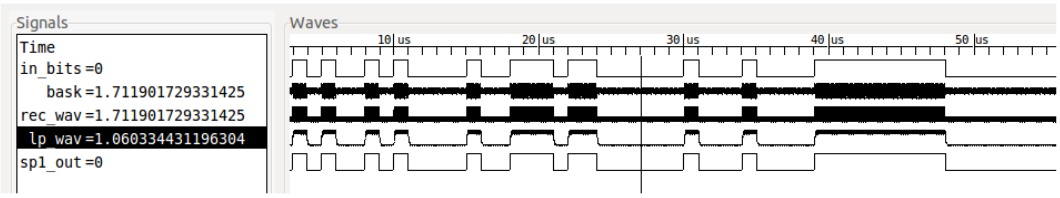

Figura 6.2: Amplitude da tensão no filtro em nível baixo em SystemC-AMS

A Fig. 6.3 apresenta qual é a saída do conversor serial paralelo. É possível observar que existe um atraso de 12 pulsos de clock para que o conversão seja realizada, o que era o esperado, considerando que a palavra enviada pelo modulador BASK possui 12 bits e a cada pulso de clock apenas a informação de 1 bit chega ao conversor. Adicionalmente também é possível verificar que o primeiro bit recebido pelo conversor está sendo setado como o bit menos significativo do sinal.

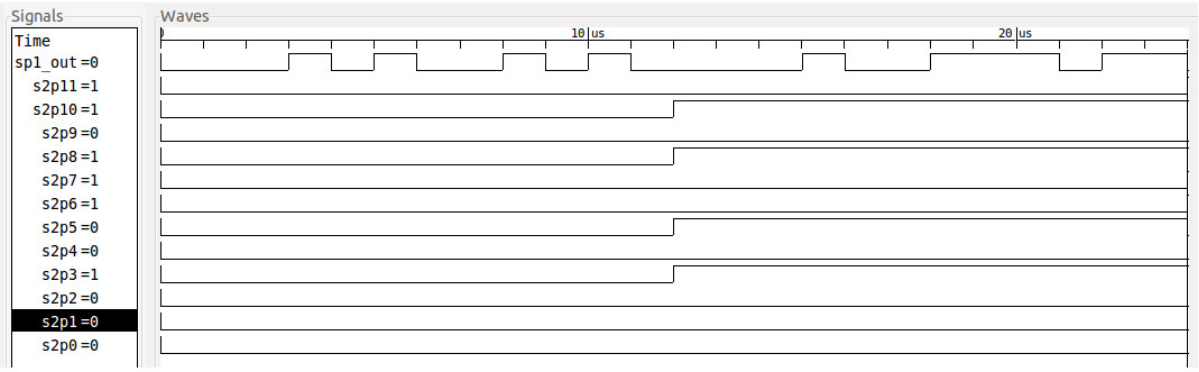

Figura 6.3: Simulação da conversão serial paralelo em SystemC-AMS

Já a Fig. 6.4 apresenta a saída do decodificador, da memória e da conversão paralelo serial. É possível observar que o dado paralelo obtido na saída do conversor serial paralelo é um dado válido de acesso do endereço 1. A memória possui no endereço 0 o valor 0 e nos outros endereços valores arbitrários. É possível perceber que quando o endereço enviado a memória foi 1, a mesma conseguiu acessar o arquivo e enviar o dado existente neste arquivo para uma saída paralela. Desta forma a mesma está funcionando corretamente. O conversor serial paralelo por sua vez também foi executado com sucesso, é novamente foi adotada a convenção de que o bit menos significativo é enviado primeiro.

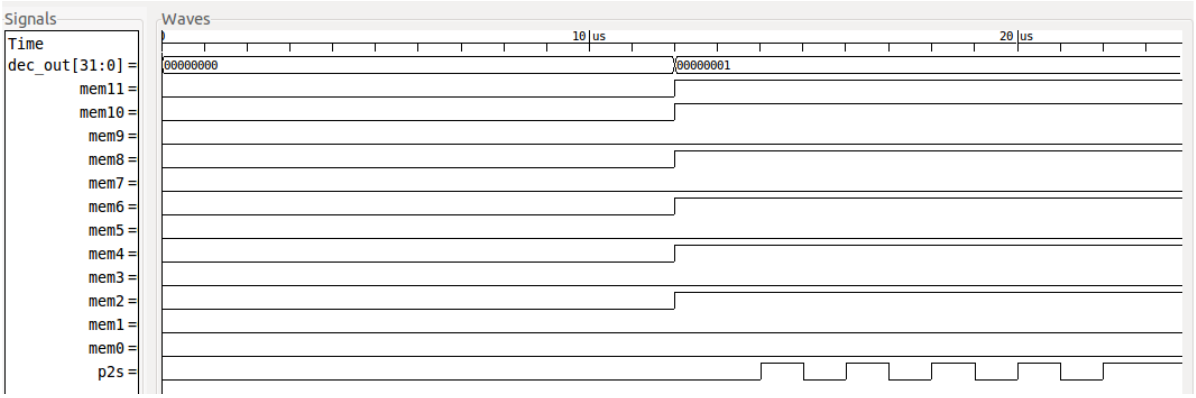

Figura 6.4: Simulação da decodificação, leitura na memória e conversão paralelo serial em SystemC-AMS

Por fim, tem-se o transceptor BPSK que conforme apresentado na Fig. 6.5 também foi capaz de converter corretamente o sinal sem adicionar atraso no sistema, desta forma a arquitetura apresentada acima foi bastante funcional.

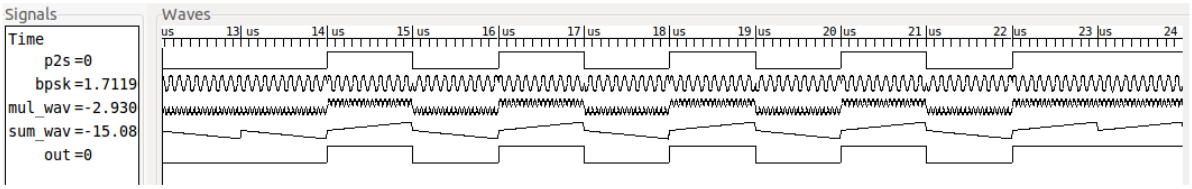

Figura 6.5: Simulação do transceptor BPSK em SystemC-AMS

Apesar de funcional e possuir somente 12 bits de atraso entre o envio de um comando e o recebimento da resposta, esta arquitetura possui uma comunicação bastante ineficiente. Primeiramente é notado que não tem como selecionar qual será a *tag* que irá responder ao comando podendo causar diversas colisões de sinais caso diferentes *tags* resolvam responder o sinal. Uma alternativa para este problema é em cada *tag* adicionar valores arbitrários diferentes para se ter acesso a algum endereço, porém novamente esta não é uma alternativa muito interessante, visto que evita uma padronização na construção dos comandos, o que pode ser bastante complicado de se implementar na prática.

Outro problema desta arquitetura é que a memória deve ser interligada ao sistema, não existe um protocolo de acesso a memória que permita adicionar mais blocos de memória caso seja necessário, ou inclusive outros periféricos para tornar o sistema mais adaptável, e por fim as operações requeridas são sempre operações de leitura, de forma com que o leitor não tenha tanta autonomia para modificar ou adicionar alguma informação para *tag*. Basicamente esta arquitetura seria restrita a aplicações bem simples de identificação e ainda assim, seria complicado utilizar esta arquitetura de forma com que não haja colisões no envio de sinais.

6.2 Simulação da Segunda Versão

O transceptor BASK nesta segunda versão foi obtido através do modelo de computação LSF diferentemente da primeira versão. É possível observar pela Fig. 6.6 a seguir, que o transceptor BASK modelado em SystemC-AMS apresentou o comportamento esperado, sendo capaz de recuperar o sinal modulado mesmo estando contaminado com bastante ruído.

Porém a contaminação com o ruído, bask_wn, e o fato da amostragem do sinal estar sendo realizada no término de cada período da onda senoidal, faz com que a saída do demodulador BASK apresente um pequeno atraso de 1 pulso de clock.

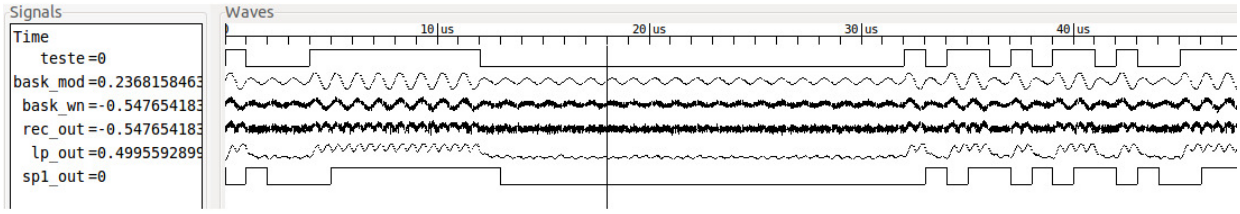

Figura 6.6: Simulação da segunda versão transceptor BASK em SystemC-AMS

Após passar pelo demodulador BASK acima, o sinal deve ser decodificado, onde esta decodicação envia paralelamente para a unidade de controle os sinais que compõem o comando, endereço, número aleatório e dado enviados através do modulador BASK. Conforme é possível verificar na Fig. 6.7 temos que os sinais foram enviados corretamente já que este modulo atua de forma semelhante a um conversor serial paralelo, onde o primeiro bit que chega corresponde ao bit mais significativo e o ultimo bit de cada informação corresponde ao bit menos significativo.

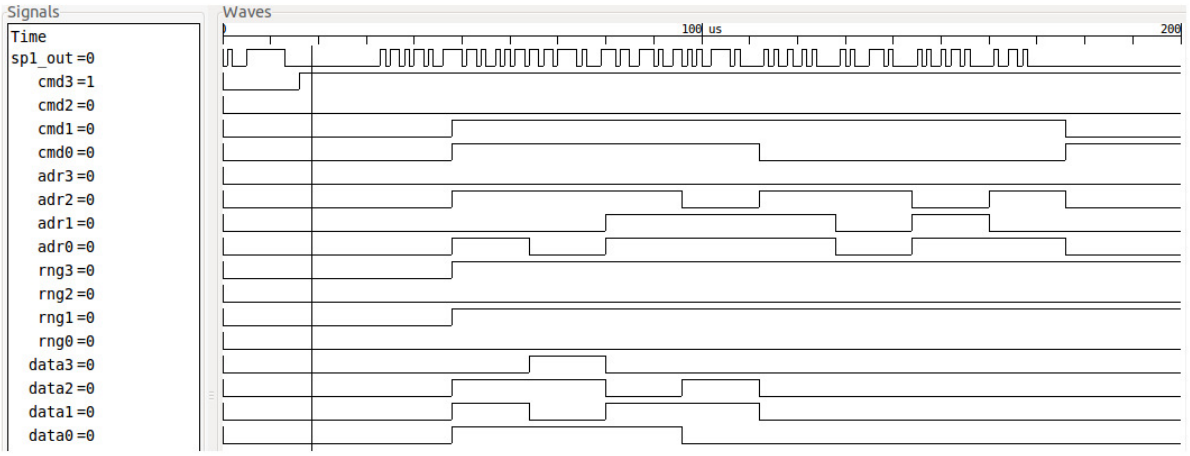

Figura 6.7: Decodificação do sinal demodulado na segunda versão em SystemC-AMS

O decoficador assim que detecta o sinal do comando, permanece sempre com os dois bits mais significativos em "10"o que era esperado já que este é o indicador de comando válido para este caso. É válido observar que o número aleatório enviado pelo leitor é igual a "1010", e para a *tag* poder interagir com o leitor neste caso será necessário que o número aleatório da *tag* seja igual ao número aleatório enviado pelo leitor. Como a comunicação entre *tag* e leitor ocorreu com sucesso, tem-se que que estes dois valores terão que ser necessariamente iguais. Conforme é possível de observar ao verificar a Fig. 6.8.

A Fig.6.8 também permite verificar que o gerador randômico é ativado somente durante o nível alto que se segue o comando de ativação, alterando de forma aleatória o seu valor até possuir um indentificador próprio da *tag*.

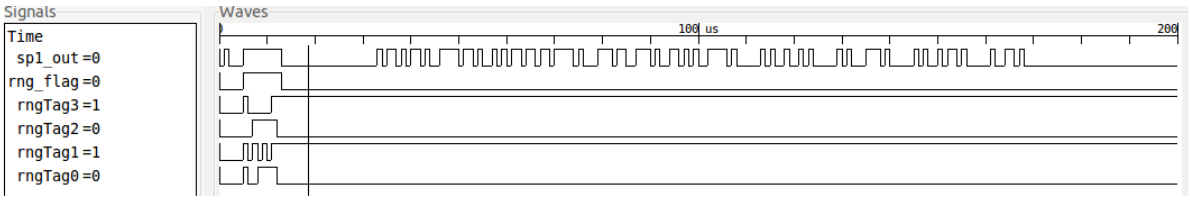

Figura 6.8: Ativação do gerador randômico da *tag* SystemC-AMS

Em seguida, os dados decodificados são enviados para unidade de controle que possui como saída um comando i2c de 12 bits Fig. 6.9. Os comandos i2c que estão sendo obtidos nesta simulação bateram com a resposta que se esperava da unidade de controle, sendo assim, foi verificado que a mesma também estava funcionando corretamente.

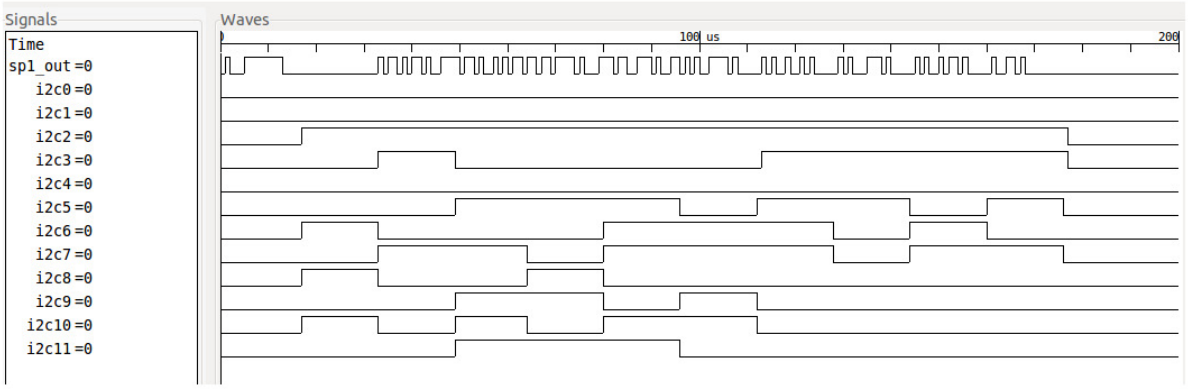

Figura 6.9: Comando i2c paralelo enviado para a unidade de controle de saída em SystemC-AMS

Este comando é então enviado a unidade de controle que serializa este sinal em 16 pulsos de clock, onde nos 4 ultimos pulsos de clock o sinal recebe zero, conforme é apresentado na Fig. 6.10. Foi adotado uma serialização com mais pulsos de clock do que o necessário, para que a interface com a memória conseguisse ajustar este sinal inserindo as *flags* de ACK utilizadas no protocolo de comunicação i2c sem correr o risco de perder a sincronia com a unidade de controle de saída.

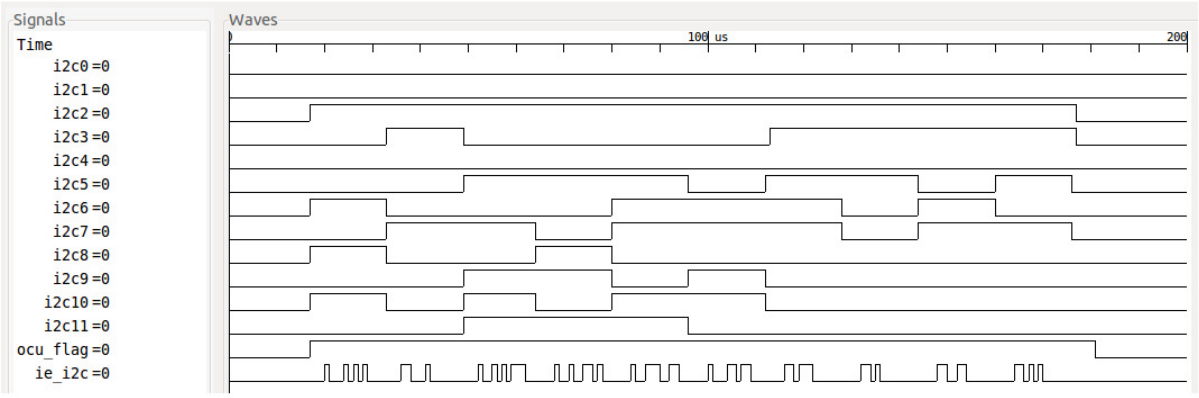

Figura 6.10: Serialização do comando i2c enviado para interface de memória

A interface de memória, por sua vez, recebe o sinal oriundo da unidade de controle de saída e o quebra em 4 bits para comando, ACK, 4 bits para endereçamento, ACK e 4 bits para dados lidos ou escritos 6.11. Este protocolo está um pouco diferente do protocolo i2c citado nas referências devido a uma falha inicial de interpretação, portanto neste protocolo existe bastante espaço para otimização, reduzindo um pouco do atraso que está ocorrendo entre o envio do comando pelo leitor e tempo em que a *tag* demora a enviar a resposta.

O protocolo i2c da memória foi implementado da mesma forma que o protocolo i2c na interface de memória, inclusive se os dois sinais forem mesclados irão gerar o sinal do barramento bidirecional de dados i2c. A interface de memória também é responsável por organizar o dado recebido pela memória e o disponilizar de forma paralela para a unidade de controle de memória, outra operação que está sendo efeutada com sucesso na Fig. 6.11.

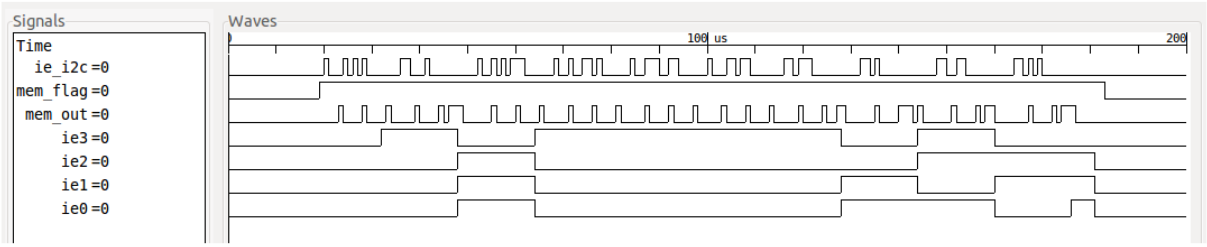

Figura 6.11: Sinais da comunicação memória e interface

Por fim, tem-se que a unidade de controle de saída gera a saída que será enviada ao leitor contendo informações do comando que está respondendo, do endereço ao qual foi requisitado a operação, o número aleatório da *tag* que está respondendo o comando e a confirmação de escrita do dado na memória ou envio do dado lido da memória.

Podemos observar que este valor enviado para o modulador BPSK foi modulado com sucesso, e apesar de termos inserido ruído a este sinal a demodulação também conseguiu recuperar o sinal de entrada. O tempo em que a *tag* demorou para responder o comando do leitor foi de 22 ciclos de clock nesta implementação, conforme pode ser observado na Fig. 6.12.

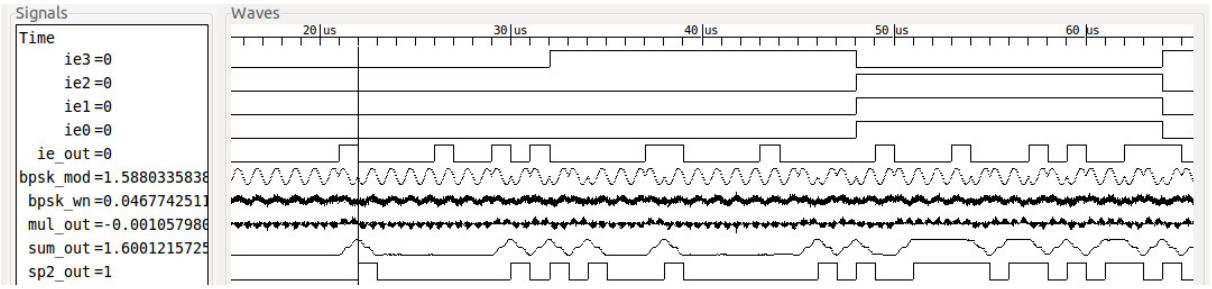

Figura 6.12: Simulação do transceptor BPSK da segunda versão implementado em SystemC-AMS

A arquitetura da *tag* modelada acima, apresentou a saída esperada, porém em alguns pontos essa arquitetura pode ser otimizada visando obter uma resposta mais rápida da *tag* para o leitor.

Em relação a primeira versão, tem-se melhorias significativas, já que agora o leitor passa a ter mais autonomia sobre a *tag*, podendo realizar operações de escrita e leitura, além de minimizar o risco de ocorrer colisões, já que o leitor requisita quem irá responder ao comando através do número aleatório da *tag* e por fim, temos um demodulador que mesmo recebendo um sinal submetido a um ruído extremamente alto, ainda assim consegue obter a saída correta do sinal.

Cabe ressaltar que em ambos os casos, a *tag* recebe continuamente um sinal DC do filtro passa baixa, sendo que este sinal é o que deverá ser limitado e utilizado na *tag* como sua alimentação.

Capítulo 7

Conclusão

Neste documento foi apresentado uma revisão bibliográfica a respeito dos conceitos necessários para se realizar uma modelagem de sistemas heterogêneos, assim como as ferramentas disponíveis para se levantar estes modelos, apresentando pontos fortes e limitações das mesmas. Em seguida foi-se realizada um estudo da arte das arquiteturas de RFID existentes, onde baseando-se nestas arquiteturas foi proposta uma nova arquitetura a ser implementada.

Resumidamente a primeira parte deste trabalho buscou familiarizar o leitor com o sistema heterogêneo que será modelado e com a ferramenta que foi utilizada. Em seguida buscou-se através da modelagem de duas versões de arquiteturas apresentar como a modelagem de um sistema heterogêneo pode ser feita em SystemC/SystemC-AMS. Para tal, a princípio foi descrito detalhadamente as duas versões de arquiteturas que foram modeladas, sendo posteriormente apresentado parte dos códigos que descreveram cada componente do sistema e a interação entre estes blocos de forma a formar o sistema. Os códigos completos das arquiteturas modeladas estão apresentados no anexo I e II.

Apesar de ter sido modeladas duas versões em SystemC-AMS que apresentaram uma saída funcional, ainda existem diversas otimações de arquiteturas a aspectos a ser abordados visando obter uma modelagem mais completa do sistema. Tais como uma estimativa do consumo de potência; uma diminuição do tempo de resposta da *tag*; o teste do modelo em condições extremas de funcionamento; e por fim, otimações do modelo de comunicação implementado visando aproximá-lo ao padrão EPC C1G2 que é o padrão adotado atualmente. Sendo estes aspectos ideias para se realizar um trabalho futuro a respeito desde tema. Outro tema interessante para trabalhos futuros é a inclusão da arquitetura do leitor no modelo, tornando-o cada vez mais próximo da realidade.

O objetivo final deste trabalho era validar a funcionalidade de uma *tag* de RFID em alto nível de abstração visando avaliar o comportamento e limitações da mesma. A modelagem em nível sistêmico provou ser uma estratégia interessante para entender fraquezas existentes no projeto, principalmente em partes referentes a comunicação, ajudando o projetista a tomar as melhores opções para tornar o sistema otimizado. Sendo assim, o objetivo deste trabalho foi alcançado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] CALAZANS, N. L. V. Métodos e ferramentas para o projeto de sistemas digitais. *Anais Sociedade Brasileira de Computação*, p. 34–53, Maio 1995.

[2] Disponível em: <https://forsyde.ict.kth.se>.

[3] Disponível em: <http://www.systemc-ams.org>.

[4] SHILABHADRA, S.; YASSER, M.

Baseband Processor of Multi-Purpose RFID Tag — Department of Electronics and Communication Engineering National Institute of Technology, Rourkela, 2011.

[5] ZONG, H. et al. An ultra low power ask demodulator for passive uhf rfid tag. *IEEE 9th International Conference on ASIC (ASICON),2011*, p. 637–640, Outubro 2011.

[6] YAN, H. et al. Design of low-power baseband-processor for rfid tag. *International Symposium on Applications and the Internet Workshops, 2006. SAINT Workshops 2006*, p. 60–63, Janeiro 2006.

[7] YING, C.; FU-HONG, Z. A system design for uhf rfid reader. *11th IEEE International Conference on Communication Technology, 2008. ICCT 2008.*, p. 301–304, Novembro 2008.

[8] SEMICONDUCTOR, N. Išc-bus specification and user manual. rev. 6. *UM10204, 2014*, Abril 2014.

[9] Disponível em: <https://projects.gnome.org/dia/downld.html>.

[10] GRIMM, C. et al. An introduction to modeling embedded analog/mixed-signal systems using systemc ams extensions. *Open SystemC Initiative (OSCI),2008*, Junho 2008.

[11] ELMROTH, E.; HERNáNDEZ, F.; TORDSSON, J. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Computer Systems*, v. 26, n. 2, p. 245–256, 2010. ISSN 0167-739X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X09001174>.

[12] Disponível em: <http://ptolemy.berkeley.edu>.

[13] PEDRONI, V. A. *Circuit design with VHDL*. [S.l.]: Massachusetts Institute of Technology, 2004.

[14] Disponível em: <http://www.accellera.org>.

[15] SHEN, X. et al. A low-cost uhf rfid tag baseband with an idea cryptography engine. *Internet of Things (IOT), 2010*, p. 1–5, Dezembro 2010.

[16] EPCGLOBAL. Epc radio-frequency identity protocols class-1 generation-2 uhf rfid protocol for communications at 860 mhz - 960 mhz version 1.0.9. *EPCglobal Standard Specification*, 2004.

[17] (OSCI), O. S. I. Systemc ams extensions user?s guide. *SystemC*, Março 2010.

[18] (OSCI), O. S. I. Standard systemc ams extensions language reference manual. *SystemC*, Março 2010.

[19] Disponível em: <http://gtkwave.sourceforge.net/>.

ANEXOS

I. ANEXO I - CÓDIGOS DA PRIMEIRA VERSÃO

Os códigos da primeira versão assim como o *Makefile*, *testbanch* e simulações obtidas neste trabalho estão apresentados na pasta denominada Anexo I no CD.

II. ANEXO II - CÓDIGOS DA SEGUNDA VERSÃO

Os códigos da segunda versão assim como o *Makefile*, *testbanch* e simulações obtidas neste trabalho estão apresentados na pasta denominada Anexo II no CD.