José Edenilson Oliveira Reges

Implementação em VHDL de Sensor Inteligente com Módulo CAN e Amplificador Sensível à Fase

Recife

2010

Universidade Federal de Pernambuco

Programa de Pós-graduação em Engenharia Elétrica

Implementação em VHDL de Sensor Inteligente com Módulo CAN e Amplificador Sensível à Fase

Dissertação

submetida à Universidade Federal de Pernambuco

como parte dos requisitos para obtenção do grau de

Mestre em Engenharia Elétrica

José Edenilson Oliveira Reges

Recife, Junho de 2010.

R333i Reges, José Edenilson Oliveira.
Implementação em VHDL de Sensor Inteligente com Módulo CAN e Amplificador Sensível à Fase / José Edenilson Oliveira Reges. Recife: O Autor, 2010.
xix, 206 folhas., il., gráfs., tabs.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2010.

Orientador: Prof. Edval José Pinheiro Santos.
Inclui Referências e Apêndice.

1. Engenharia Elétrica. 2. Sensores Inteligentes. 3.Redes de Sensores. 4.Amplificador Sensível. 5.Rede CAN. I. Título.

UFPE
621.3 CDD (22. ed.) BCTG/2010-166

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DO MESTRADO ACADÊMICO DE

JOSÉ EDENILSON OLIVEIRA REGES

TÍTULO

"IMPLEMENTAÇÃO EM VHDL DE SENSOR INTELIGENTE COM MÓDULO CAN E AMPLIFICADOR SENSÍVEL À FASE"

A comissão examinadora composta pelos professores: EDVAL JOSÉ PINHEIRO SANTOS, DES/UFPE, HÉLIO MAGALHÃES DE OLIVEIRA, DES/UFPE, e JOSÉ SÉRGIO DA ROCHA NETO, DEE/UFCG sob a presidência do primeiro, consideram o candidato **JOSÉ EDENILSON OLIVEIRA REGES APROVADO.**

Recife, 14 de junho de 2010.

RAFAEL DUEIRE LINS
Coordenador do PPGEE

EDVAL JOSÉ PINHEIRO SANTOS
Orientador e Membro Titular Externo

JOSÉ SÉRGIO DA ROCHA NETO
Membro Titular Externo

HÉLIO MAGALHÃES DE OLIVEIRA
Membro Titular Interno

Aos meus pais.

Agradecimentos

À Deus, pela Luz que Ele tem colocado em meus caminhos durante todos os momentos de minha vida.

Aos meus pais, pelo carinho e pela dedicação proporcionados ao longo de todos esses anos de luta.

Ao professor Edval José Pinheiro Santos, pela orientação, compreensão e, principalmente, pela confiança depositada na execução deste trabalho.

Aos meus amigos do LDN, em especial à Filipe Esteves Távora, pelo enorme apoio oferecido durante o período em que, por motivos profissionais, precisei me ausentar do Mestrado.

Aos meus ex-colegas de trabalho do CESAR e agora amigos, em especial à Marília Souto Maior Lima, pela compreensão durante o período em que precisei me afastar das atividades na *Design House* e priorizar meus estudos no Mestrado.

Aos meus colegas de trabalho da PETROBRAS, em especial à Walmy André Cavalcante Melo da Silva, pela compreensão durante o período em que precisei me afastar das atividades no ATP-ARG e concentrar meus esforços na conclusão deste trabalho.

José Edenilson Oliveira Reges

Universidade Federal de Pernambuco

14 de Junho de 2010

Resumo da Dissertação apresentada à UFPE como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

Implementação em VHDL de Sensor Inteligente com Módulo CAN e Amplificador Sensível à Fase

José Edenilson Oliveira Reges

Junho/2010

Orientador: Edval José Pinheiro Santos, Ph.D.
Área de Concentração: Eletrônica
Palavras-chave: Sensores Inteligentes, Redes de Sensores, IEEE 1451, Rede CAN, Amplificador Sensível à Fase
Número de páginas: xix+206

Neste trabalho são apresentadas a descrição em linguagem VHDL e a implementação em FPGA de um amplificador sensível à fase *(lock-in)* e de um módulo de comunicação CAN para o desenvolvimento de um sensor inteligente inspirado na família de padrões IEEE 1451. O amplificador sensível à fase é utilizado para detecção e condicionamento de sinais. Sua implementação em formato digital possibilita a utilização de técnicas de processamento digital de sinais. A síntese do amplificador utilizou apenas 6% dos recursos lógicos da FPGA escolhida, possibilitando a implementação de vários amplificadores em paralelo, na mesma FPGA. O módulo CAN implementado é capaz de se comunicar em rede com outros módulos CAN, disponíveis comercialmente. A utilização lógica do módulo CAN implementado foi comparada à do módulo *HurriCANe*, desenvolvido pela ESA. O funcionamento dos circuitos foi verificado com êxito a partir de simulações e de testes realizados após a implementação em FPGA. A interligação do módulo CAN ao amplificador *lock-in* foi realizada com sucesso, sendo ocupados apenas 14% dos recursos da FPGA.

Abstract of Dissertation presented to UFPE as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

VHDL Implementation of a Smart Sensor with CAN Module and Lock-in Amplifier

José Edenilson Oliveira Reges

June/2010

Supervisor: Edval José Pinheiro Santos, Ph.D.
Area of Concentration: Electronics
Keywords: Smart Sensors, Sensors Networks, IEEE 1451, CAN Network, Lock-in Amplifier
Number of pages: xix+206

In this work, the VHDL description and the FPGA implementation of both a lock-in amplifier, and a CAN communication module are presented for the development of a smart sensor inspired in the IEEE 1451 family of standards. The lock-in amplifier is used for signal detection and conditioning. Its implementation in digital format allows for the application of digital signal processing techniques. The lock-in amplifier synthesis used only 6% of the logic resources for the selected FPGA, allowing for the implementation of many parallel amplifier in the same FPGA. The implemented CAN module is capable of communicating in a network with other CAN modules, available in the market. The logic utilization of this CAN module is compared to the *HurriCANe* module, developed by ESA. The correct operation of the circuits was verified with simulations, and tests performed after the FPGA implementation. The interligation of the CAN module to the lock-in amplifier was succesfully carried out, using only 14% of the FPGA resources.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Sensores Inteligentes

Um sensor inteligente é um dispositivo que combina circuitos de sensoriamento, transdução, condicionamento, medição, aquisição de dados e comunicação digital [1, 2, 3, 4]. Sensores inteligentes possuem internamente funções de compensação e de processamento de dados, sendo capazes de detectar valores anormais e de fazer tratamento dos valores normais através de seus algoritmos e de parâmetros gravados em sua memória; possuem ainda a capacidade de se comunicar com outros dispositivos, utilizando uma rede de comunicação [3, 4].

Caso o dispositivo seja capaz de atuar num determinado processo, de acordo com um algoritmo pré-definido, tem-se um atuador inteligente. De maneira geral, utiliza-se o termo transdutor inteligente para se referir tanto a um dispositivo sensor quanto a um dispositivo atuador [5]. O diagrama em blocos de um transdutor inteligente genérico é apresentado na Figura 1.1.

Sensores inteligentes possuem inúmeras aplicações nas indústrias petroquímica, automotiva e aeroespacial; no controle de processos, na biomedicina, na agropecuária, entre outros segmentos de mercado [6]. Devido a essa diversidade, os fabricantes de sensores têm buscado desenvolver dispositivos cada vez mais "inteligentes", disponibilizando novas funcionalidades, adicionando maior capacidade de processamento e interligando estes dispositivos a uma rede de comunicação. Tudo isso aliado a uma maior autonomia (baixo consumo de energia) e a um menor custo.

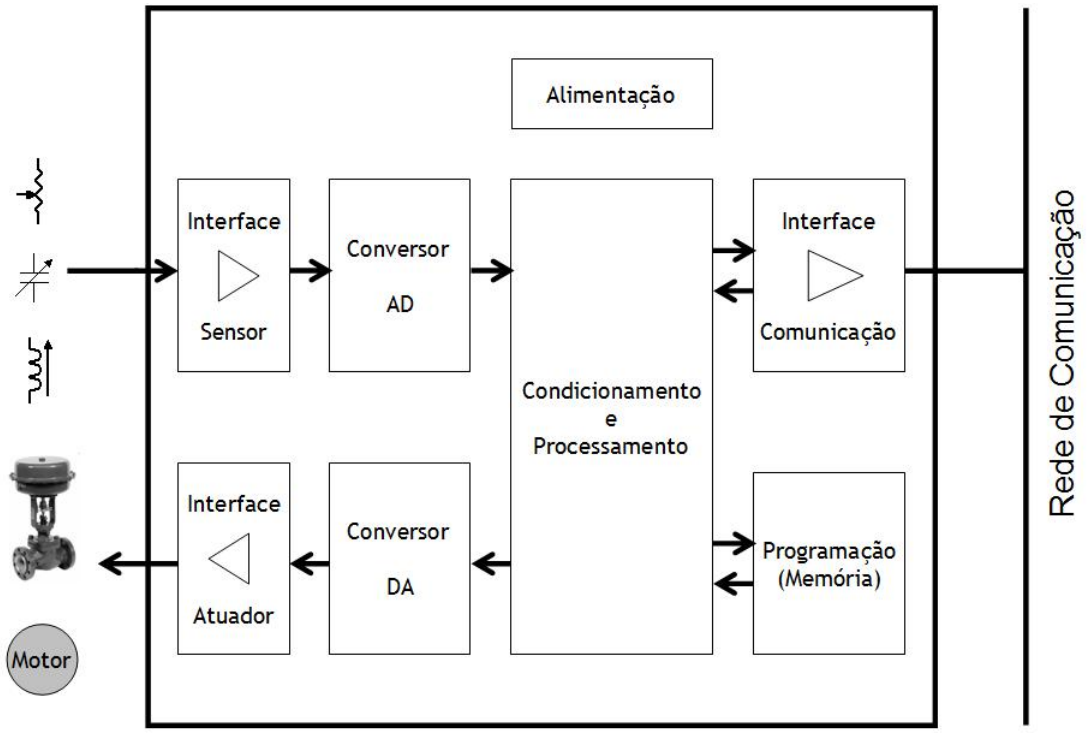


Figura 1.1: Diagrama em blocos de um transdutor inteligente genérico.

Um sensor inteligente tradicional compreende tanto o sistema de medição quanto a interface de comunicação com a rede no mesmo dispositivo. Desse modo, o desenvolvimento do sensor está fortemente relacionado ao tipo de rede na qual o dispositivo será inserido [5].

Existem atualmente diversas implementações de redes de sensores e protocolos de comunicação, cada qual com suas vantagens e desvantagens, dependendo do tipo de aplicação. Dessa forma, os fabricantes de sensores se depararam com o seguinte problema: como integrar os seus dispositivos a toda essa variedade de protocolos de comunicação existentes? Obviamente que o desenvolvimento de um dispositivo para cada tipo de rede existente se tornaria um processo tecnicamente complexo e de custo bastante elevado. Surgiu, então, a necessidade da criação de um padrão, aceito universalmente, que permitisse o desenvolvimento e a integração de novos dispositivos aos sistemas existentes e emergentes [5]. Nesse sentido, o IEEE *(Institute of Electrical and Electronic Engineers)*, em parceria com o NIST *(National Institute of Standards and Technology)* e representantes da indústria, criou um projeto com o objetivo de conceber uma família de padrões que tornasse mais fácil o desenvolvimento de transdutores inteligentes e a integração desses dispositivos às redes, sistemas e ins-

trumentos baseados nas tecnologias atuais e futuras. Nascia aí a família de padrões IEEE 1451 [5].

Neste trabalho é apresentada a proposta de um sensor inteligente baseado na família de padrões IEEE 1451 e a implementação dos circuitos de medição e de comunicação utilizados neste dispositivo.

Na próxima seção, serão apresentados alguns dos principais tipos de redes industriais de comunicação existentes. Uma vez que o estado da arte do desenvolvimento das redes industriais de comunicação está associado à evolução dos sistemas de automação industrial, será realizada uma breve revisão histórica das arquiteturas dos sistemas de controle de processos industriais.

Em seguida, serão apresentados os vários subconjuntos da família de padrões IEEE 1451. Posteriormente, será discutida a arquitetura do sensor inteligente proposto neste trabalho. Finalmente, será apresentado um exemplo de aplicação deste sensor inteligente na medição de impedâncias utilizando tomografia por impedância elétrica.

1.2 Redes Industriais de Comunicação

Uma rede industrial de comunicação é um sistema que permite a troca de informações entre dispositivos como: sensores, atuadores, controladores e estações de supervisão. É, em geral, mais robusta que uma rede de comunicação convencional. Apresenta, tipicamente, os seguintes requisitos [7, 8]:

- Alta disponibilidade: garantia de que o sistema de comunicação estará disponível, apresentando um baixo índice de falhas e um alto tempo médio entre falhas;
- Comunicação em tempo real: a comunicação deve apresentar tempo de resposta previsível;
- Confiabilidade: garantia de que em determinado momento a comunicação será realizada;
- Escalonamento: a rede deve possuir uma política de comunicação bem definida, controlando o acesso dos seus componentes ao meio de comunicação;

- Escalabilidade: a rede deve está preparada para se expandir, sem que a tecnologia utilizada se torne obsoleta ou deixe de atender às necessidades do usuário;
- Facilidade de operação e manutenção: o sistema de comunicação deve utilizar dispositivos *plug-and-play*, facilitando, por exemplo, a substituição de componentes defeituosos;
- Robustez mecânica: os sistemas de conexão utilizados devem ser mais resistentes à desconexões acidentais e à condições ambientais adversas;
- Robustez elétrica: a camada física deve utilizar cabeamento mais imune à ruídos, além de possuir dispositivos de proteção, por exemplo, contra curto-circuitos.

O estado da arte do desenvolvimento das redes industriais de comunicação está associado à evolução dos sistemas de automação industrial. Um breve histórico dessa evolução é apresentado a seguir.

1.2.1 Evolução dos Sistemas de Automação Industrial

Até o início dos anos 40, a instrumentação existente disponibilizava apenas a indicação local das variáveis de processo. Dessa forma, o controle de processos industriais era predominantemente manual, realizado localmente pelo próprio operador do processo. Consequentemente, existia uma grande demanda de operadores no campo. Essa arquitetura de controle além de ineficiente e lenta, era bastante insegura e susceptível à falhas [8].

Um exemplo típico de controle de processo manual e local é apresentado na Figura 1.2. Trata-se de um processo de aquecimento de água fria a partir da troca de calor com o vapor. A temperatura da saída de água quente (variável controlada ou variável de processo) é medida por um sensor e indicada localmente para o operador do processo. O operador (controlador do processo), baseado na indicação da temperatura, ajusta manualmente a abertura da válvula de controle da vazão de entrada de vapor (variável manipulada), de forma a manter a saída de água quente na temperatura desejada (valor de referência ou *set-point*).

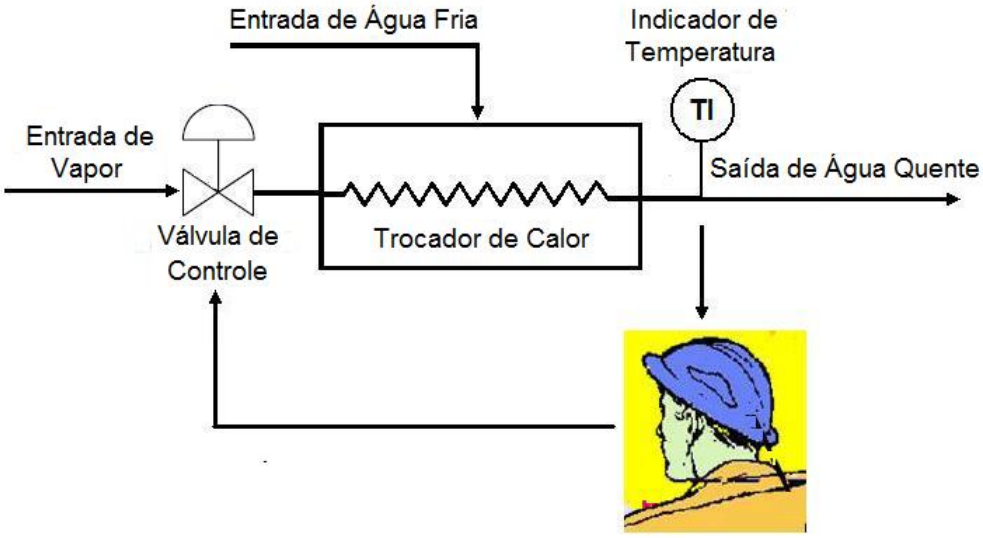


Figura 1.2: Exemplo típico de controle de processo manual e local. O processo de aquecimento de água é controlado manualmente, na planta de processo, pelo operador.

Entre 1940 e 1960 surgiram os primeiros transmissores[1] pneumáticos. Com a utilização desses transmissores, a grandeza medida passou a ser enviada para os controladores automáticos e locais de processo.

Os controladores, baseados no sinal pneumático dos transmissores e utilizando estratégias de controle configuradas no campo, calculavam a ação de correção a ser realizada numa determinada variável manipulada. Consequentemente, o controle passou a ser realizado de forma automática, demandando um menor número de operadores, aumentando a eficiência, a velocidade de resposta e a segurança do processo. Além disso, como os controladores estavam espalhados na planta de processo, o sistema de controle era totalmente distribuído [8]. Um exemplo típico de controle de processo automático e local é apresentado na Figura 1.3.

Na década de 60, com o aumento da complexidade das plantas de processo, se fazia necessário o acompanhamento e o controle remoto das variáveis de processo. Surgiu, então, a sala de controle contendo um painel elétrico com os controladores de processo e as informações das principais variáveis da planta. Consequentemente, houve a migração do operador e dos controladores de processo do campo para a sala

[1] Os transmissores são dispositivos capazes de transmitir um sinal proporcional a uma determinada grandeza medida (pressão, nível, temperatura, etc.). No caso de dispositivos pneumáticos, por exemplo, o sinal transmitido é tipicamente uma pressão de ar na faixa de 3 a 15 psi. Já os transmissores eletrônicos utilizam um sinal de corrente ou de tensão (normalmente 4 a 20 mA ou 0 a 5V, respectivamente).

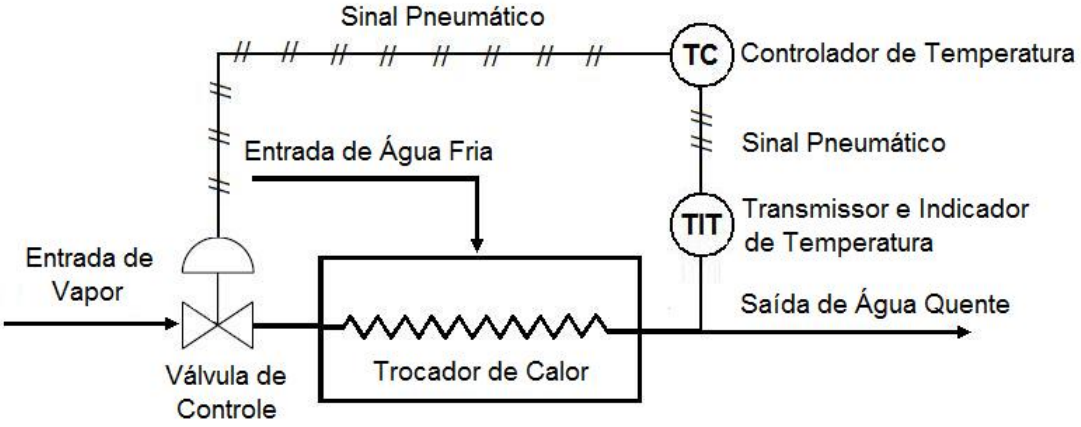


Figura 1.3: Exemplo típico de controle de processo automático e local. O processo de aquecimento de água é controlado automaticamente, na planta de processo, pelo controlador de temperatura.

de controle. Entretanto, apesar dos controladores serem colocados no mesmo painel, o controle continuou a ser distribuído, pois cada controlador era responsável por uma determinada malha de controle [8]. Um exemplo típico de controle de processo automático e remoto é apresentado na Figura 1.4.

Com esse novo paradigma, aumentou-se a segurança do processo e das pessoas, uma vez que o operador não precisaria estar presente no campo durante toda a jornada de trabalho. Além disso, com a migração dos controladores para o painel de controle, aumentou-se a proteção dos equipamentos, pois grande parte desses não ficavam mais expostos ao tempo [8].

Em contrapartida, essa nova arquitetura ocasionou um maior atraso na resposta do sistema. Além disso, o grande número de cabos e o comprimento destes aumentou o custo de instalação e manutenção do sistema. Finalmente, novos modos de falha foram criados (por exemplo, o rompimento de um cabo interligando um transmissor no campo e o controlador no painel de controle) [8].

Na Figura 1.5 é apresentado um antigo painel utilizado numa sala de controle de uma refinaria de petróleo. O detalhe dos controladores no painel é apresentado na Figura 1.6.

Em meados dos anos 70 passaram a ser utilizados instrumentos e transmissores eletrônicos analógicos, em substituição aos instrumentos e transmissores pneumáticos. Além disso, com o desenvolvimento dos microcomputadores e de dispositivos eletrônicos mais resistentes às condições de operação industriais, o controle do pro-

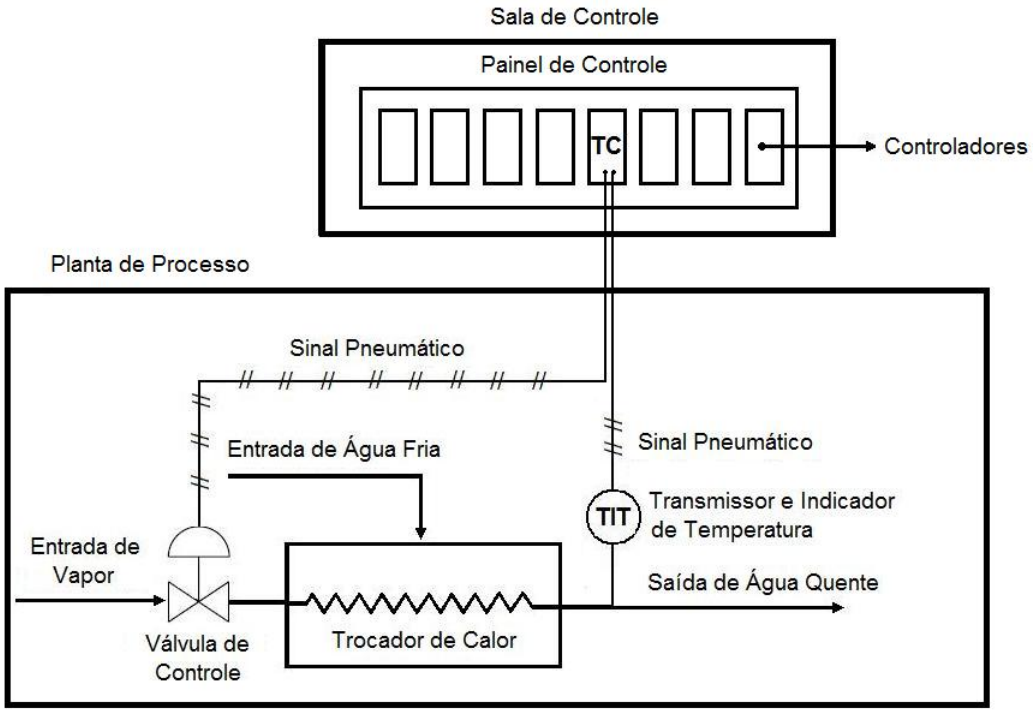


Figura 1.4: Exemplo típico de controle de processo automático e remoto. O processo de aquecimento de água é controlado automaticamente, a partir de um painel remoto na sala de controle.

Figura 1.5: Ilustração de um antigo painel de controle de uma refinaria de petróleo. Na parte superior do painel é representado o fluxograma de engenharia da planta de processo. Na parte inferior estão presentes os controladores de processo, as chaves de configuração, as botoeiras e as indicações das variáveis do processo.

Figura 1.6: Ilustração dos controladores de processo no painel.

cesso passou a ser realizado por um computador central, responsável por controlar toda a planta. O sistema de controle passou então a utilizar uma arquitetura centralizada, conforme ilustrado na Figura 1.7 [7].

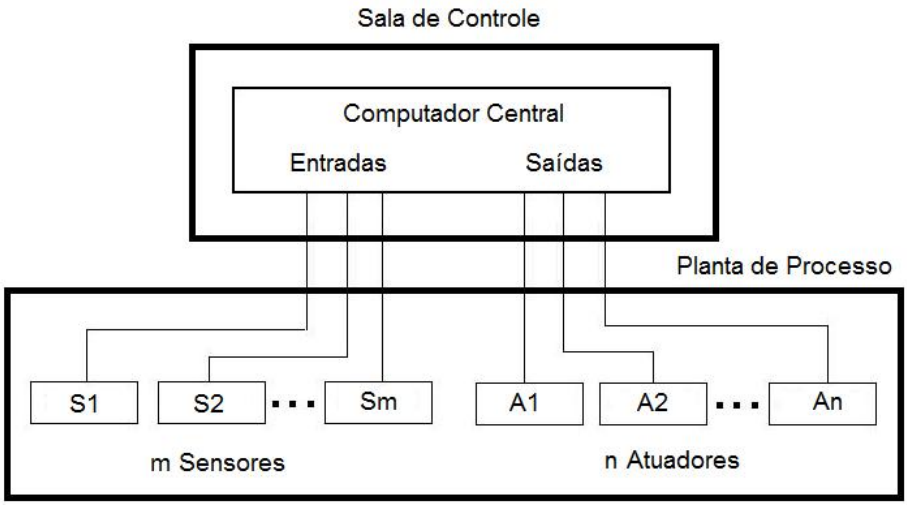


Figura 1.7: Arquitetura típica de um sistema de controle centralizado.Um único computador central é utilizado para controlar todo o processo.

Uma vez que os processos se tornaram cada vez mais complexos, aumentando o número de entradas e saídas e a complexidade dos algoritmos de controle, o computador central passou a necessitar de uma maior capacidade de processamento, de memória e de armazenamento para satisfazer aos requisitos de tempo de resposta, confiabilidade e disponibilidade do sistema. Entretanto, a grande desvantagem desse sistema era a possibilidade de falha no computador central e consequente parada de toda a planta industrial [7].

A partir da década de 80, os sistemas de controle se tornaram parcialmente dis-

tribuídos, utilizando vários computadores interligados desenvolvendo tarefas específicas, descentralizando a capacidade de processamento. Além disso, os dispositivos de aquisição de dados também passaram a ser distribuídos (unidades terminais remotas). Surgem, então, as redes de supervisão e controle e os sistemas DCS[2] *(Distributed Control System)* e SCADA *(Supervisory Control and Data Acquisition)* [7, 8].

Nos sistemas DCS, os níveis de supervisão e controle são fornecidos no mesmo pacote, utilizando normalmente uma rede de comunicação proprietária. Sistemas DCS são geralmente utilizados em aplicações que demandam estratégias de controle muito complexas e intertravamentos simples, admitindo tempos de resposta mais altos [8]. Além disso, por serem constituídos por um pacote fechado, os sistemas DCS são tipicamente utilizados em locais de pequena dispersão geográfica, como por exemplo, uma refinaria de petróleo ou uma usina termoelétrica.

Por outro lado, nos sistemas SCADA, os níveis de supervisão e controle são fornecidos em pacotes distintos, utilizando tipicamente uma rede de comunicação aberta. O nível de controle é geralmente composto por CLP (Controladores Lógicos Programáveis). Sistemas SCADA são geralmente utilizados em aplicações que demandam estratégias de controle simples e intertravamentos mais complexos, apresentando um menor tempo de resposta [8]. São utilizados normalmente em locais de grande dispersão geográfica, tais como: campos de produção de petróleo (terrestres e marítimos), monitoramento de dutos, etc.

Vale salientar, entretanto, que as diferenças entre os sistemas DCS e SCADA vêm desaparecendo ao longo do tempo, uma vez que os sistemas DCS vêm apresentando tempo de resposta cada vez menor e os CLPs utilizados em sistemas SCADA estão cada vez mais poderosos e com maior capacidade de processamento. As arquiteturas dos sistemas DCS e SCADA são apresentadas nas Figuras 1.8 e 1.9, respectivamente.

Finalmente, em meados dos anos 90 até os dias atuais, com o acelerado desenvolvimento da microeletrônica, surgiram novos dispositivos cada vez mais baratos, de menores dimensões e com capacidade cada vez maior de processamento. Os dispositivos de campo (sensores e atuadores) passaram a englobar novas funções, entre elas as de processamento de sinais e de controle. Isso fez com que a descentralização geográfica dos controladores se tornasse viável. Além disso, esses dispostivos pas-

[2]Também conhecido como SDCD, Sistema Digital de Controle Distribuído.

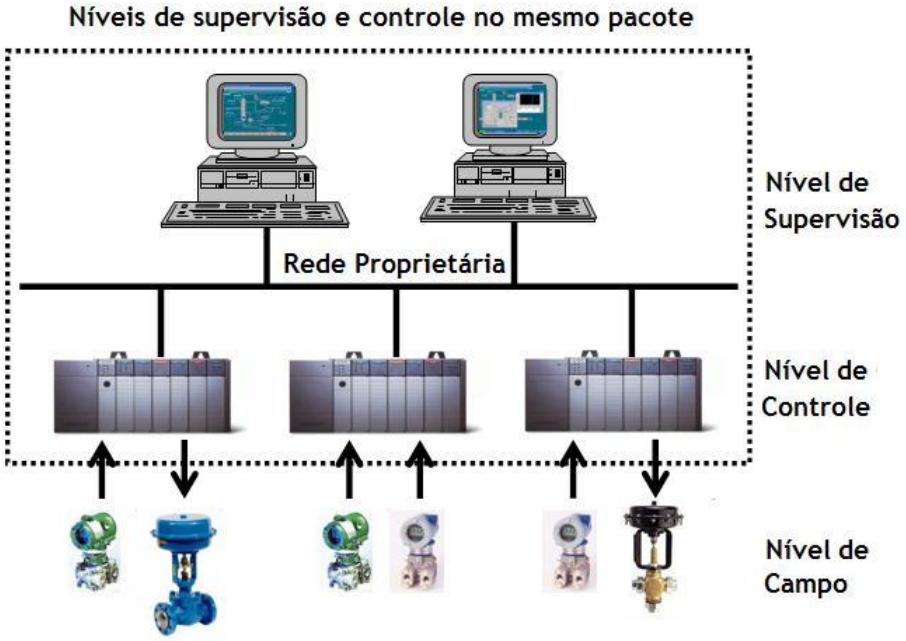


Figura 1.8: Arquitetura típica de um sistema DCS.

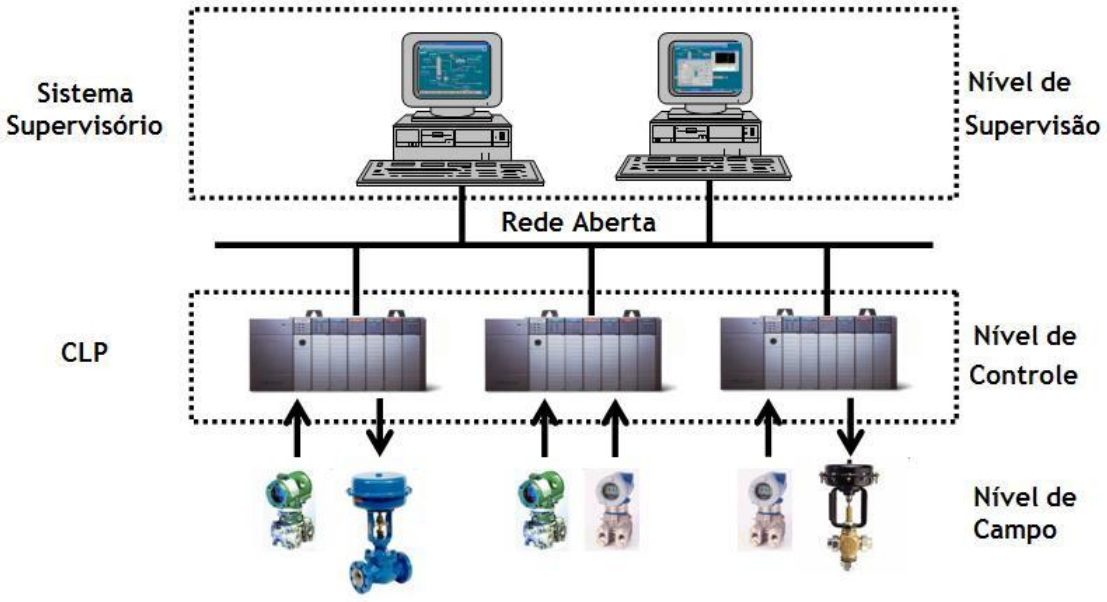


Figura 1.9: Arquitetura típica de um sistema SCADA.

saram a ser interligados em rede. Dessa forma, o controle do processo volta ao campo (controle local) e o sistema volta a ser totalmente distribuído. Surgem as redes de campo [7, 8].

A interligação em rede de dispositivos de campo é ilustrada na Figura 1.10. Alguns exemplos de redes de dispositivos de campo: *Foundation Fieldbus, DeviceNet, CANopen, Profibus, Seriplex, FIP I/O, ASi, Interbus*, entre outros [7].

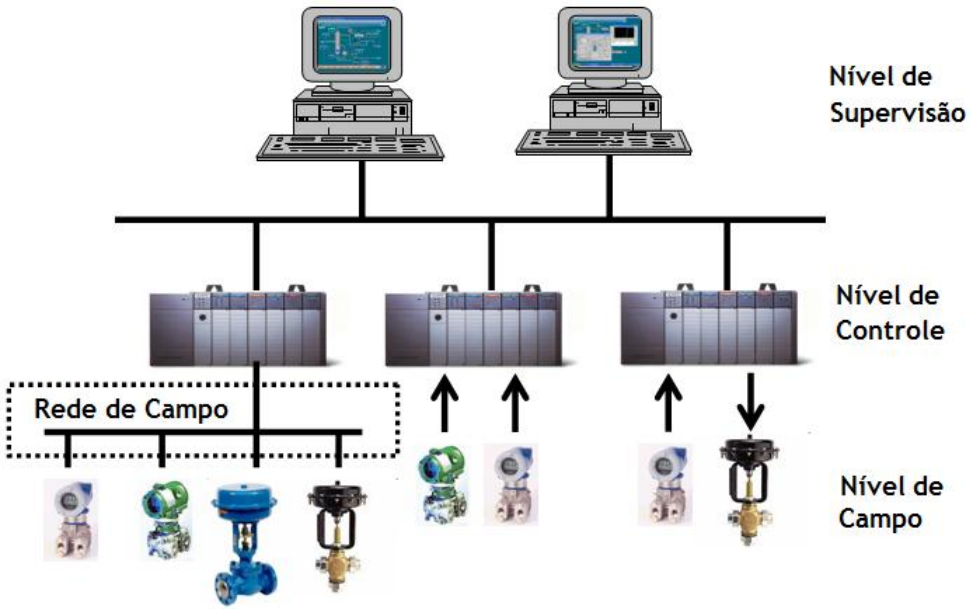


Figura 1.10: Interligação em rede de dispositivos de campo. Através da rede de campo, sensores, atuadores e outros equipamentos de campo podem comunicar-se entre si e/ou com o controlador.

Conforme discutido anteriormente, toda essa variedade de protocolos de comunicação tem dificultado o desenvolvimento de transdutores inteligentes e a integração de novos dispositivos à diversidade de redes de campo e sistemas existentes. Uma alternativa de padronização é proposta na família de padrões IEEE 1451, que será discutida na próxima seção.

1.3 IEEE 1451

A família IEEE 1451 é composta por oito padrões e descreve um conjunto aberto de interfaces de comunicação que permite o acesso a dados do transdutor inteligente, independente da rede na qual o dispositivo está inserido, conectando sensores e atuadores a microprocessadores, sistemas de instrumentação, redes industriais de comu-

nicação, tanto ao nível de campo quanto ao nível de controle [5]. É composta pelos seguintes padrões:

- IEEE P1451.0: *Common Functions, Communications Protocols and Transducer Electronic Data Sheets (TEDS) Formats*;
- IEEE 1451.1: *Network Capable Application Processor (NCAP) Information Model*;
- IEEE 1451.2: *Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*;
- IEEE 1451.3: *Digital Communication and Transducer Electronic Data Sheet (TEDS) Formats for Distributed Multidrop Systems*;
- IEEE 1451.4: *Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*;
- IEEE P1451.5: *Wireless Communication Protocols and Transducer Electronic Data Sheets (TEDS) Formats*;
- IEEE P1451.6: *A High-Speed CANOpen Based Transducer Network Interface for Intrinsically Safe and non-Intrinsically Safe Applications*;
- IEEE P1451.7: *Transducers to Radio Frequency Identification (RFID) Systems Communication Protocols and Transducer Electronic Data Sheet Formats.*

Alguns padrões encontram-se na fase de desenvolvimento (proposta). Isto é indicado pela letra "P", antecedendo o número do padrão. A seguir, serão apresentados alguns conceitos fundamentais existentes na descrição de cada padrão.

1.3.1 *Smart Transducer Interface Module* (STIM)

O STIM é o módulo transdutor propriamente dito e independe do tipo rede de comunicação na qual o transdutor inteligente será inserido. Contém os circuitos de transdução, condicionamento de sinal, medição, conversão e aquisição de dados. É composto por sensores, atuadores, conversores analógico-digital (AD), conversores

digital-analógico (DA), entradas e saídas digitais, em qualquer combinação. O STIM pode conter mais de um transdutor. Neste caso, tem-se um STIM multicanal ou multivariável [5].

O STIM também possui o TEDS *(Transducer Electronic Data Sheet)*, dispositivo de memória que armazena os dados e as informações de configuração do transdutor. Estes dados e informações, por sua vez, são transferidos entre o STIM e o NCAP *(Network Capable Application Processor)* através de uma lógica de controle. Esta lógica pode ser implementada, por exemplo, por um circuito discreto, por um ASIC *(Application Specific Integrated Circuit)* ou por um microprocessador [5].

O diagrama em blocos de um STIM sensor, um STIM atuador e um STIM sensor e atuador são ilustrados nas Figuras 1.11, 1.12 e 1.13, respectivamente.

1.3.2 *Network Capable Application Processor* (NCAP)

O NCAP é o dispositivo que implementa a interface entre o STIM e a rede de comunicação. Consequentemente, o NCAP depende do tipo de rede na qual o transdutor inteligente será inserido. É responsável pela solicitação e obtenção de dados do STIM, transmissão e recepção de mensagens via rede de comunicação e pela execução das funções de aplicação. Também provê alimentação ao circuito do STIM [5].

O NCAP é composto basicamente por um *Driver* STIM, por um programa aplicativo (usualmente gravado em *Firmware*), por um controlador de protocolo de rede e por um modelo orientado à objeto, conforme especificado no padrão IEEE 1451.1, que será descrito posteriomente.

O *driver* STIM possui quatro funções principais [5]:

- Analisador do TEDS: conhece a estrutura do TEDS e monta o dado em peças significativas;
- *Driver* de Interface: *software* responsável pela aquisição de dados através da interface com o STIM;
- *Driver* API: provê acesso aos blocos do TEDS, leitura dos sensores, controle dos atuadores, disparos e interrupções, conforme descrito no padrão IEEE 1451.2;

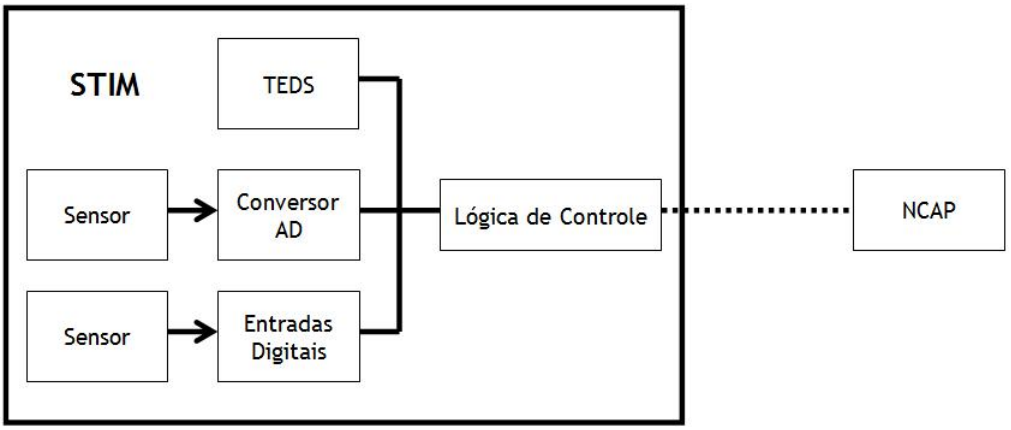


Figura 1.11: Diagrama em blocos de um STIM sensor.

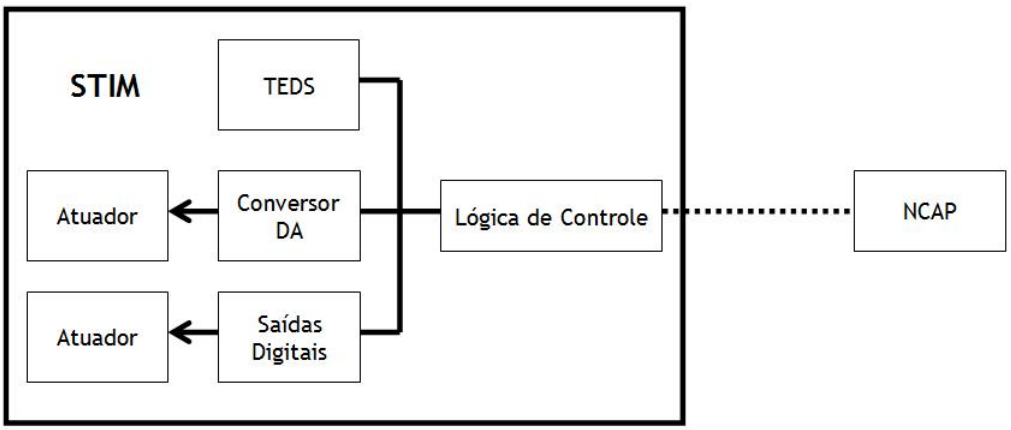


Figura 1.12: Diagrama em blocos de um STIM atuador.

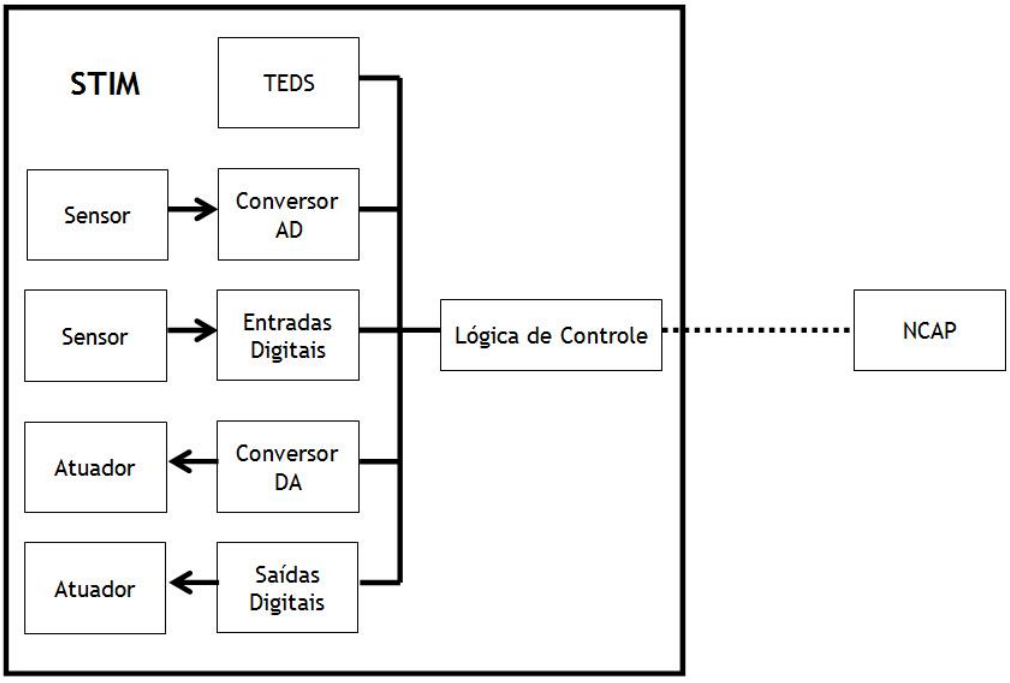


Figura 1.13: Diagrama em blocos de um STIM sensor e atuador.

- Correção de Dados: algoritmo que converte os dados brutos lidos do STIM em unidades especificadas no TEDS.

Um NCAP inicia a medição ou uma determinada ação após disparar uma requisição ao STIM. Uma vez que a medição foi realizada ou a ação completada, o STIM responde com um sinal de reconhecimento. Além disso, o STIM pode interromper o NCAP se uma exceção ocorrer. Tipos comuns de exceção: erro de *hardware*, falha de calibração, falha de auto-teste, etc.

O diagrama em blocos de um NCAP é ilustrado na Figura 1.14.

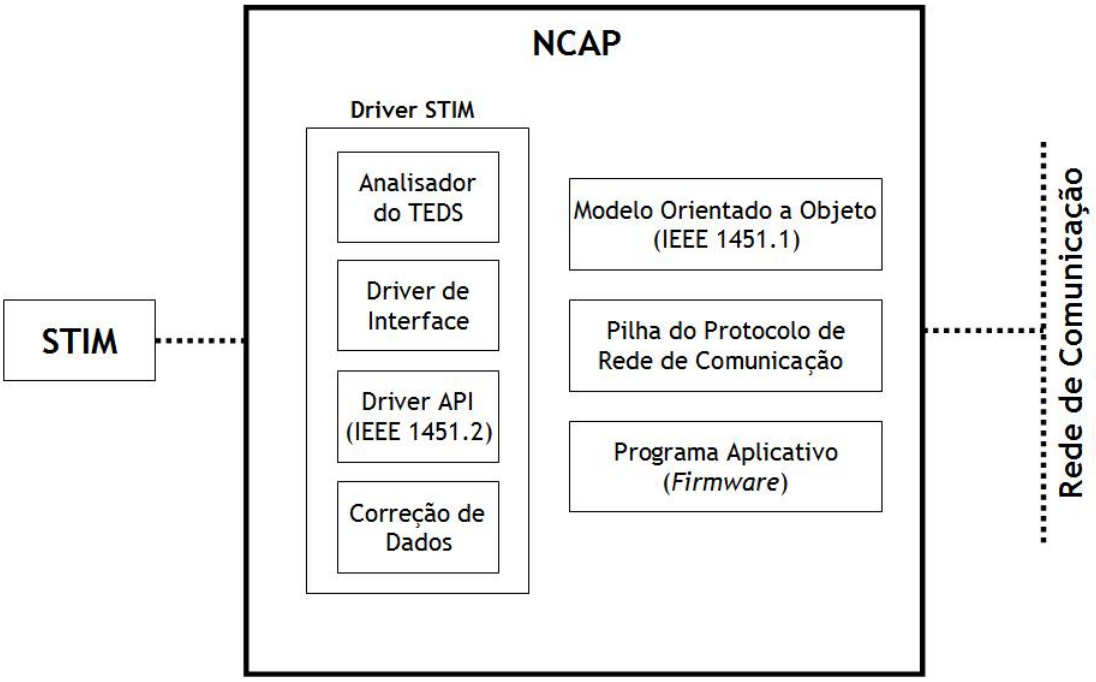


Figura 1.14: Diagrama em blocos de um NCAP.

1.3.3 *Transducer Electronic Data Sheet* (TEDS)

O TEDS é um dispositivo de memória presente no transdutor, que armazena informações como: identificação do transdutor, nome do fabricante, modelo do equipamento, número de série, dados de calibração, faixa de medição, etc. É a documentação, a folha de dados eletrônica do transdutor [5].

O TEDS permite que o transdutor se auto identifique na rede, facilitando a configuração automática do sistema. Esta capacidade de auto-identificação do transdutor é importante na manutenção do sistema, no diagnóstico de falhas, etc. O TEDS

pode ser atualizado para o sistema na sua energização *(power-up)* ou a partir de uma requisição.

Uma vez que o STIM é energizado, os dados contidos no TEDS tornam-se disponíveis ao NCAP para uso local e para serem enviados a outros dispositivos a partir da rede de comunicação, caso necessário. A partir da leitura do TEDS, o NCAP conhece quão rapidamente ele pode se comunicar com o STIM, qual o número de canais do STIM e qual o formato de dados de cada canal.

A estrutura de dados do TEDS é dividida em cinco partes [5]: Meta-TEDS, TEDS de Canal, TEDS de Calibração, TEDS de Aplicação e TEDS de Expansão.

- Meta-TEDS: contém o campo de dados comum a todos os transdutores conectados ao STIM. Também contém uma descrição geral da estrutura de dados do TEDS, parâmetros de temporização, etc.;
- TEDS de Canal: contém informações sobre unidades físicas, incerteza, faixa de medição, etc.;
- TEDS de Calibração: contém informações sobre os parâmetros de calibração e o intervalo de calibração de um transdutor. Também provê as constantes necessárias à conversão dos dados brutos para unidades de engenharia, no caso de sensores, ou à conversão de dados em unidades de engenharia para a forma requerida por um atuador;
- TEDS de Aplicação: contém a aplicação específica de acordo com o uso do transdutor pelo usuário final;
- TEDS de Expansão: disponibilizada para futuras implementações.

A arquitetura de rede de sensores inteligentes baseada no padrão IEEE 1451 é apresentada na Figura 1.15, ilustrando como a aplicação dos conceitos de STIM e NCAP podem facilitar o desenvolvimento de transdutores inteligentes. Os fabricantes podem desenvolver módulos transdutores (STIM) mais genéricos, independentes da rede de comunicação. Em paralelo ao desenvolvimento de STIM independentes, os fabricantes podem desenvolver módulos processadores de comunicação voltados à uma determinada rede, ou à várias redes diferentes, e integrar estes módulos de comunicação ao STIM genérico. Além disso, surge a figura do integrador de sistemas, que

pode utilizar um STIM e um NCAP de fabricantes diferentes, uma vez que as interfaces de comunicação entre STIM e NCAP seguem um padrão aberto. O usuário final tem, portanto, uma vasta gama de possibilidades, podendo implementar sua rede de sensores inteligentes de acordo com as suas necessidades, sem que seja necessária a criação de um projeto específico, iniciado do zero, restrito à aplicação transdutora e à interface de comunicação.

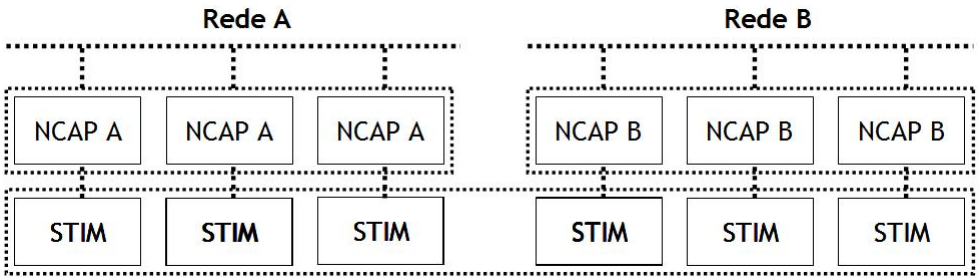


Figura 1.15: Arquitetura de rede de sensores inteligentes baseada na família de padrões IEEE 1451. Nesta ilustração, o mesmo módulo transdutor (STIM) é utilizado, independentemente da rede de comunicação. Por outro lado, o NCAP é projetado de acordo com o tipo de rede.

1.3.4 IEEE P1451.0

No padrão proposto IEEE P1451.0 são definidos o TEDS e um conjunto de comandos e operações comuns à família de padrões IEEE 1451, de maneira a permitir que o acesso aos transdutores seja realizado independentemente do meio físico de comunicação entre os transdutores e o NCAP. Consequentemente, o acesso ao transdutor pelo NCAP deve ser realizado da mesma forma, independentemente se o meio físico entre transdutor e NCAP é implementado utilizando rede cabeada ou sem fio [5].

O padrão IEEE P1451.0 encontra-se atualmente em desenvolvimento.

1.3.5 IEEE 1451.1

No padrão IEEE 1451.1 é definido um modelo orientado à objeto, presente no NCAP, que descreve o comportamento do módulo de transdução inteligente (STIM). São definidos ainda os modelos de comunicação suportados. Entre eles estão os modelos cliente-servidor e produtor-consumidor. O programa aplicativo executado no NCAP

comunica-se com os transdutores através de diferentes camadas físicas, de acordo com a aplicação [5].

O padrão IEEE 1451.1 já foi publicado e pode ser adquirido através do IEEE.

1.3.6 IEEE 1451.2

No padrão IEEE 1451.2 é definida uma interface ponto-a-ponto entre o STIM e o NCAP. O padrão original descreve a camada de comunicação baseada na interface SPI *(Serial Peripheral Interface)*, adicionando linhas HW para controle de fluxo e temporização. Entretanto, o padrão está sendo revisado no intuito de oferecer suporte também à interface UART *(Universal Asynchronous Receiver and Transmitter)* [5].

O padrão IEEE 1451.2 já foi publicado e pode ser adquirido através do IEEE.

Na Figura 1.16 é ilustrada a aplicação dos padrões IEEE 1451.1 e IEEE 1451.2. Enquanto no IEEE 1451.1 é definido o modelo orientado à objeto do STIM (independente da interface STIM-NCAP), no padrão IEEE 1451.2 é definida a interface de comunicação entre o STIM e o NCAP.

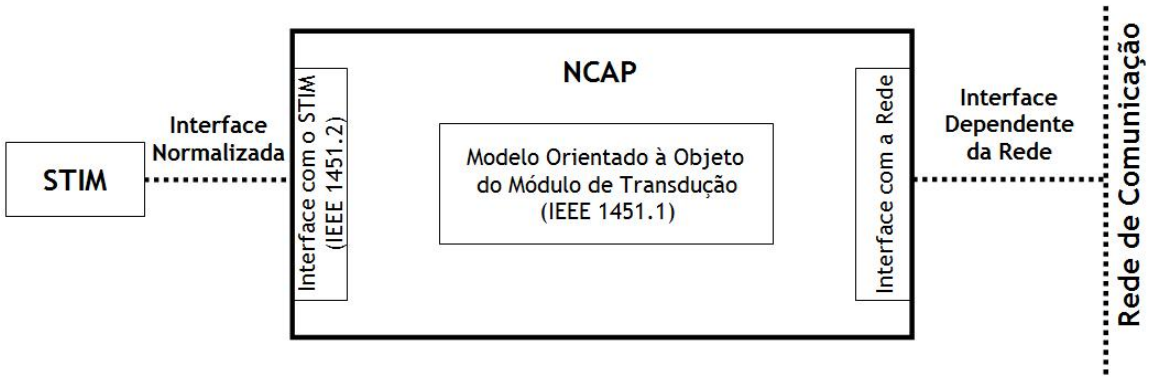


Figura 1.16: Exemplo de aplicação dos padrões IEEE 1451.1 e IEEE 1451.2. O modelo orientado à objeto do módulo de transdução (STIM) é padronizado de acordo com IEEE 1451.1. Por outro lado, a interface normalizada entre o STIM e o NCAP é definida no padrão IEEE 1451.2.

1.3.7 IEEE 1451.3

No padrão IEEE P1451.3 é definida uma interface entre o STIM e o NCAP, baseada numa arquitetura de comunicação distribuída. Essa interface permite que vários

STIMs sejam interligados entre si e ao NCAP a partir de uma rede multi-ponto, compartilhando o mesmo par de fios [5].

O padrão IEEE 1451.3 já foi publicado e pode ser adquirido através do IEEE.

Na Figura 1.17 é ilustrada a aplicação do padrão IEEE 1451.3, definindo uma interface de comunicação entre o NCAP e uma rede de transdutores.

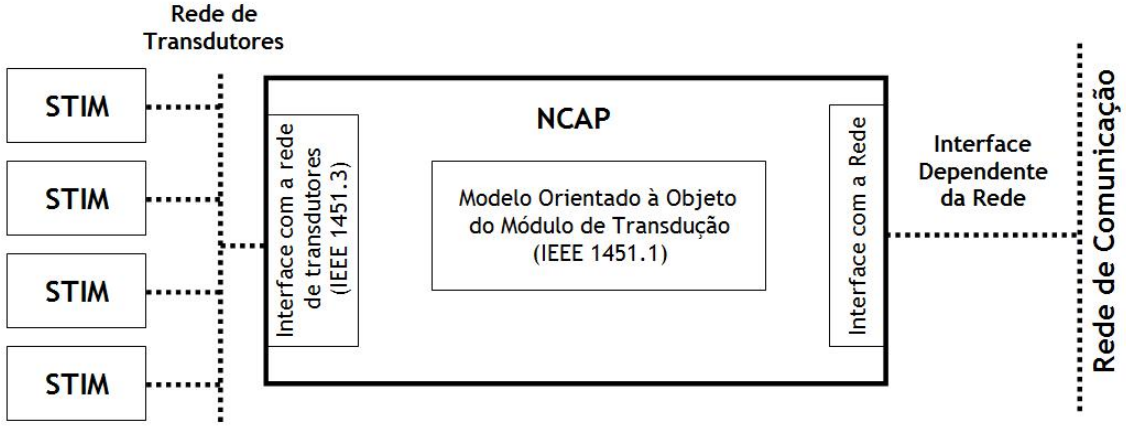


Figura 1.17: Exemplo de aplicação do padrão IEEE 1451.3 definindo uma interface normalizada entre o NCAP e uma rede de transdutores.

1.3.8 IEEE 1451.4

No padrão IEEE 1451.4 é definida uma interface mista para transdutores analógicos, com modos de operação analógico e digital. Um TEDS é adicionado a um sensor tradicional, contendo um amplificador FET, excitado por uma corrente constante através de um par de fios. O modelo de TEDS também é refinado para permitir que um mínimo de dados pertinentes fossem armazenados num dispositivo de memória fisicamente pequena, como requerido por pequenos sensores (mais simples). Modelos (padronizados) são usados para descrever a estrutura de dados da TEDS. Os modelos atuais abrangem acelerômetros, extensômetros *(strain gauges)*, sensores de malha de corrente, microfones, etc. [5].

O padrão IEEE 1451.4 já foi publicado e pode ser adquirido através do IEEE.

1.3.9 IEEE P1451.5

No padrão IEEE P1451.5 é definida uma interface sem fio entre o STIM e o NCAP. Algumas das interfaces físicas suportadas são definidas nos padrões IEEE 802.11

(WiFi), IEEE 802.15.1 *(Bluetooth)* e IEEE 802.15.4 *(ZigBee)* [5].

A interface sem fio definida no padrão IEEE P1451.5 em conjunto com a TEDS, as operações e os comandos definidos no padrão IEEE P1451.0, visam proporcionar a interoperabilidade entre os dados transmitidos a partir de qualquer um dos três protocolos sem fio considerados [5].

O padrão IEEE P1451.5 encontra-se atualmente em desenvolvimento.

1.3.10 IEEE P1451.6

No padrão IEEE P1451.6 é definida uma interface de rede de alta velocidade *CANopen* para comunicação entre o STIM e o NCAP, suportando aplicações nas áreas de instrumentação e controle de processos industriais, tanto em ambientes intrínsecamente seguros quanto não-intrínsecamente seguros [5].

Os parâmetros do TEDS são definidos de forma a permitir a compatibilidade de dados entre os dispositivos na rede *CANopen*, que podem ser desde um simples sensor até um controlador de malha fechada de alto desempenho [5].

Nesse sentido, é definido um mapeamento do TEDS no dicionário objeto *CANopen*, como também mensagens de comunicação, dados de processo, parâmetros de configuração e diagnóstico, adotando como referência o perfil de dispositivo CiA *(CAN in Automation)* 404 *(CANopen Device Profile for Measuring Devices and Closed-Loop Controllers)*.

O padrão IEEE P1451.6 encontra-se atualmente em desenvolvimento.

Um exemplo de aplicação do padrão IEEE P1451.6 é apresentado na Figura 1.18.

1.3.11 IEEE P1451.7

No padrão IEEE P1451.7 são descritos métodos de comunicação, formatos de dados e TEDS para sensores trabalhando em cooperação com sistemas de identificação por rádio-frequência, RFID *(Radio Frequency Identification)*.

O padrão IEEE P1451.7 encontra-se atualmente em desenvolvimento.

1.3.12 Aplicações da Família de Padrões IEEE 1451

A família de padrões IEEE 1451 pode ser aplicada, por exemplo, em [5]:

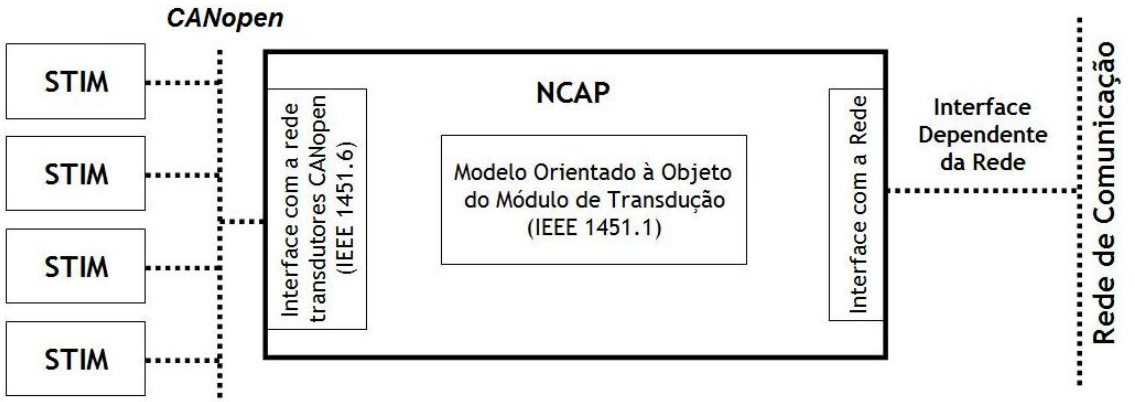


Figura 1.18: Exemplo de aplicação do padrão IEEE 1451.6 definindo uma interface normalizada entre o NCAP e uma rede de transdutores *CANopen*.

- Monitoração Remota: os parâmetros físicos medidos por um STIM sensor podem ser monitorados remotamente através do NCAP, que possui a capacidade de enviar os dados resultantes das medições do sensor através da rede de comunicação. Cada estação conectada à rede pode monitorar as medições realizadas e os demais dados do sensor. Dependendo da necessidade, essas informações podem até mesmo serem enviadas através da Internet;

- Atuação Remota: permite que uma estação remota possa manipular a saída de um STIM atuador a partir do NCAP;

- Controle Distribuído (atuação baseada em medição local): um STIM contendo tanto sensores quanto atuadores pode realizar a medição de uma variável de processo e executar a ação de correção numa determinada variável manipulada, exercendo a função de controle local. A função de controle pode ser configurada por qualquer NCAP da rede;

- Controle e Medição Colaborativa: um conjunto formado por um NCAP conectado a um STIM sensor e outro NCAP conectado a um STIM atuador pode realizar medições remotas e controlar operações, de forma colaborativa, comunicando-se um ao outro através de uma rede de NCAPs.

Estes exemplos de aplicação são apresentados de forma simplificada na Figura 1.19.

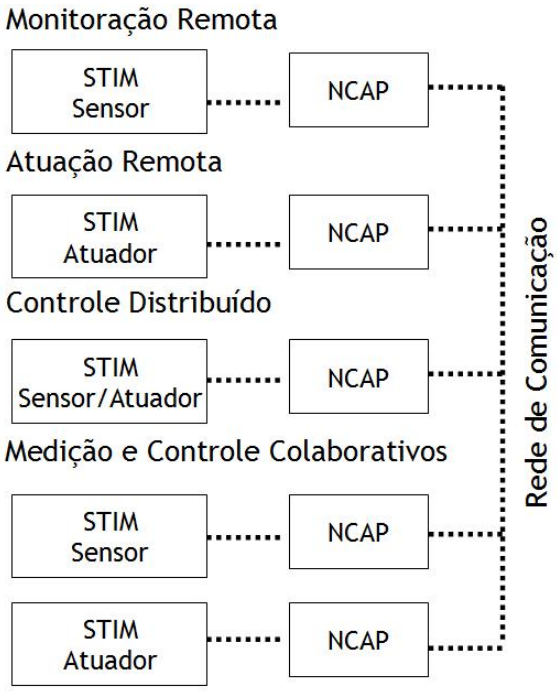


Figura 1.19: Exemplos de aplicação da família de padrões IEEE 1451.

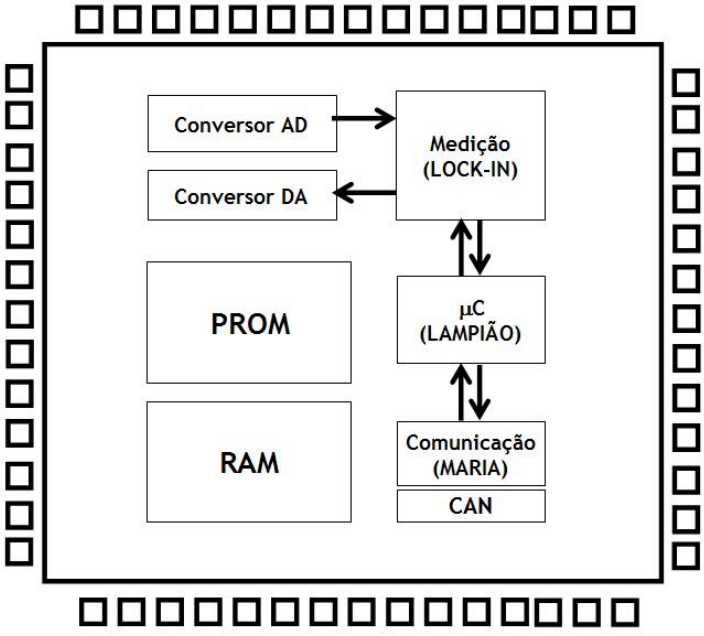


Figura 1.20: Arquitetura do sensor inteligente proposto.

1.4 Arquitetura do Sensor Inteligente Proposto

Na Figura 1.20 é apresentada a arquitetura do sensor inteligente integrado em desenvolvimento no LDN (Laboratório de Dispositos e Nanoestruturas) da UFPE (Universidade Federal de Pernambuco). Este dispositivo reúne, no mesmo *chip*:

- Circuitos de aquisição de dados e geração de sinal: conversores AD e DA;
- Circuito de medição, condicionamento e processamento de sinais: amplificador sensível à fase (LOCK-IN);
- Lógica de Controle: microcontrolador LAMPIÃO (LDN - Arquitetura de Microcontrolador e Propriedade Intelectual para automaçÃO) com um SOTR (Sistema Operacional de Tempo Real) embarcado para aquisição de dados e comunicação;
- Interface de Rede: módulo MARIA (Módulo de Acesso à Rede para Instrumentação Avançada), contendo uma implementação do protocolo CAN *(Controller Area Network)*;
- Memória PROM: para armazenamento da folha de dados eletrônica do sensor;
- Memória RAM: para armazenamento de dados durante a execução da aplicação.

Um exemplo de aplicação do sensor inteligente proposto neste trabalho é discutido na próxima seção.

1.4.1 Exemplo de Aplicação: Medição de Vazão de Fluxos Multifásicos utilizando Tomografia por Impedância Elétrica

Fluxos multifásicos são fluidos em movimento em que estão presentes mais de um componente. Eles ocorrem em diversos sistemas reais, desde a medicina (por exemplo, no fluxo sanguíneo) até a indústria de petróleo.

Em particular, na indústria de petróleo, o óleo extraído é uma mistura de óleo pesado, água, gás, além de alguns sedimentos [9]. Com a quebra de monopólio do

setor, tornou-se interessante o desenvolvimento de medidores de vazão para fluxos multifásicos desse tipo. Esses medidores serão de grande importância na determinação precisa da produção de um poço e para a correta cobrança de impostos [10].

A medição de vazão em fluxos multifásicos pode ser dividida em duas etapas. Na primeira, mede-se a vazão total e na segunda mede-se a fração de volume dos componentes do fluxo [10].

Para medir a vazão total pode-se utilizar uma das diversas técnicas existentes para medição de vazão em fluxo monofásico: Placa de Orifício, Venturi, V-Cone, Annubar, Vortex, Coriolis, Ultrassom, Rotâmetro, Medidor Térmico, Deslocamento Rotativo, Turbina, Medidor Magnético, etc.

Para medir a fração de volume pode-se utilizar uma técnica tomográfica não-invasiva, como a Tomografia por Impedância Elétrica. Com esta técnica, a distribuição de fases no interior de uma tubulação de petróleo pode ser obtida, estimando-se o percentual de óleo, água e gás que compõem o óleo extraído.

Tomografia por Impedância Elétrica é uma técnica não-intrusiva de reconstrução de imagens baseada no princípio que materiais diferentes têm propriedades elétricas diferentes. Consiste em medir a impedância elétrica (resistência, capacitância, indutância ou combinação dessas quantidades) usando uma matriz de eletrodos instalados ao redor de uma tubulação [11, 12, 13, 14], conforme ilustrado na Figura 1.21. Um algoritmo de reconstrução de imagens explora estas propriedades elétricas de maneira a produzir uma imagem da seção transversal sobre a qual os eletrodos estão distribuídos.

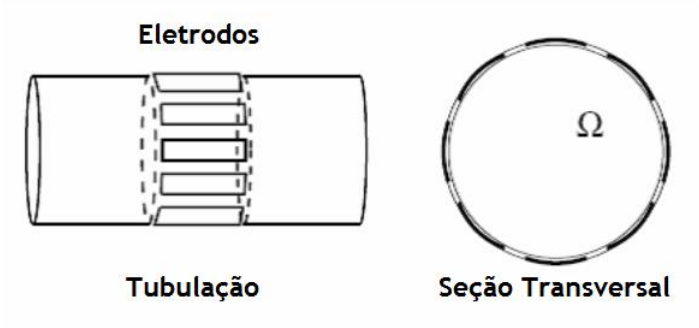


Figura 1.21: Configuração de uma tubulação com eletrodos de medição e sua seção transversal Ω.

Existem basicamente dois métodos de medição utilizados em Tomografia por

Impedância elétrica. No primeiro método, um dos eletrodos de medição é aterrado eletricamente e uma tensão elétrica conhecida é aplicada em cada um dos eletrodos de medição restantes, medindo-se as correntes elétricas resultantes. As correntes elétricas resultantes são desconhecidas a priori e dependem da distribuição de impedância elétrica no interior da tubulação [15].

Tipicamente, por questões de segurança, um segundo método de medição é utilizado. Nesse método alternativo, aterra-se eletricamente um dos eletrodos de medição e injeta-se uma corrente elétrica conhecida em cada um dos eletrodos de medição restantes, medindo-se as tensões elétricas resultantes. Neste caso, as tensões elétricas resultantes são desconhecidas a priori e dependem da distribuição de impedância elétrica no interior da tubulação [15].

A injeção de corrente elétrica e a medição dos potenciais nos eletrodos podem seguir diversas topologias [15]. Uma delas é apresentada na Figura 1.22.

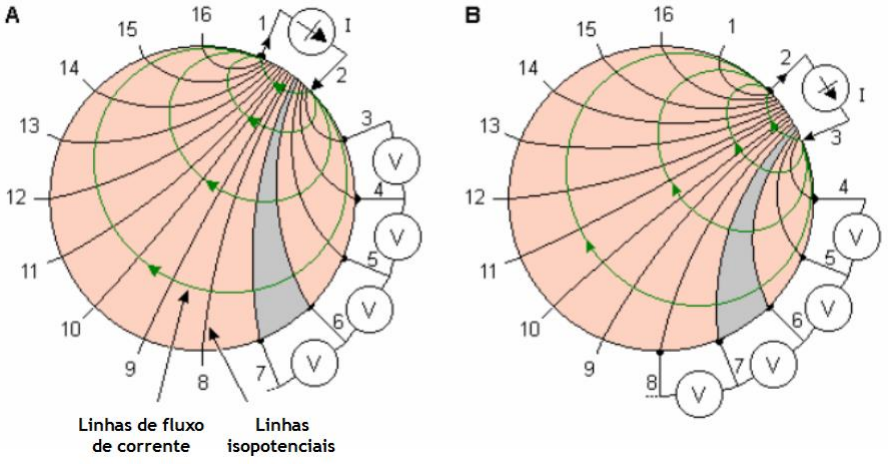


Figura 1.22: Topologia de injeção de corrente e medição de potencial nos diversos eletrodos do tomógrafo. Na ilustração "A", uma corrente elétrica é injetada entre os eletrodos 1 e 2 e as tensões elétricas resultantes são medidas nos demais eletrodos. Este procedimento é refeito (ilustração "B") até que as $N(N-1)/2$ medidas (combinações lineares) sejam realizadas.

A partir das medições realizadas, aplica-se um algoritmo de reconstrução para se obter o perfil de condutividade e permissividade elétrica da área investigada. São necessárias pelo menos $N(N-1)/2$ medidas para se obter os perfis de condutividade e permissividade da tubulação, onde N é o número de eletrodos. Com um conhecimento prévio da condutividade e permissividade dos vários componentes contidos na área investigada, pode-se determinar a composição do material no interior do tubo.

Uma vez que as propriedades elétricas do óleo, da água e do gás são bem conhecidas, pode-se obter a distribuição de fases no interior da tubulação de petróleo, a partir da medição das tensões nos eletrodos e da utilização de um algoritmo de reconstrução de imagens. Dessa forma pode-se estimar o percentual de água, óleo e gás presentes na tubulação.

A grande dificuldade da técnica de impedância elétrica é que ela é muito sensível à ruído e a qualidade da reconstrução deteriora com o aumento da resolução espacial. A precisão da medida é fundamental para se obter uma boa resolução [10].

Na Figura 1.23 é ilustrado o diagrama esquemático de um equipamento de Tomografia por Impedância Elétrica [10]. O equipamento consiste de um gerador de sinal, uma fonte de corrente controlada por tensão, um multiplexador, um demultiplexador, um amplificador de instrumentação, um demodulador, um filtro passa-baixa, e um microcomputador com placa de aquisição de dados.

Esse equipamento usa a técnica do amplificador sensível à fase *(lock-in)* para medir impedâncias [16]. Um sinal de 50 kHz é utilizado para modular a fonte de corrente controlada por tensão. Um dos eletrodos é aterrado (retorno de corrente) e o multiplexador é utilizado para selecionar o eletrodo em que será injetada a corrente. Isso é feito para cada um dos eletrodos. Cada vez que um eletrodo é selecionado para a injeção de corrente, a tensão elétrica é medida em cada um dos N-1 eletrodos através do demultiplexador e do amplificador de instrumentação acoplado. O sinal medido é então demodulado, gerando um sinal constante e um sinal de 100 kHz, que é eliminado pelo filtro passa baixa. A tensão constante, proporcional à impedância, é então medida pelo microcomputador, para ser utilizada pelo algoritmo de reconstrução. Essa técnica permite reduzir o nível de ruído, aumentando a resolução da imagem.

Na construção de um sistema desse tipo é necessário levar em consideração os seguintes aspectos [13]:

- Geometria dos eletrodos;
- Técnica de medição e aquisição de dados;
- Algoritmo de inversão tomográfica.

Em particular, este trabalho está diretamente relacionado à técnica de medição e aquisição de dados. Um sensor utilizado nesta aplicação deve, portanto, ser capaz

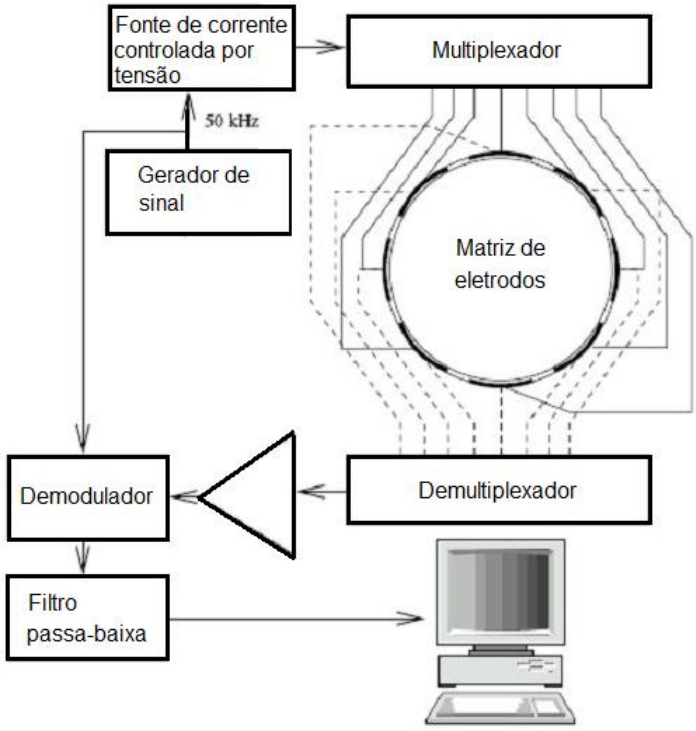


Figura 1.23: Esquema de um equipamento de tomografia por impedância elétrica. Uma corrente elétrica senoidal, produzida por uma fonte de corrente controlada por um gerador de sinal de 50 kHz, é multiplexada e injetada nos diversos eletrodos do tomógrafo. Utilizando a técnica *lock-in*, as medições resultantes são demultiplexadas, amplificadas e demoduladas, a partir do sinal de referência (gerador de sinal). O filtro passa-baixa separa a componente CC do sinal demodulado, proporcional à condutância ou capacitância da seção transversal da matriz de eletrodos. Finalmente, a imagem da seção transversal, obtida a partir de um algoritmo de reconstrução de imagens, é exibida na tela do microcomputador.

de realizar uma medição de tensão bastante precisa. Além disso, é interessante que o sensor seja capaz de processar o sinal medido e transmitir os resultados em formato digital para serem usados em algum algoritmo de reconstrução de imagens.

A técnica *lock-in* permite realizar medições bastante precisas e de alta resolução de sinais relativamente limpos, isto é, essencialmente sem ruído. Além disso, essa técnica é bastante eficiente na recuperação de sinais que estão efetivamente abaixo do nível de ruído, isto é, com baixa relação sinal-ruído. O amplificador sensível à fase também pode ser utilizado na caracterização elétrica de dispositivos, na medição de impedâncias, entre outras aplicações. Portanto, o amplificador sensível à fase é um método de medição versátil, sendo a técnica *lock-in* bastante apropriada para ser usada em Tomografia por Impedância Elétrica.

A maior parte do processamento realizado em Tomografia por Impedância Elétrica

se deve ao algoritmo de reconstrução de imagens. Este algoritmo possui uma complexidade computacional elevada, sendo tipicamente executado por um microcomputador. Uma vez que o sensor inteligente não será responsável pela execução do algoritmo de reconstrução, ele não necessita possuir uma grande capacidade de processamento. Entretanto, é interessante que o dispositivo seja capaz de realizar funções simples de controle, de acordo com algum algoritmo específico do processo no qual o sensor está inserido. Estas funções podem estar relacionadas, por exemplo, com a detecção de algumas anomalias no processo. Nesse sentido, será utilizado o microcontrolador LAMPIÃO, em desenvolvimento no LDN, executando um SOTR embarcado para aquisição de dados e comunicação. A arquitetura do microcontrolador LAMPIÃO e a especificação do SOTR embarcado são apresentados no Apêndice D.

Finalmente, o módulo MARIA será responsável por conectar o sensor inteligente à rede de comunicação. Nesse caso, será utilizado um módulo de comunicação CAN. O protocolo CAN suporta eficientemente controle distribuído em tempo-real com elevado nível de segurança. Por ser um protocolo robusto, vem sendo bastante utilizado em aplicações industriais.

Na Figura 1.24 é apresentado um exemplo de aplicação do sensor inteligente proposto num sistema de tomografia por impedância elétrica.

Considerando a arquitetura proposta no padrão IEEE 1451, o STIM é representado pelo sensor inteligente proposto. O NCAP é implementado num microcomputador que executa o algoritmo de reconstrução de imagens a partir das medições realizadas pelo STIM. A interface entre o STIM e o NCAP é realizada utilizando a rede *CANopen*, conforme proposto no padrão IEEE P1451.6. Além disso, outros dispositivos STIM podem ser inseridos na rede de sensores, enviando ao NCAP medições de outras variáveis como temperatura e pressão, por exemplo, para correção da vazão medida.

1.4.2 Objetivo do Trabalho

O objetivo deste trabalho é a descrição em linguagem VHDL *(VHSIC*[3] *Hardware Description Language)* e a implementação em FPGA *(Field Programmable Gate Array)* de um amplificador sensível à fase digital e de um módulo de comunicação CAN para

[3]VHSIC: *Very High Speed Implementation Circuit*

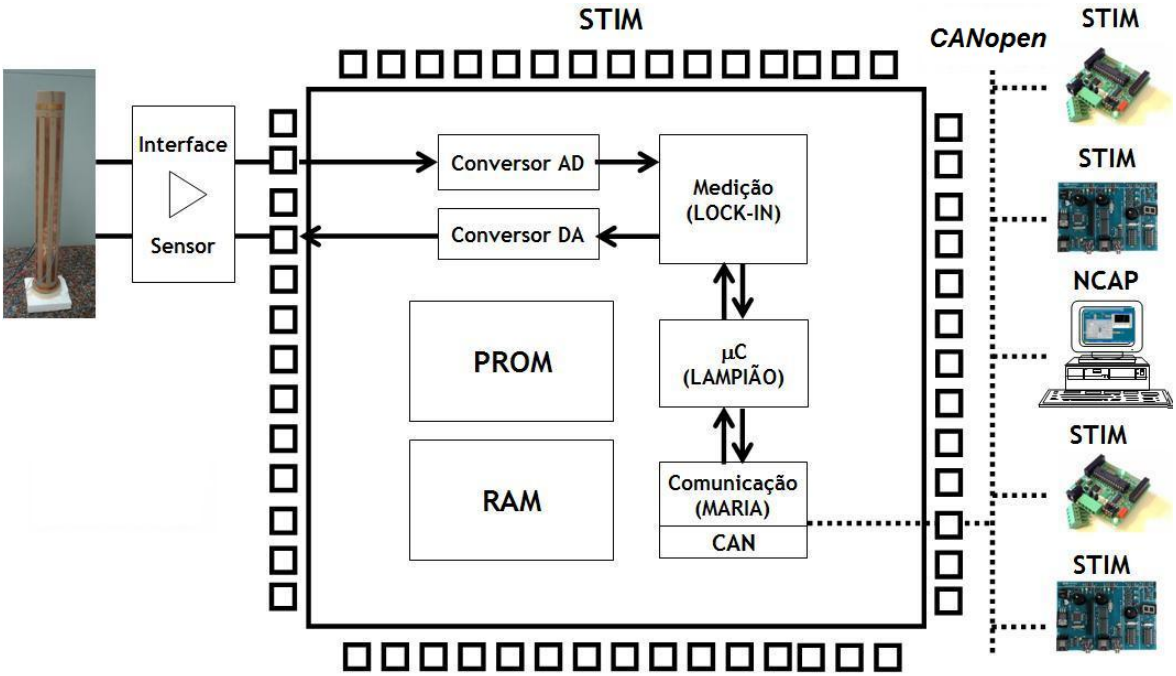


Figura 1.24: Sensor inteligente aplicado a um sistema de tomografia por impedância elétrica. A interface entre o STIM e o NCAP é realizada utilizando a rede *CANopen*, conforme definido no padrão IEEE 1451.6.

utilização no sensor inteligente proposto.

Uma vez que o amplificador sensível à fase e o módulo de comunicação CAN estejam devidamente validados, eles podem ser integrados ao microcontrolador LAMPIÃO, ao módulo de acesso à rede MARIA, às memórias PROM e RAM e aos circuitos de aquisição de dados e geração de sinal, também em desenvolvimento no LDN, numa mesma pastilha de silício. Neste caso, tem-se um sensor inteligente integrado.

1.5 Organização do Texto

Esta Dissertação de Mestrado está dividida em cinco capítulos:

- Capítulo 1, contendo esta introdução;
- Capítulo 2, no qual é apresentada a metodologia utilizada neste trabalho;
- Capítulo 3, no qual é discutida a teoria do amplificador sensível à fase e apresentados os resultados obtidos na implementação deste dispositivo, utilizando, num primeiro momento, um microcomputador com placa de aquisição de dados e MATLAB e, em seguida, desenvolvendo o circuito em FPGA;

- Capítulo 4, no qual são apresentados o protocolo de comunicação CAN, a implementação de uma rede CAN utilizando placas SBC28PC e microcontroladores PIC 18F258, os resultados da implementação de um módulo CAN em FPGA e da integração deste módulo ao amplificador sensível à fase;

- Capítulo 5, no qual são apresentadas as conclusões deste trabalho e as próximas etapas do projeto do sensor inteligente integrado.

  Neste texto também foram incluídos cinco apêndices:

- Apêndice A, no qual são fornecidos os códigos em linguagem VHDL dos circuitos desenvolvidos em FPGA;

- Apêndice B, no qual são fornecidos os códigos nas linguagens C e MATLAB do amplificador *lock-in* utilizando microcomputador e placa de aquisição de dados;

- Apêndice C, no qual são fornecidos os códigos em linguagem ASSEMBLY dos módulos CAN implementados com microcontroladores PIC 18F258;

- Apêndice D, no qual é apresentada uma discussão teórica sobre um Sistema Operacional de Tempo Real embarcado para gerenciamento de aquisição de dados e comunicação com o microcontrolador LAMPIÃO e módulo de acesso à rede MARIA;

- Apêndice E, no qual são apresentados os artigos publicados a partir deste trabalho.

Capítulo 2

Metodologia

Neste capítulo será apresentada a metodologia utilizada neste trabalho. Serão abordadas todas as etapas utilizadas no desenvolvimento do amplificador sensível à fase digital *(lock-in)* e do módulo de comunicação CAN *(Controller Area Network)*, desde a implementação de protótipos utilizando placa de aquisição de dados e microcontroladores PIC até a implementação dos circuitos em FPGA *(Field Programmable Gate Array)*.

2.1 Amplificador Sensível à Fase

A implementação do amplificador sensível à fase digital foi realizada através das seguintes etapas:

- Estudo da técnica *lock-in* para medição de sinais;
- Implementação de um protótipo para avaliação da técnica *lock-in* utilizando um microcomputador com placa de aquisição de dados;
- Implementação do circuito final em FPGA.

2.2 Módulo de Comunicação CAN

A implementação do módulo de comunicação CAN foi realizada através das seguintes etapas:

- Estudo do protocolo de comunicação CAN;
- Implementação de uma rede CAN utilizando microcontroladores PIC para avaliação do protocolo de comunicação;
- Implementação do circuito final em FPGA.

2.3 Etapas de Prototipação

Nesta seção serão apresentadas as principais etapas da implementação de um amplificador sensível à fase digital utilizando um microcomputador com placa de aquisição de dados e de uma rede CAN utilizando placa de desenvolvimento comercial e microcontroladores PIC.

2.3.1 Amplificador Sensível à Fase Digital utilizando Microcomputador e Placa de Aquisição de Dados

No sentido de avaliar a técnica *lock-in*, foi implementado um amplificador sensível à fase digital utilizando inicialmente um microcomputador e uma placa de aquisição de dados. O desenvolvimento deste protótipo, por sua vez, foi segregado em três tarefas principais:

- Geração do sinal de teste (tensão de excitação);
- Aquisição do sinal de entrada do amplificador (resposta à excitação);
- Execução do algoritmo utilizado na descrição da técnica *lock-in*.

Para geração do sinal de teste (tensão de excitação) e aquisição do sinal de entrada do amplificador (resposta à excitação), foi utilizada a placa de aquisição de dados DAS-20, desenvolvida pela *Keithley Instruments* e ilustrada na Figura 2.1. O controle da placa é realizado a partir de um microcomputador IBM/PC através do barramento ISA e de um *driver* de comunicação, desenvolvido por *Tony L. Keiser*. Este *driver* é constituído por uma biblioteca de funções que implementam diversos comandos da placa DAS-20 e são chamadas no código de um programa de aquisição de dados escrito em linguagem C utilizando o compilador Dev-C++ 4.9.8.0. Este compilador,

ilustrado na Figura 2.2, foi obtido gratuitamente a partir da página na Internet de seu desenvolvedor, o *Bloodshed Software* [17].

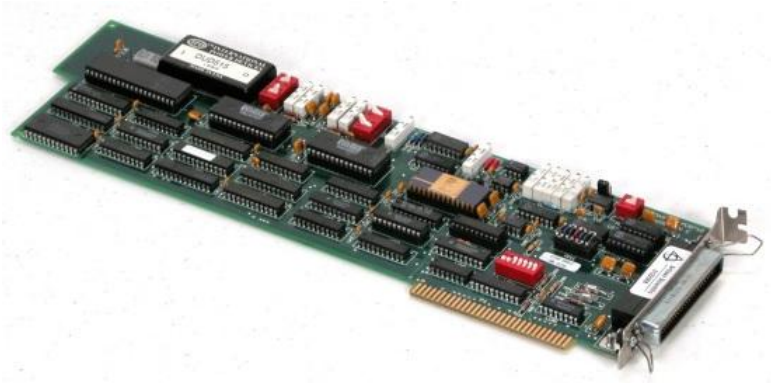


Figura 2.1: Ilustração da placa de aquisição de dados DAS-20.

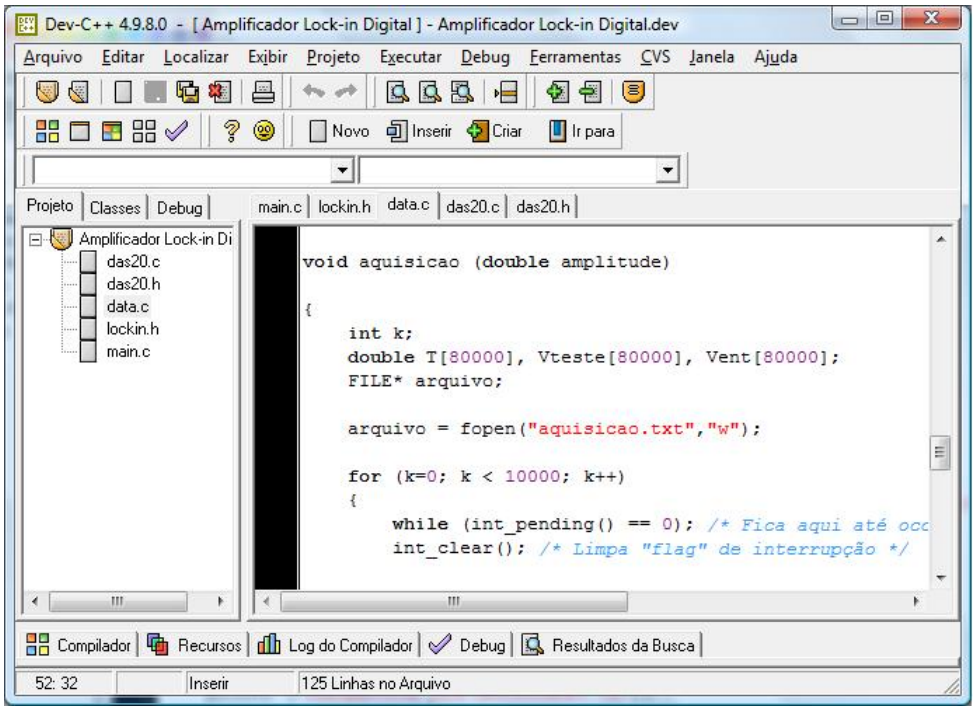


Figura 2.2: Ilustração do ambiente de programação em linguagem C.

O programa de aquisição de dados gera um arquivo de texto contendo os dados adquiridos (tensões medidas na entrada do amplificador). Este arquivo, por sua vez, é acessado por um programa implementado no MATLAB, *software* desenvolvido pela *MathWorks* e ilustrado na Figura 2.3. Este programa executa o algoritmo utilizado na descrição da técnica *lock-in* e exibe os resultados dos cálculos realizados.

Os materiais utilizados na implementação deste protótipo estão resumidos a seguir:

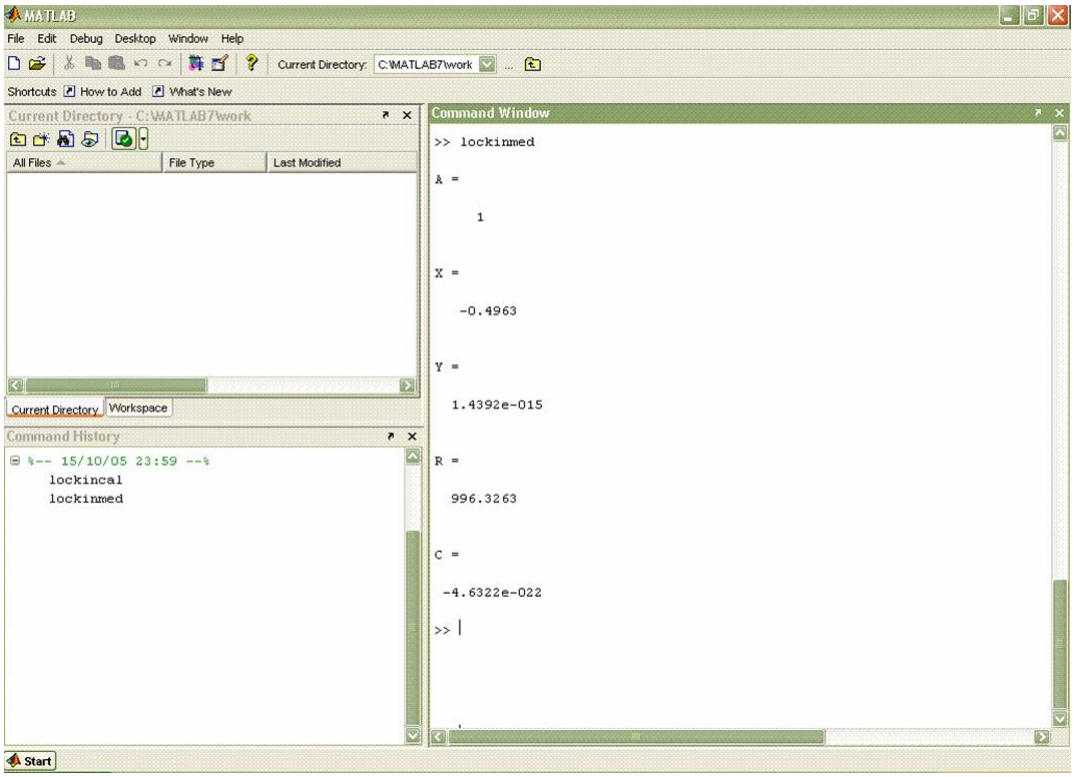


Figura 2.3: Ilustração do ambiente MATLAB.

- Microcomputador IBM/PC com barramento ISA;
- Placa de Aquisição de Dados DAS-20, desenvolvida pela *Keithley Instruments*;
- Compilador Dev-C++ 4.9.8.0, desenvolvido pela *Bloodshed Software*;
- Biblioteca de funções desenvolvida em linguagem C por *Tony L. Keiser* para acesso aos comandos da placa DAS-20;
- *Software* MATLAB, desenvolvido pela *MathWorks*.

A implementação deste protótipo e os resultados obtidos são descritos detalhadamente no Capítulo 3.

2.3.2 Rede de Comunicação CAN utilizando Placa de Desenvolvimento Comercial e Microcontroladores PIC

Com o objetivo de avaliar o protocolo de comunicação CAN, foi implementada uma rede de comunicação CAN utilizando dispositivos comerciais. Os materiais utilizados estão descritos a seguir:

- Placa de desenvolvimento SBC28PC, desenvolvida pela *Modtronix Engineering*;
- Microcontrolador PIC 18F258, desenvolvido pela *Microchip*;
- Transceptor CAN MCP2551, desenvolvido pela *Microchip*;
- Programador e Depurador ICD3, desenvolvido pela *Microchip*;
- *Software* MPLAB 8.36, desenvolvido pela *Microchip*;
- Microcomputador IBM/PC com porta serial;
- *Software* Terminal v.19b;
- Osciloscópio.

A SBC28PC, ilustrada na Figura 2.4, é uma placa comercial de baixo custo, desenvolvida para microcontroladores PIC de 28 pinos, disponibilizando tanto uma porta de comunicação serial RS-232 quanto uma interface CAN.

Figura 2.4: Ilustração da placa de desenvolvimento SBC28PC.

O microcontrolador PIC 18F258 possui uma UART e um módulo de comunicação CAN integrados. O módulo transceptor CAN MCP2551 é responsável pela conversão dos sinais de 0V e 5V do microcontrolador para os níveis de tensão padronizados pelo protocolo CAN.

Para programação dos microcontroladores PIC foi utilizada a ferramenta de projeto MPLAB 8.36, desenvolvida pela *Microchip* e ilustrada na Figura 2.5. Para

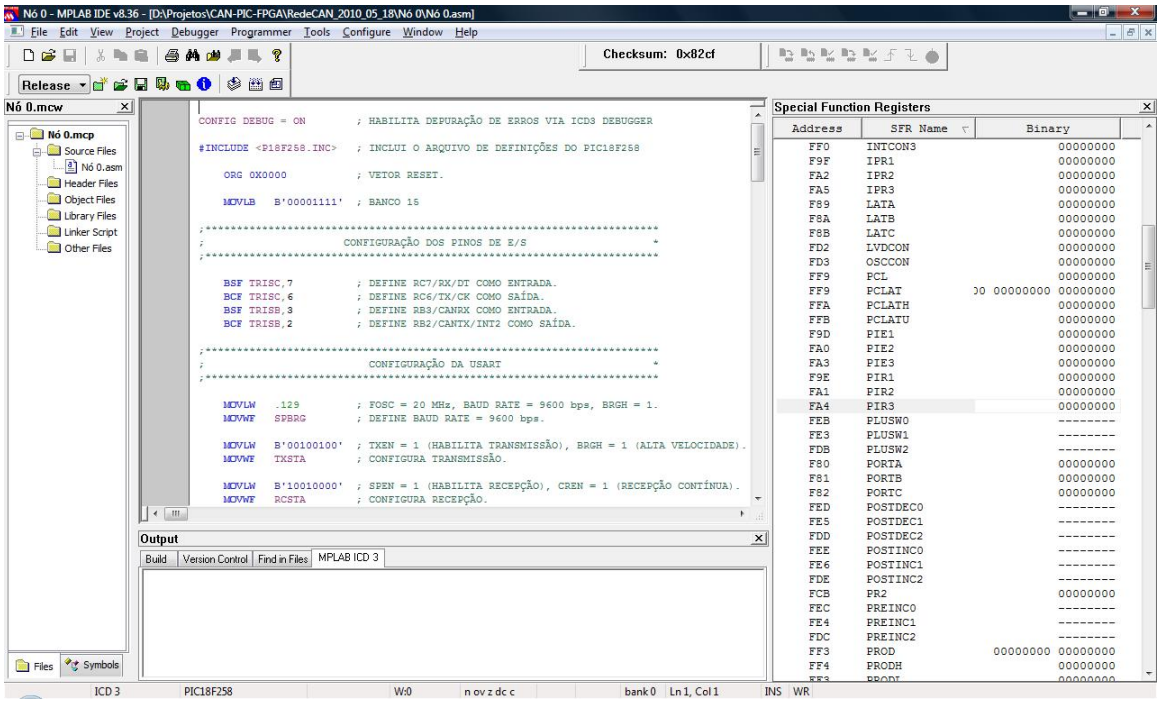


Figura 2.5: Ilustração da ferramenta de projeto MPLAB.

Figura 2.6: Ilustração do programador e depurador ICD3.

gravação e depuração do programa foi utilizado o programador e depurador ICD3, também da *Microchip* e ilustrado na Figura 2.6.

Neste trabalho foi construída uma rede CAN contendo dois nós, implementados com a placa SBC28PC, o microcontrolador PIC 18F258 e o transceptor CAN MCP2551. Um desses nós, além de enviar e receber mensagens na rede CAN, envia as mensagens trafegadas no barramento CAN para o microcomputador através da porta serial. As mensagens trafegadas na rede CAN são então exibidas na tela do microcomputador, a partir de um programa terminal, ilustrado na Figura 2.7. Este programa foi obtido gratuitamente a partir da página na Internet de seu desenvolvedor [18].

Finalmente, para uma avaliação mais detalhada do protocolo CAN, cada campo da mensagem trafegada é analisado com auxílio do osciloscópio.

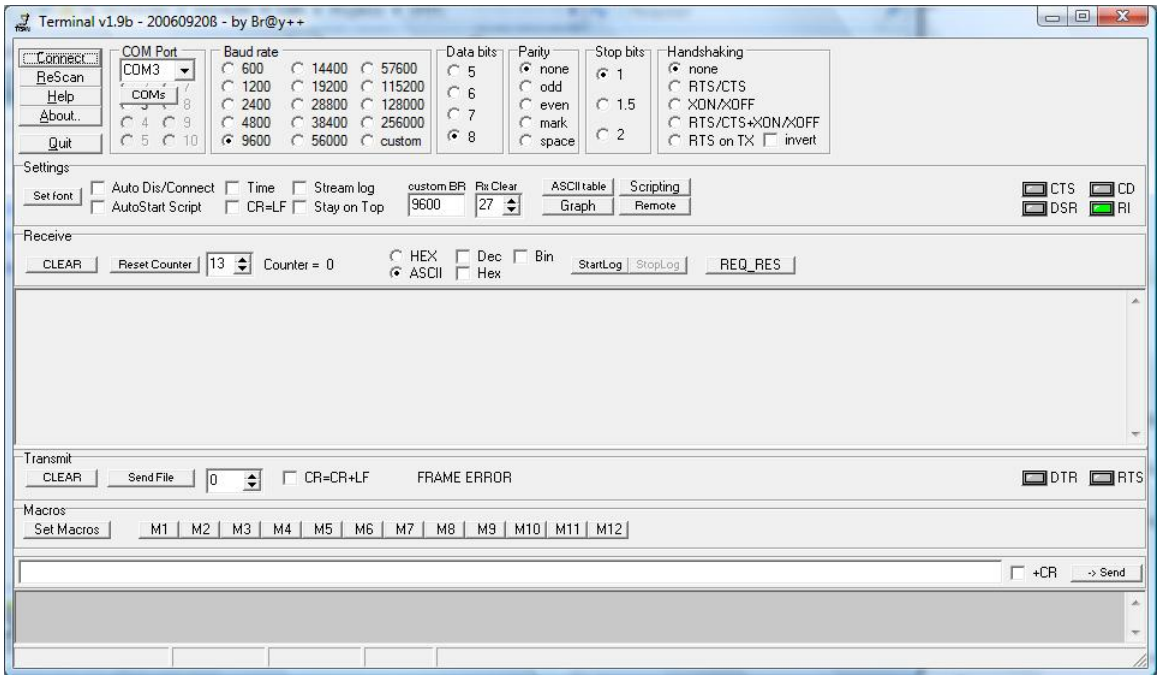


Figura 2.7: Ilustração do programa Terminal.

A implementação desta rede CAN e os resultados obtidos são descritos detalhadamente no Capítulo 4.

2.4 Etapas de Implementação em FPGA

Após entender e avaliar a técnica *lock-in* e o protocolo CAN através dos protótipos desenvolvidos, foi realizada a implementação em FPGA de um amplificador sensível à fase digital e de um módulo de comunicação CAN. Estes dois dispositivos foram desenvolvidos e testados individualmente. Em seguida, os dois blocos foram integrados na mesma FPGA.

Na próxima seção será revisado, de forma simplificada, o fluxo genérico de projeto de circuitos integrados, desde a especificação do circuito até a fabricação do *chip*. Em seguida, serão discutidas as principais características de uma FPGA e da descrição de circuitos digitais em linguagem VHDL *(VHSIC*[1] *Hardware Description Language)*.

Finalmente, será apresentado o fluxo de projeto de circuitos digitais em FPGA descritos na linguagem VHDL, utilizado na implementação final do amplificador sensível à fase e do módulo de comunicação CAN.

[1] VHSIC: *Very High Speed Implementation Circuit*

2.5 Projeto de Circuitos Integrados

O projeto de circuitos integrados envolve uma série de etapas que vão desde a especificação do sistema até a fabricação e a caracterização do *chip*. Essas etapas são apresentadas, de maneira geral e simplificada, no fluxograma da Figura 2.8.

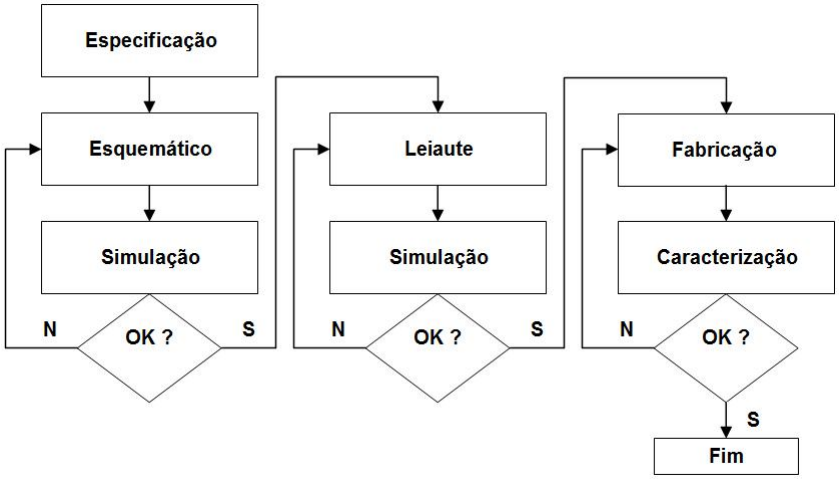


Figura 2.8: Etapas de projeto de circuitos integrados.

A primeira etapa é a especificação do circuito. Nessa etapa são definidas as funções do circuito, suas entradas e saídas, sua topologia, a tecnologia a ser utilizada, entre outras características.

Uma vez definidos os requisitos do sistema, o circuito é desenvolvido de acordo com a topologia escolhida utilizando-se, por exemplo, captura esquemática ou uma linguagem de descrição de *hardware*. Em seguida, o circuito é simulado com o objetivo de verificar se a sua funcionalidade está adequada, isto é, se o seu comportamento está de acordo com o esperado. Nessa etapa, entretanto, ainda não são considerados os elementos parasitas (resistências, capacitâncias, indutâncias, etc.) presentes no circuito.

Caso os resultados obtidos na simulação comportamental estejam de acordo com o especificado, inicia-se o desenvolvimento do leiaute. Caso contrário, refaz-se o esquemático ou a descrição do *hardware*, corrigindo-se os eventuais erros. Nesse ponto podem ser necessárias ainda algumas modificações na especificação do circuito.

Na etapa de desenvolvimento do leiaute, o circuito é implementado ao nível de transistores, respeitando-se as regras de projeto definidas na tecnologia escolhida.

Após a confecção do leiaute, o circuito é novamente simulado, extraindo-se as resistências e capacitâncias parasitas e os modelos dos dispositivos utilizados.

Depois de avaliado o desempenho do circuito, a partir da simulação do leiaute, algumas modificações ainda podem ser necessárias no leiaute, no esquemático ou até mesmo na especificação. Quando os resultados obtidos via simulação forem satisfatórios, o circuito integrado é enviado para a fabricação. Nessa etapa são confeccionadas as máscaras, as quais são utilizadas durante o processamento do silício para, em seguida, formar-se o *chip*.

Finalmente, o *chip* é testado e encapsulado, passando ainda pelos processos de controle de qualidade. Se durante a caracterização do dispositivo for verificado algum problema em seu funcionamento, a etapa de fabricação e, eventualmente, as etapas anteriores, devem ser revistas.

O projeto de circuitos integrados pode ser segmentado de acordo com os tipos de sinais a serem manipulados pelo circuito em desenvolvimento. Uma forma de segmentação é proposta pelo autor deste trabalho na Figura 2.9.

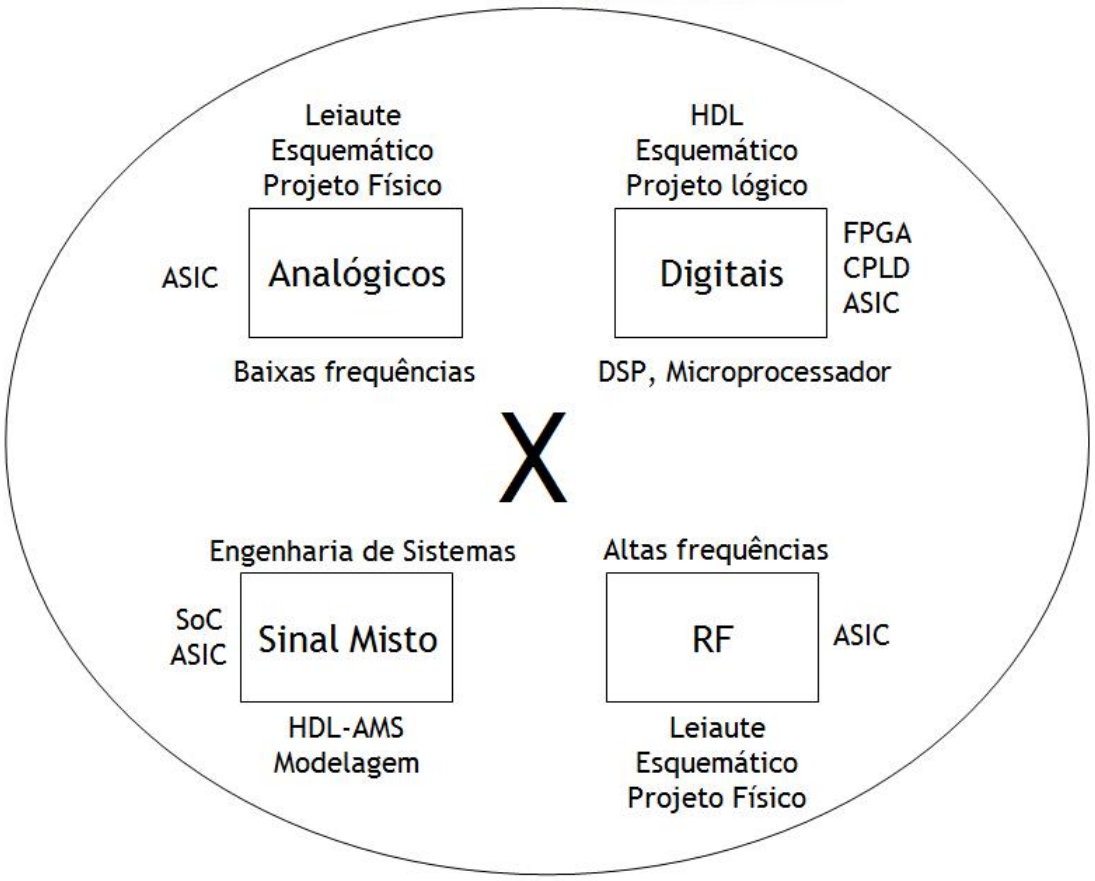


Figura 2.9: Segmentação proposta da área de projeto de circuitos integrados.

Em linhas gerais, a metodologia de projeto de circuitos integrados segue o mesmo fluxo de desenvolvimento discutido anteriormente, independente do segmento. Entretanto, cada segmento de projeto apresenta suas particularidades na forma de desenvolvimento, ferramentas de projeto, simulação e modelagem, construção física, caracterização e testes, etc. Estas particularidades são discutidas a seguir:

- Projeto de circuitos integrados analógicos: engloba tipicamente o projeto em baixas frequências de circuitos como amplificadores operacionais, referências de tensão, espelhos de corrente, etc. O projeto é realizado a nível físico, isto é, a nível de transistores. Utiliza geralmente ferramentas de projeto baseadas em captura esquemática. O leiaute do circuito é realizado manualmente. As ferramentas de simulação são baseadas em modelos SPICE. Em geral, são implementados em circuitos integrados para aplicações específicas, denominados ASIC *(Application Specified Integrated Circuits)*;

- Projeto de circuitos integrados digitais: engloba o projeto de circuitos baseados em microprocessadores, DSPs *(Digital Signal Processors)*, FPGAs *(Field Programmable Gate Arrays)*, etc. O projeto é realizado a nível lógico, utilizando captura esquemática ou linguagem de descrição de *hardware*. O leiaute do circuito é gerado automaticamente pela ferramenta de síntese e implementação. As ferramentas de simulação se baseiam em modelos temporais dos dispositivos lógicos, levando em conta apenas os parasitas das interconexões, reduzindo o tempo de simulação.

- Projeto de circuitos integrados de sinal misto: engloba o projeto de circuitos analógicos e digitais no mesmo *chip*, utilizando conversores AD e DA. Utiliza ferramentas de modelagem baseadas em linguagens de descrição de circuitos analógicos e de sinal misto, de forma a reduzir o tempo de simulação. Entretanto, o leiaute do circuito analógico continua sendo feito manualmente;

- Projeto de circuitos integrados de RF: engloba tipicamente o projeto de circuitos de transmissão e recepção em rádio-frequência, moduladores, demoduladores, amplificadores de RF, amplificadores de potência, etc. Utiliza metodologia de projeto semelhante aos circuitos analógicos. Entretanto, utiliza ferramentas de

simulação mais poderosas, capazes de realizar simulações de campos eletromagnéticos e de sinais em altas frequências, englobando modelos mais refinados de parasitas.

Na implementação final do amplificador sensível à fase e do módulo de comunicação CAN foi utilizada a metodologia de projeto de circuitos digitais em FPGA. Esta metodologia é discutida em mais detalhes nas próximas seções.

2.6 FPGA - *Field Programmable Gate Array*

A FPGA é uma matriz de portas programável em campo. Trata-se de um módulo programável capaz de implementar sistemas digitais complexos, sendo utilizado tanto na prototipação de sistemas digitais que serão implementados em ASIC quanto como dispositivo final. Na Figura 2.10 são apresentadas FPGAs dos principais fabricantes: Actel, Altera e Xilinx.

Figura 2.10: FPGAs dos principais fabricantes: Actel, Altera e Xilinx.

Neste trabalho foi utilizada uma FPGA *Xilinx XC3S500E -5 FG320*. A arquitetura básica de uma FPGA *Xilinx* [19, 20] é apresentada na Figura 2.11.

A família *Spartan-3E* da *Xilinx* possui 5 elementos fundamentais [20]:

- Blocos lógicos configuráveis (CLB - *Configurable Logic Blocks*);
- Blocos de entrada e saída (IOB - *Input/Output Blocks*);
- Blocos de memória RAM;
- Blocos multiplicadores dedicados;
- Blocos de gerenciamento de relógio digital (DCM - *Digital Clock Manager*).

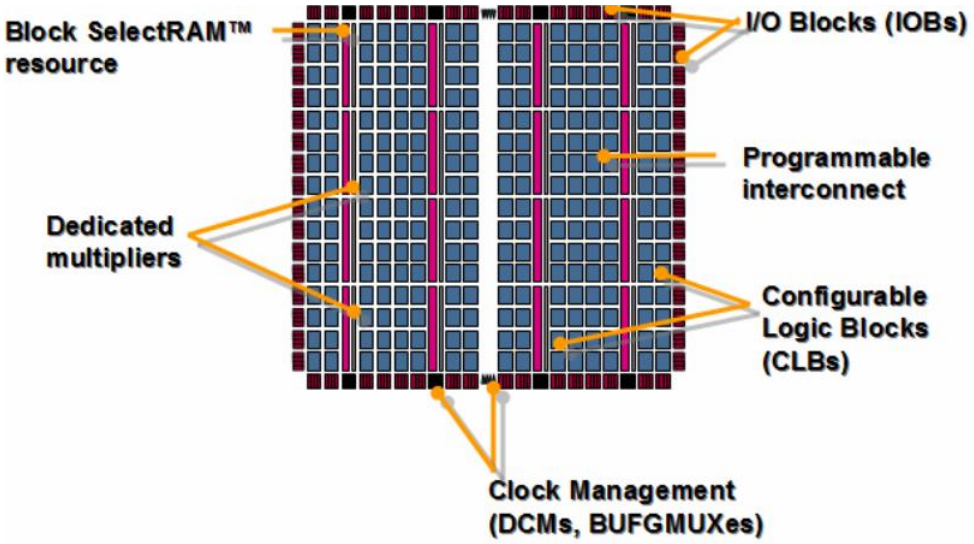


Figura 2.11: Arquitetura básica de uma FPGA *Xilinx*.

Os CLBs contêm tabelas de pesquisa (*Look up Tables* - LUTs) para implementar elementos lógicos e de armazenamento (portas lógicas, *flip-flops* e *latches*). Constituem os principais recursos lógicos para a implementação de circuitos combinacionais e sequenciais.

Cada CLB é composto por quatro *slices* interconectados. Os *slices* são agrupados em pares. Cada par é organizado como uma coluna com um canal de *carry* independente. Um *slice* possui os seguintes elementos: geradores de função lógica, elementos de armazenamento, multiplexadores, lógica de *carry* e portas aritméticas. O arranjo de *slices* num CLB e o diagrama simplificado de um *slice* de uma FPGA *Xilinx Spartan II* [21] são apresentados nas Figuras 2.12 e 2.13.

Cada *slice* numa FPGA *Xilinx Spartan II* contém duas células lógicas (LC - *Logic Cells*). Por sua vez, cada LC possui os seguintes componentes:

- Um gerador de funções de quatro entradas, implementado como uma LUT *(Look Up Table)*;
- Uma lógica de controle e carry *(Carry and Control Logic)*;
- Um elemento de armazenamento.

Além dos sinais de *Clock* e *Clock Enable*, cada *slice* possui sinais de *set* e *reset* síncronos (SR e BY). Alternativamente, estes sinais podem ser configurados para operar de maneira assíncrona.

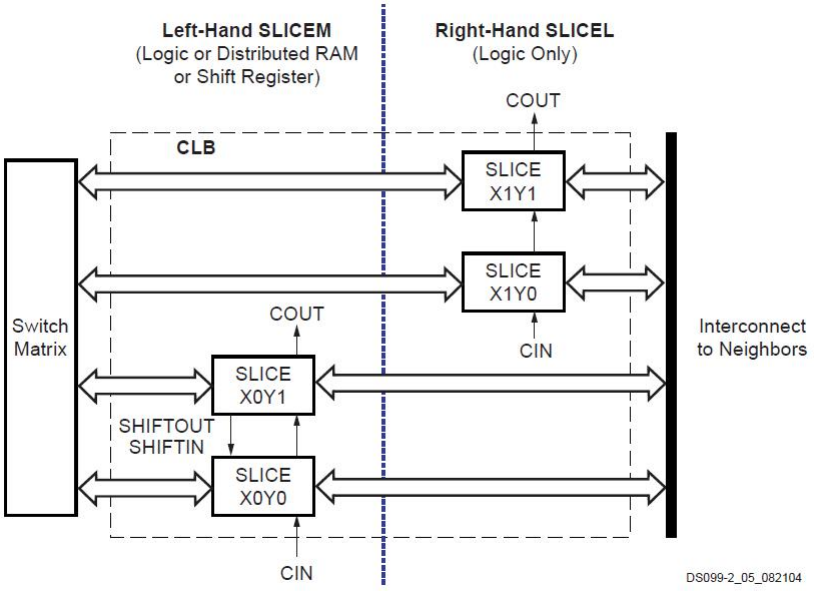


Figura 2.12: Arranjo de *slices* num CLB.

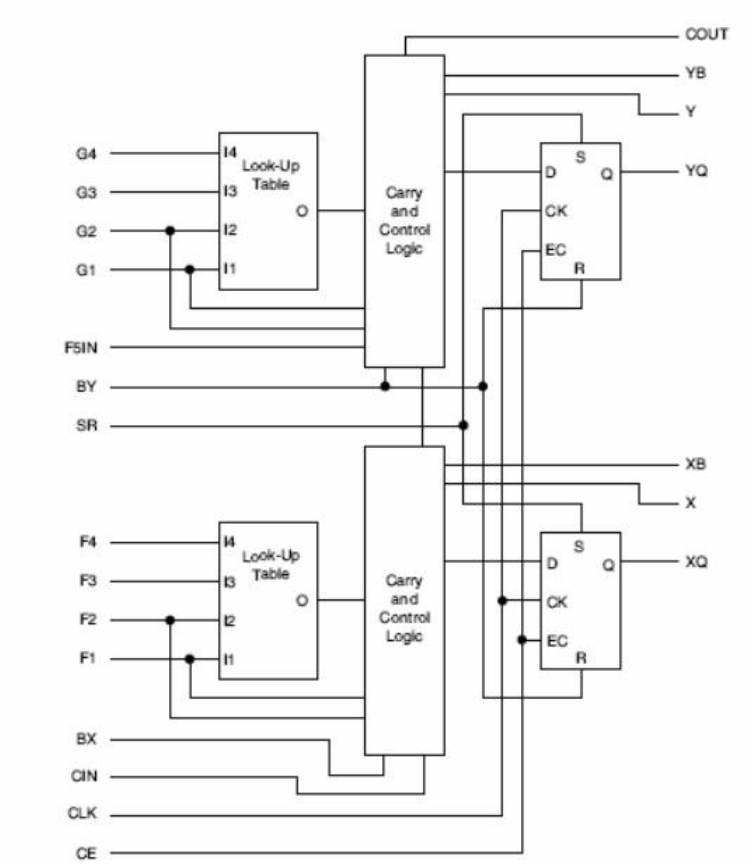


Figura 2.13: Arquitetura de um *slice* numa FPGA *Xilinx Spartan II*.

Cada *slice* possui ainda um multiplexador, denominado F5 (não apresentado na Figura 2.13). Este multiplexador combina as saídas do gerador de funções. Esta combinação pode ser utilizada para implementar qualquer função de 5 entradas, um multiplexador 4:1 ou funções selecionadas de até 9 entradas.

Similarmente, outro multiplexador, denominado F6, combina as saídas de todos os quatro geradores de função num CLB, selecionando uma das saídas dos multiplexadores F5. Isto permite a implementação de qualquer função de 6 entradas, um multiplexador 8:1 ou funções selecionadas de até 19 entradas.

Os IOBs controlam o fluxo de dados entre os pinos de entrada e saída e a lógica interna do dispositivo. Cada IOB suporta fluxo de dados bidirecional e operação 3-estados.

Os blocos de memória RAM permitem o armazenamento de dados na forma de blocos duais de 18-kbit. Os blocos multiplicadores calculam o produto de dois números binários de 18 bits. Finalmente, os blocos de gerenciamento de relógio digital são utilizados na calibração de sinais de relógio.

Normalmente a FPGA é configurada a partir de um ambiente de desenvolvimento, utilizando uma linguagem de descrição de *hardware* (HDL - *Hardware Description Language*). Neste trabalho foi escolhida a linguagem VHDL.

2.7 VHDL - *VHSIC Hardware Description Language*

VHDL *(VHSIC Hardware Description Language)* é uma linguagem de descrição de *hardware* utilizada na simulação e na síntese de circuitos digitais, possibilitando o intercâmbio de informações entre fabricantes, fornecedores de sistemas e empresas de projetos. Surgiu a partir da necessidade de uma ferramenta de projeto e documentação padrão para o projeto VHSIC *(Very High Speed Integrated Circuit)* da DARPA *(Defense Advanced Research Projects Agency)* [22].

VHDL suporta projetos com múltiplos níveis de hierarquia, tornando mais fácil o desenvolvimento de projetos mais complexos utilizando a metodologia *top-down*. Também suporta diversos níveis de abstração. Além disso, VHDL permite a definição

de subprogramas, funções, bibliotecas, pacotes, novos tipos de dados, etc. [22].

Diferentemente da maioria das linguagens de programação, os comandos em VHDL são executados concorrentemente, exceto em regiões específicas do código. Essa é uma das características mais importantes da linguagem, uma vez que os elementos dos sistemas digitais trabalham em paralelo, realizando suas funções conjuntamente.

Vale salientar que a linguagem VHDL não foi concebida inicialmente para síntese de circuitos. O objetivo inicial era a modelagem de circuitos digitais e a documentação de projetos. Consequentemente, nem todas as construções da linguagem são suportadas por uma ferramenta de síntese, responsável por compilar o código VHDL e gerar o circuito digital [22].

Por exemplo, um registrador com dois sinais de relógio pode ser facilmente modelado e seu comportamento simulado em VHDL. Entretanto, esse circuito não pode ser sintetizado, devido à inexistência de um dispositivo físico desse tipo.

Outro exemplo simples é a impossibilidade da síntese direta de um multiplicador de dois números reais, embora um dispositivo desse tipo possa ser modelado e simulado facilmente utilizando VHDL.

A descrição de um circuito em VHDL pode ser realizada seguindo três estilos: comportamental, RTL *(Register Transfer Level)* e estrutural.

Na descrição comportamental, o circuito é modelado a partir das especificações do comportamento do sistema, atingindo um maior nível de abstração. Uma descrição neste nível de abstração é geralmente mais rápida de ser desenvolvida, podendo ser utilizada para modelagem e simulações de sistemas digitais. Entretanto, em geral, um circuito digital descrito de forma comportamental leva à um maior número de recursos lógicos utilizados (e, consequentemente, uma maior área no *chip*) e a um menor desempenho (caracterizado por uma menor frequência de operação máxima). Além disso, nem sempre um circuito digital descrito de forma comportamental poderá ser sintetizado.

Já na descrição RTL, o circuito é descrito de maneira a separar o fluxo de dados (registradores, somadores, contadores, etc.) do fluxo de controle (máquina de estados). Consequentemente, a descrição RTL apresenta um nível de descrição mais próximo do circuito implementado. Apesar de apresentar um tempo de desenvolvimento maior que na descrição comportamental, este nível de abstração é mais fácil

de ser sintetizado pela ferramenta de síntese, apresentando melhores resultados na implementação (menor utilização de recursos lógicos e maior frequência de operação máxima).

Finalmente, na descrição estrutural, o sistema é composto pela interconexão de outros circuitos que executam funções mais básicas (por exemplo, portas lógicas, multiplexadores, somadores e registradores). Estes circuitos (componentes) interligados formam a estrutura do sistema global. Este estilo, portanto, possui um menor nível de abstração. Um comparativo entre os estilos de descrição de *hardware* são ilustrados na Tabela 2.1.

Tabela 2.1: Comparativo entre os estilos de descrição de *hardware*.

Domínio	Descreve	Nível
Comportamental	Funcionalidade	Algoritmo
RTL	Fluxo de Dados e Fluxo de Controle	Unidades Funcionais e Máquina de Estados
Estrutural	Arquitetura	Interconexão entre os módulos

2.8 Fluxo de Projeto de Circuitos Digitais em FPGA utilizando VHDL

Conforme mencionado nas seções anteriores, a implementação final do amplificador sensível à fase e do módulo de comunicação CAN foi realizada de acordo com o fluxo de projeto de circuitos digitais em FPGA utilizando VHDL, que será descrito a seguir.

O processo de síntese de circuitos digitais em VHDL pode ser dividido em sete etapas, conforme ilustrado na Figura 2.14. Cada etapa é discutida a seguir, através da implementação de uma porta lógica básica: a porta NAND (não-e).

2.8.1 Especificação

Nessa etapa é realizada a formulação do problema, ou seja, a especificação do projeto a ser desenvolvido. Nesse sentido, a função do sistema, suas entradas e saídas são definidas.

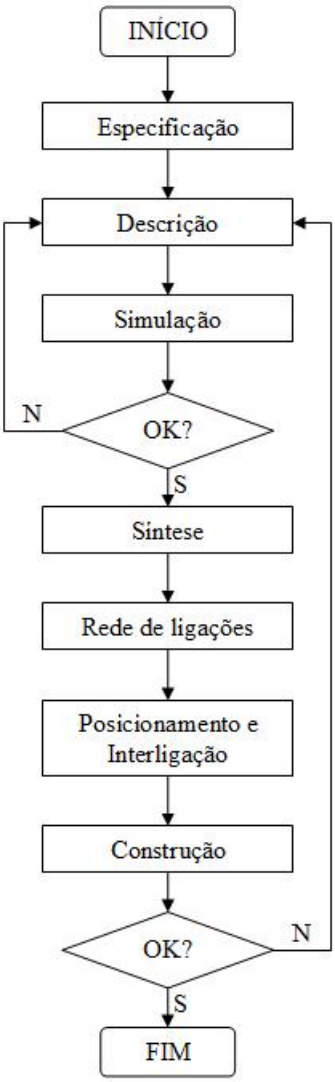


Figura 2.14: Etapas gerais de um processo de síntese em FPGA utilizando VHDL.

- Definição do projeto: descrever em VHDL uma porta NAND;
- Entradas: X e Y;
- Saída: Z
- Função lógica: $Z = XnandY$;

2.8.2 Descrição

Nessa etapa, o projetista descreve o funcionamento do circuito, de acordo com a sua especificação. A descrição é composta por uma entidade de projeto (ou por um

conjunto de entidades interligadas na forma de componentes) que pode representar desde uma simples porta lógica até um sistema digital completo.

A entidade de projeto é constituída por uma declaração da entidade e por uma arquitetura, conforme ilustrado na Figura 2.15. A declaração da entidade define as portas de entrada e saída, enquanto a arquitetura descreve como o circuito funciona (relações entre as portas).

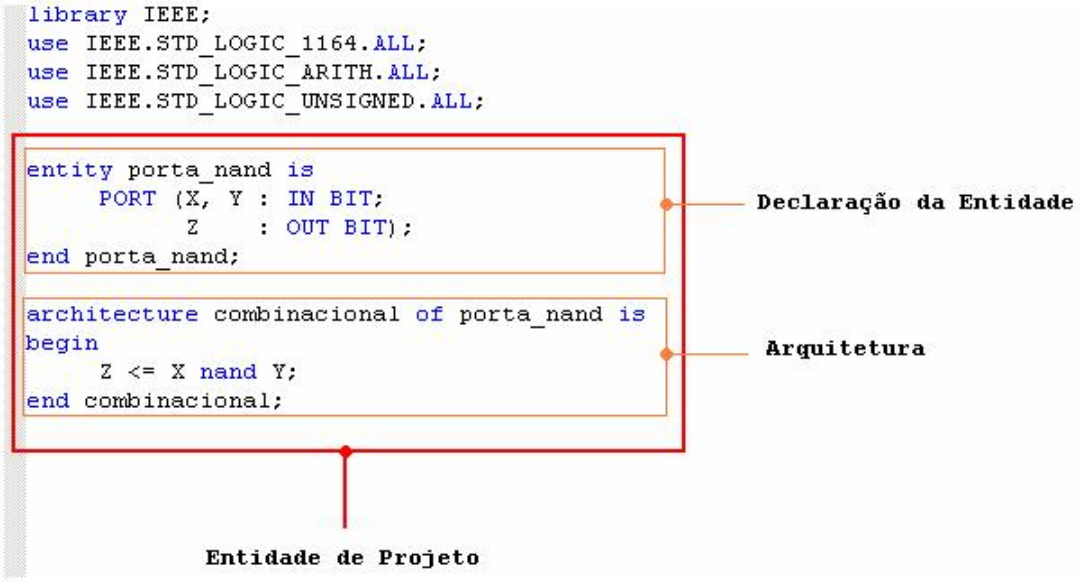


Figura 2.15: Descrição VHDL da porta NAND: entidade e arquitetura.

2.8.3 Simulação

Após a descrição do circuito e a compilação do código é realizada a simulação comportamental. Nessa etapa, estímulos de teste são gerados e o comportamento do circuito é verificado. Se os resultados da simulação forem satisfatórios o circuito está pronto para ser sintetizado. Caso contrário, a descrição deve ser revista.

Na Figura 2.16 é apresentada as formas de onda obtidas na simulação da descrição VHDL de uma porta NAND.

2.8.4 Síntese

Nessa etapa são executadas a inferência e a interligação das estruturas necessárias à geração do circuito descrito. No processo de síntese dois fatores são levados em

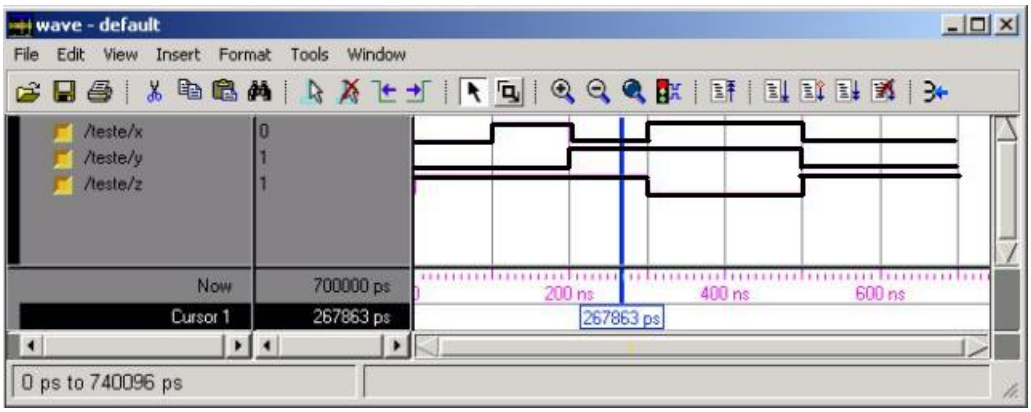


Figura 2.16: Resultados da simulação comportamental da porta NAND.

consideração: as primitivas disponíveis na ferramenta de síntese e os recursos presentes na FPGA escolhida.

Inicialmente, a ferramenta de síntese gera um circuito no nível RTL utilizando dispositivos básicos disponíveis na ferramenta, como portas lógicas e registradores. Em seguida, um novo circuito é criado contendo apenas dispositivos disponíveis na FPGA escolhida.

O resultado da síntese da porta NAND é ilustrado na Figura 2.17.

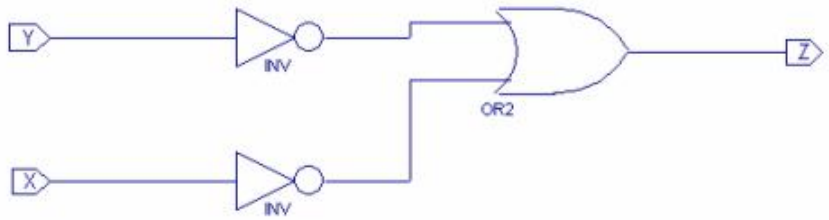


Figura 2.17: Circuito sintetizado a partir da descrição VHDL da porta NAND.

2.8.5 Rede de Ligações, Posicionamento, Interligação e Construção

Após a etapa de síntese, é definida a rede de ligações a ser implementada. Nesse momento, são assinalados os pinos da FPGA a serem utilizados. Em seguida, a ferramenta de síntese posiciona e interliga os componentes do circuito. Finalmente, o circuito é implementado, a partir da gravação dos arquivos de configuração da FPGA.

Nas Figuras 2.18 e 2.19 são ilustradas algumas janelas da ferramenta de posicionamento e interligação, nas quais os pinos da FPGA são assinalados, os componentes

posicionados e interligados.

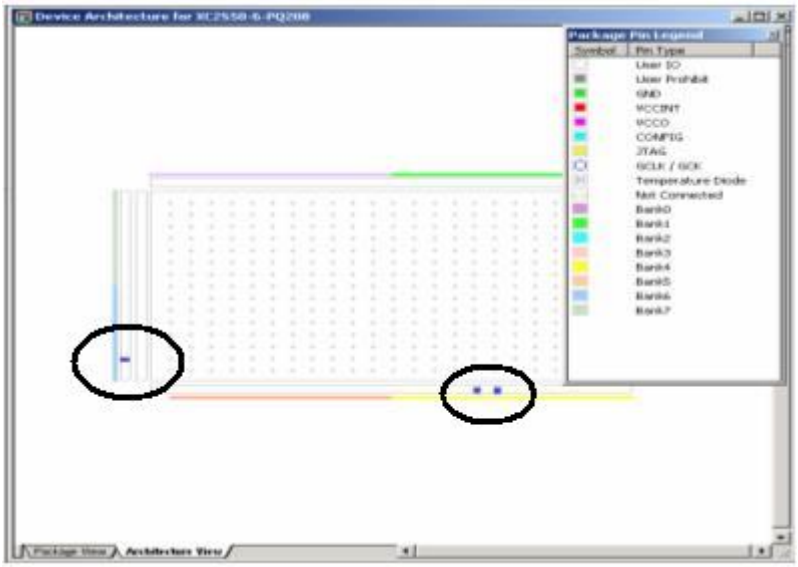


Figura 2.18: Ilustração da janela utilizada na assinalação dos pinos de E/S da porta NAND na FPGA.

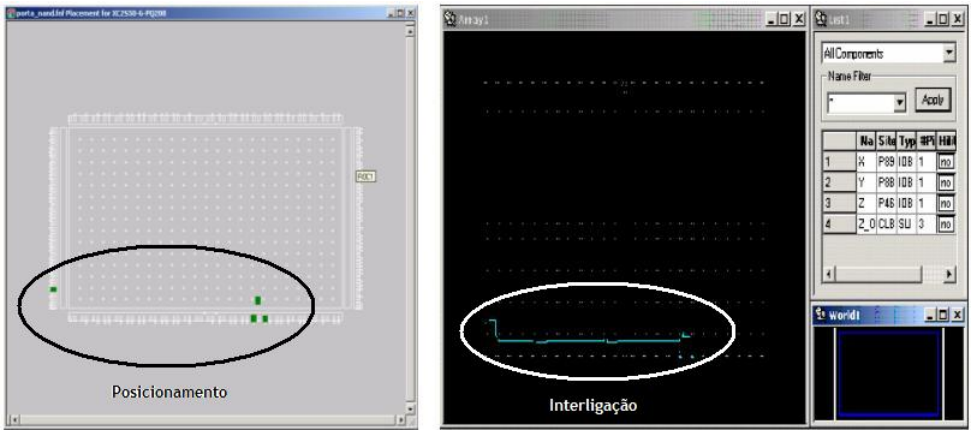


Figura 2.19: Ilustração da janela utilizada para posicionamento e interligação dos componentes na FPGA.

Após todas essas etapas, um arquivo de configuração da FPGA é gerado e, posteriormente, gravado na FPGA ou numa memória PROM externa, conforme ilustrado nas Figuras 2.20 e 2.21.

2.8.6 Materiais e Métodos

Um resumo do fluxo de projeto utilizado na implementação final em FPGA do amplificador sensível à fase e do módulo CAN é apresentado nas Figuras 2.22 e 2.23.

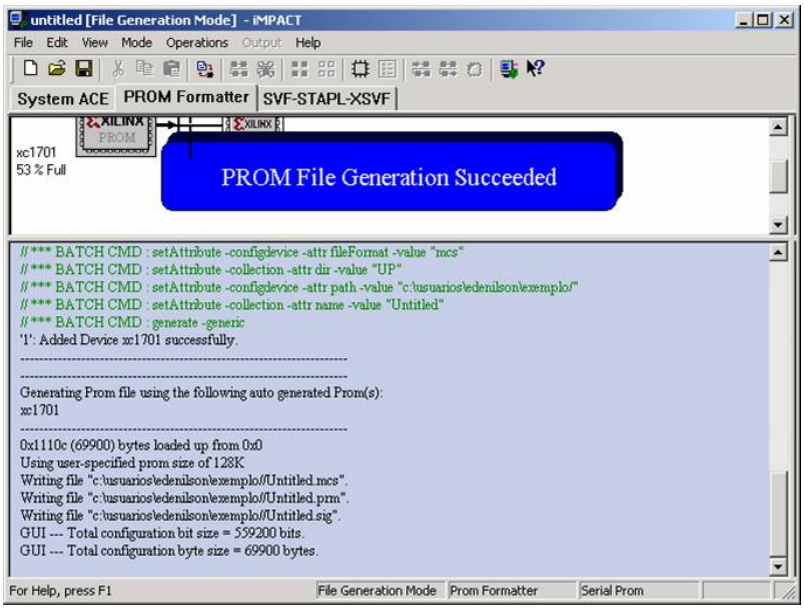


Figura 2.20: Ilustração da janela para geração do arquivo de configuração da FPGA.

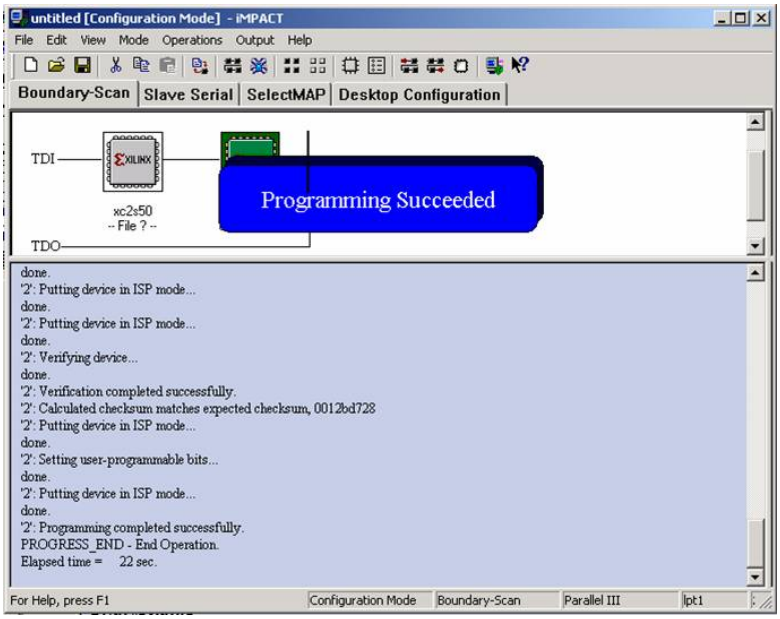


Figura 2.21: Ilustração da janela para gravação do arquivo de configuração da FPGA numa memória PROM externa.

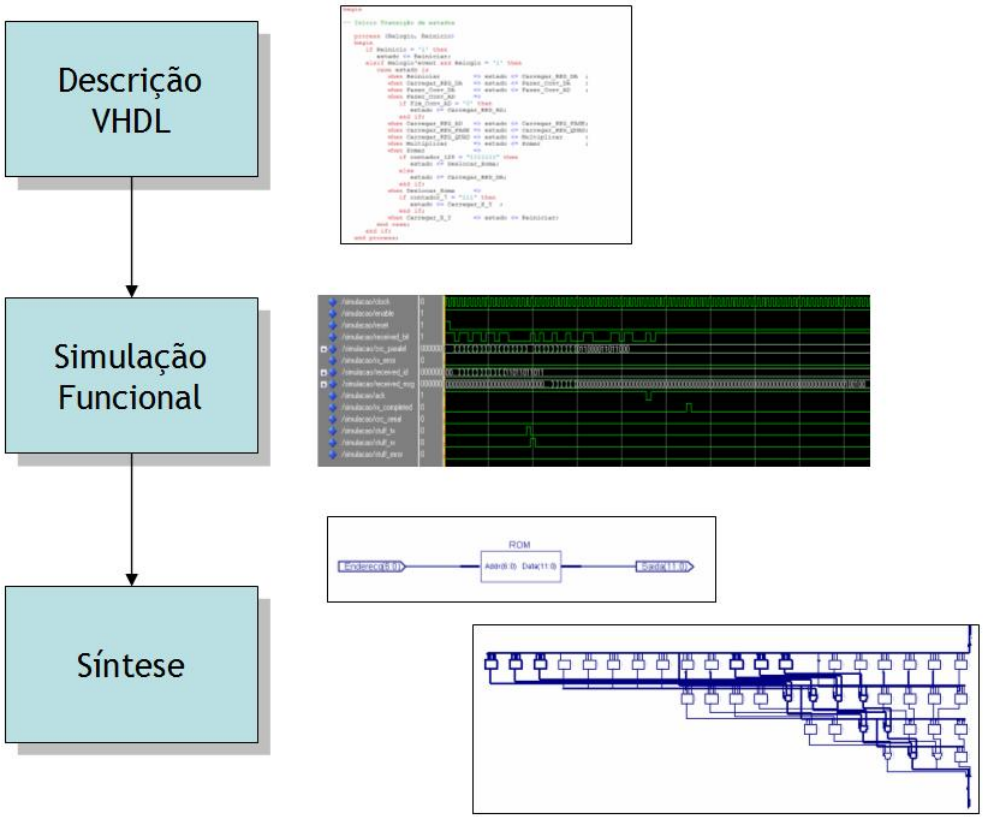


Figura 2.22: Fluxo de projeto utilizado.

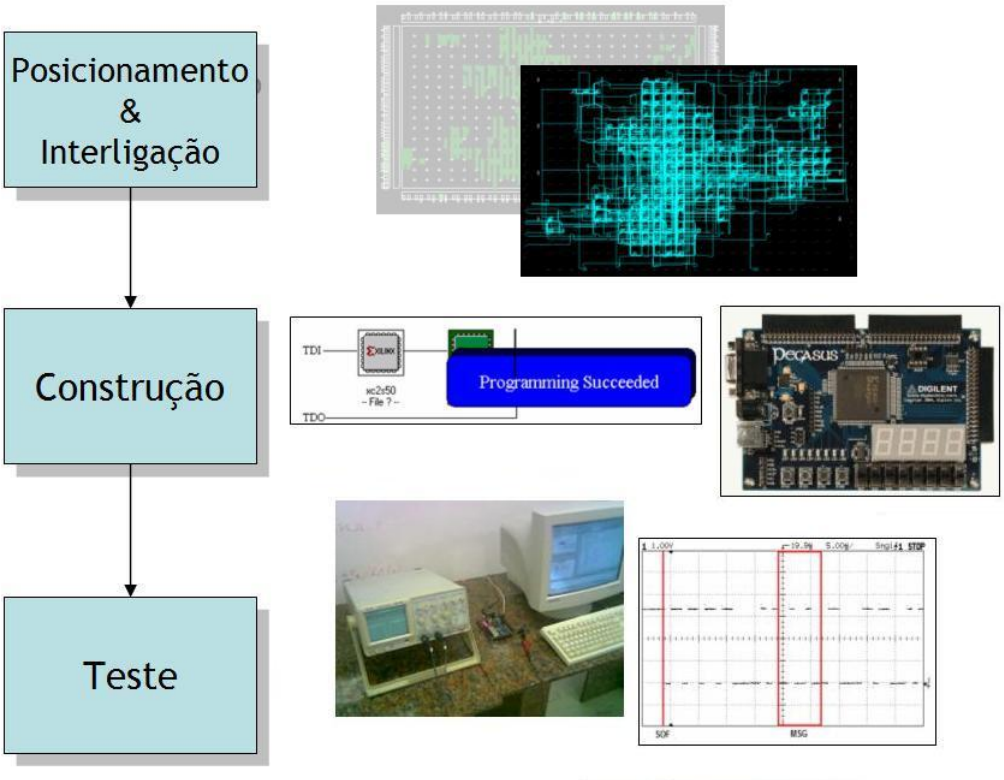


Figura 2.23: Fluxo de projeto utilizado (continuação).

Na descrição e síntese dos circuitos digitais desenvolvidos foi utilizado o conjunto de ferramentas de projeto *Xilinx ISE 11*, ilustrado na Figura 2.24. Para simulação dos circuitos foi utilizada a ferramenta *ModelSim XE III 6.4* (Figura 2.25). Finalmente, para implementação física e teste do circuito foi utilizada a plataforma de desenvolvimento *Spartan 3E Starter Kit*, do fabricante *Digilent* (Figura 2.26).

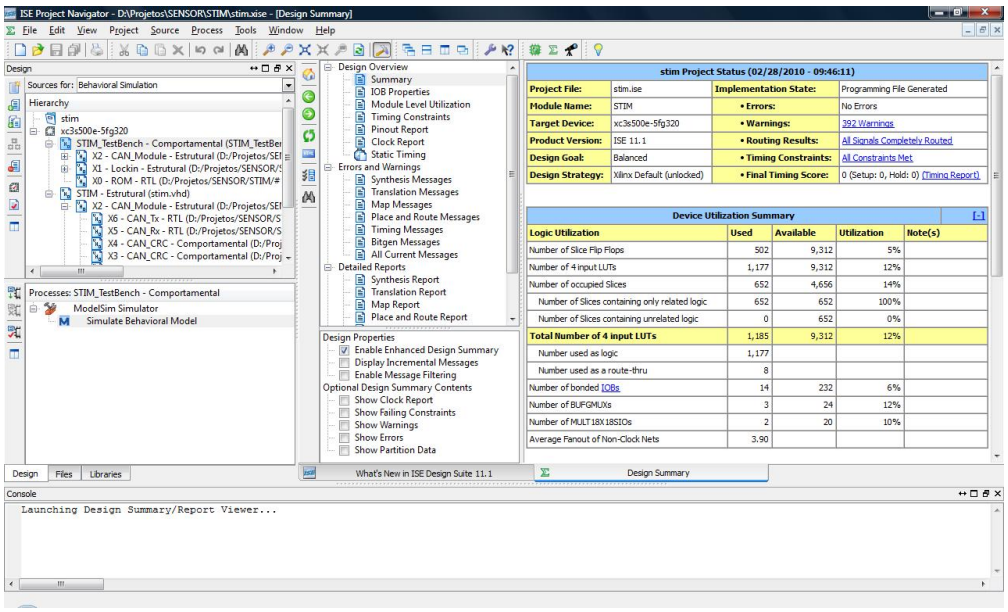


Figura 2.24: Ilustração do ambiente de projeto *Xilinx ISE 11*.

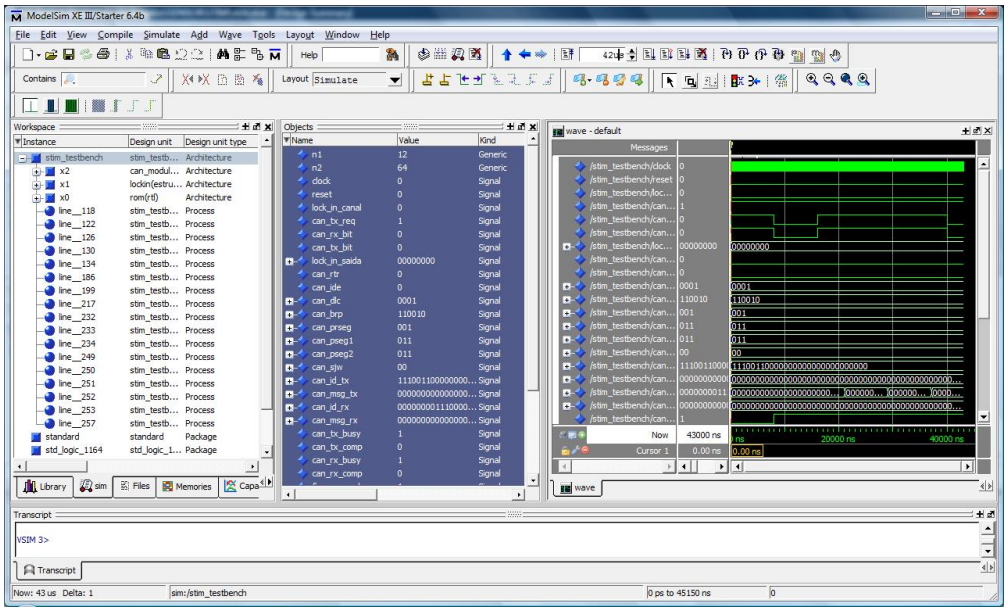


Figura 2.25: Ilustração da ferramenta de simulação *ModelSim XE III 6.4*.

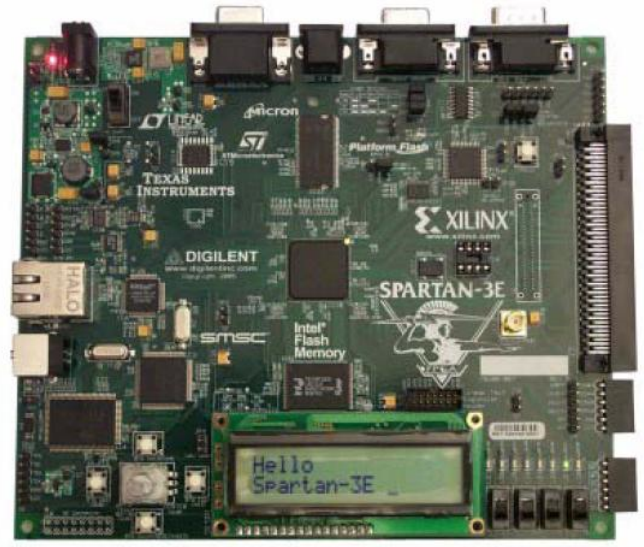


Figura 2.26: Ilustração da plataforma de desenvolvimento *Spartan 3E*.

2.9 Considerações Finais

Neste capítulo foi apresentada a metodologia utilizada neste trabalho, sendo abordadas todas as etapas utilizadas no desenvolvimento do amplificador sensível à fase digital *(lock-in)* e do módulo de comunicação CAN *(Controller Area Network)*.

Foram apresentados os materiais e métodos utilizados na implementação de um amplificador sensível à fase utilizando microcomputador com a placa de aquisição de dados DAS-20 e de uma rede CAN utilizando os módulos SBC28PC e microcontroladores PIC 18F258, disponíveis comercialmente.

Finalmente, foi discutida a metologia de projeto de circuitos integrados digitais descritos em VHDL e implementados em FPGA. Os materiais e métodos utilizados no desenvolvimento em VHDL/FPGA de um amplificador sensível à fase e de um módulo CAN foram apresentados. O fluxo de projeto ilustrado também pode ser utilizado na implementação de outros trabalhos.

Capítulo 3

Amplificador Sensível à Fase

Neste capítulo será realizada uma discussão teórica da técnica *lock-in*, abordando o princípio de funcionamento do amplificador sensível à fase[1]. Em seguida, será apresentada a implementação de um amplificador *lock-in* digital utilizando microcomputador com placa de aquisição de dados e MATLAB. Finalmente, será discutida a descrição em VHDL e o desenvolvimento em FPGA deste dispositivo.

3.1 Discussão Teórica

O amplificador sensível à fase é um dispositivo utilizado na medição de sinais elétricos que estão efetivamente abaixo do nível de ruído. O conceito utilizado é o de estreitar a banda de frequência, de forma que o sinal se sobressaia ao ruído [23].

O sinal de saída do amplificador sensível à fase é um sinal CC proporcional a um sinal CA sob investigação que, por sua vez, possui informações sobre determinadas propriedades elétricas (resistência, capacitância, etc.) de um dispositivo sob teste.

O princípio fundamental de funcionamento de um amplificador sensível à fase se baseia na técnica da detecção de fase, que retifica apenas o sinal de interesse, suprimindo os efeitos indesejados devidos ao ruído. Um sinal senoidal com amplitude, frequência e fase ajustáveis é aplicado a um dispositivo sob teste. A resposta à excitação é então capturada e demodulada internamente pelo sinal de referência. Em seguida, um filtro passa-baixa separa a componente contínua proporcional à propriedade elétrica desejada.

[1] Também conhecido como amplificador travado em fase ou amplificador *lock-in*.

A seguir serão discutidos os blocos que compõem um amplificador sensível à fase e a formulação matemática que torna possível a sua implementação digital.

Na Figura 3.1 é apresentado o diagrama em blocos de um amplificador sensível à fase.

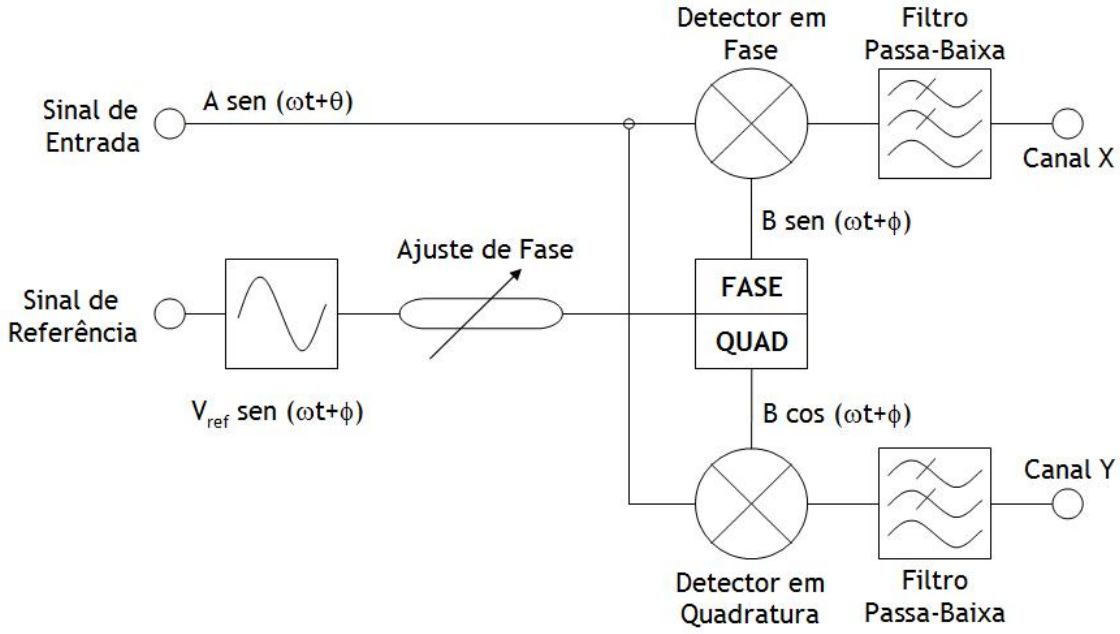


Figura 3.1: Diagrama em blocos de um amplificador sensível à fase.

As medições utilizando um amplificador sensível à fase são realizadas numa frequência fixa. Um sinal de frequência conhecida é aplicado a um dispositivo sob teste e a resposta à excitação é capturada. Um sinal de referência, de mesma frequência, é gerado internamente pelo amplificador e utilizado para demodular o sinal de entrada.

O sinal de entrada é multiplicado pelas componentes em fase e em quadratura do sinal de referência. Na saída de cada detector é obtido um sinal composto por um nível constante e outro modulado ao dobro da frequência de referência. O nível constante é proporcional à amplitude do sinal de entrada e ao cosseno (ou seno) do ângulo de fase entre o sinal de entrada e o sinal de referência.

Utilizando um filtro passa-baixa na saída do demodulador separa-se o nível constante, livre de ruído, recuperando assim as propriedades do dispositivo sob investigação.

A formulação matemática no domínio temporal de um amplificador sensível à fase é descrita a seguir.

Considere o sinal de entrada dado por

$$V_{en}(t) = Asen(\omega t + \theta), \tag{3.1}$$

onde A é a amplitude do sinal de entrada em Volts; $\omega = 2\pi f$, com f igual à frequência do sinal de entrada em Hertz; e θ é a fase do sinal de entrada em radianos.

Considere também as componentes em fase e em quadratura do sinal de referência, descritas por

$$V_{fase}(t) = Bsen(\omega t + \phi), \tag{3.2}$$

$$V_{quad}(t) = Bcos(\omega t + \phi), \tag{3.3}$$

onde B é a amplitude das componentes do sinal de referência em Volts; $\omega = 2\pi f$, com f igual à frequência do sinal de referência em Hertz; e ϕ é a fase do sinal de referência em radianos. Note que a frequência do sinal de entrada é considerada igual à frequência do sinal de referência.

Os detectores de fase multiplicam o sinal de entrada pelas componentes em fase e em quadratura do sinal de referência. Após os detectores de fase, tem-se que

$$V_{det,fase}(t) = ABsen(\omega t + \theta)sen(\omega t + \phi), \tag{3.4}$$

$$V_{det,quad}(t) = ABsen(\omega t + \theta)cos(\omega t + \phi). \tag{3.5}$$

As Equações 3.4 e 3.5 podem ser reescritas de forma que

$$V_{det,fase}(t) = \frac{1}{2}AB[cos(\theta - \phi) - cos(2\omega t + \theta + \phi)], \tag{3.6}$$

$$V_{det,quad}(t) = \frac{1}{2}AB[sen(\theta - \phi) + sen(2\omega t + \theta + \phi)]. \tag{3.7}$$

Como pode ser observado a partir das Equações 3.6 e 3.7, na saída de cada detector de fase é obtido um sinal composto por um nível constante, proporcional às amplitudes e ao cosseno (e seno) da diferença de fase do sinal de entrada e de referência, e um sinal modulado ao dobro da frequência de referência.

Os filtros passa-baixa separam o nível constante, rejeitando a componente modulada ao dobro da frequência de referência. Portanto, após os filtros, tem-se que

$$X = \frac{1}{2}ABcos(\theta - \phi), \tag{3.8}$$

$$Y = \frac{1}{2}ABsen(\theta - \phi). \tag{3.9}$$

De maneira geral, as equações de saída de um amplificador sensível à fase analógico, cujos sinais envolvidos são de tempo contínuo, podem ser escritas na forma

$$X = \frac{1}{T}\int_0^T V_{en}(t) \times V_{fase}(t)dt, \tag{3.10}$$

$$Y = \frac{1}{T}\int_0^T V_{en}(t) \times V_{quad}(t)dt, \tag{3.11}$$

onde T é uma janela de tamanho igual a um período (ou múltiplos inteiros de um período) do sinal de entrada.

No caso de um amplificador sensível à fase digital, os sinais envolvidos são provenientes de conversores analógico-digital (AD) e digital-analógico (DA), ou seja, são sinais de tempo discreto. Portanto, as equações de saída podem ser escritas como

$$X = \frac{1}{N}\sum_0^{N-1} V_{en}[i] \times V_{fase}[i], \tag{3.12}$$

$$Y = \frac{1}{N}\sum_0^{N-1} V_{en}[i] \times V_{quad}[i], \tag{3.13}$$

onde N é o número de pontos amostrados do sinal de entrada durante um período (ou múltiplos inteiros de um período).

A seguir serão apresentados alguns exemplos ilustrando graficamente o princípio de funcionamento do amplificador sensível à fase.

3.1.1 Caso 1: Sinal de Entrada em Fase com o Sinal de Referência

Considere um sinal de entrada de mesma amplitude e em fase com o sinal de referência. O sinal de referência possui amplitude unitária e frequência igual a 1 kHz. Os diversos sinais presentes no amplificador sensível à fase são ilustrados na Figura 3.2.

Observa-se, a partir da Figura 3.2, que os sinais após os detectores em fase e em quadratura apresentam uma freqûencia igual ao dobro da frequência do sinal de referência. Além disso, como o sinal de entrada está em fase com o sinal de referência, a tensão de saída do Canal X atinge seu valor máximo (igual à metade da amplitude do sinal de entrada), enquanto a tensão de saída do Canal Y atinge seu valor mínimo (igual à zero).

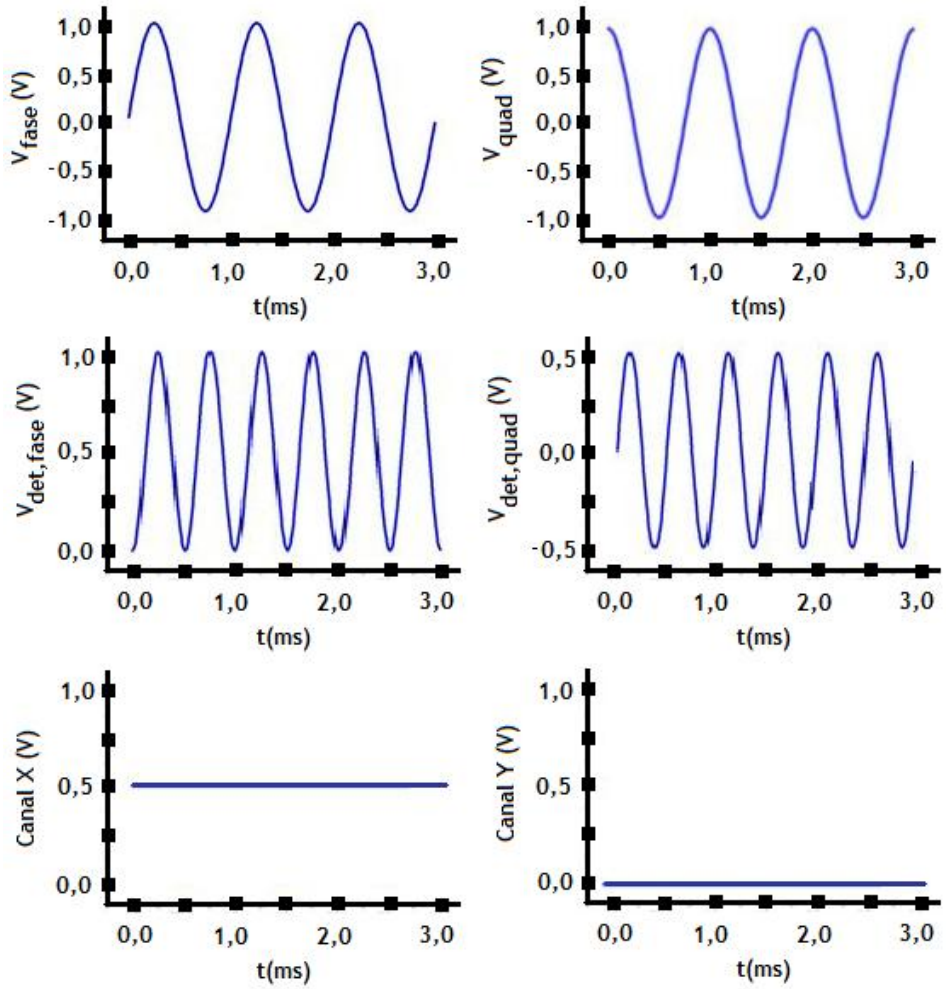


Figura 3.2: Gráficos das tensões presentes no amplificador sensível à fase para um sinal de entrada em fase com o sinal de referência.

3.1.2 Caso 2: Sinal de Entrada Defasado de 45 Graus com Relação ao Sinal de Referência

Considere agora um sinal de entrada de mesma amplitude e defasado de 45 graus em relação ao sinal de referência. Novamente, o sinal de referência possui amplitude unitária e frequência igual a 1 kHz. Os diversos sinais presentes no amplificador sensível à fase, para este caso, são ilustrados na Figura 3.3.

Observa-se, a partir da Figura 3.3, que os sinais após os detectores em fase e em quadratura apresentam o mesmo valor médio (aproximadamente 0,35V). Consequentemente, a tensão de saída do Canal X é igual à tensão de saída do Canal Y.

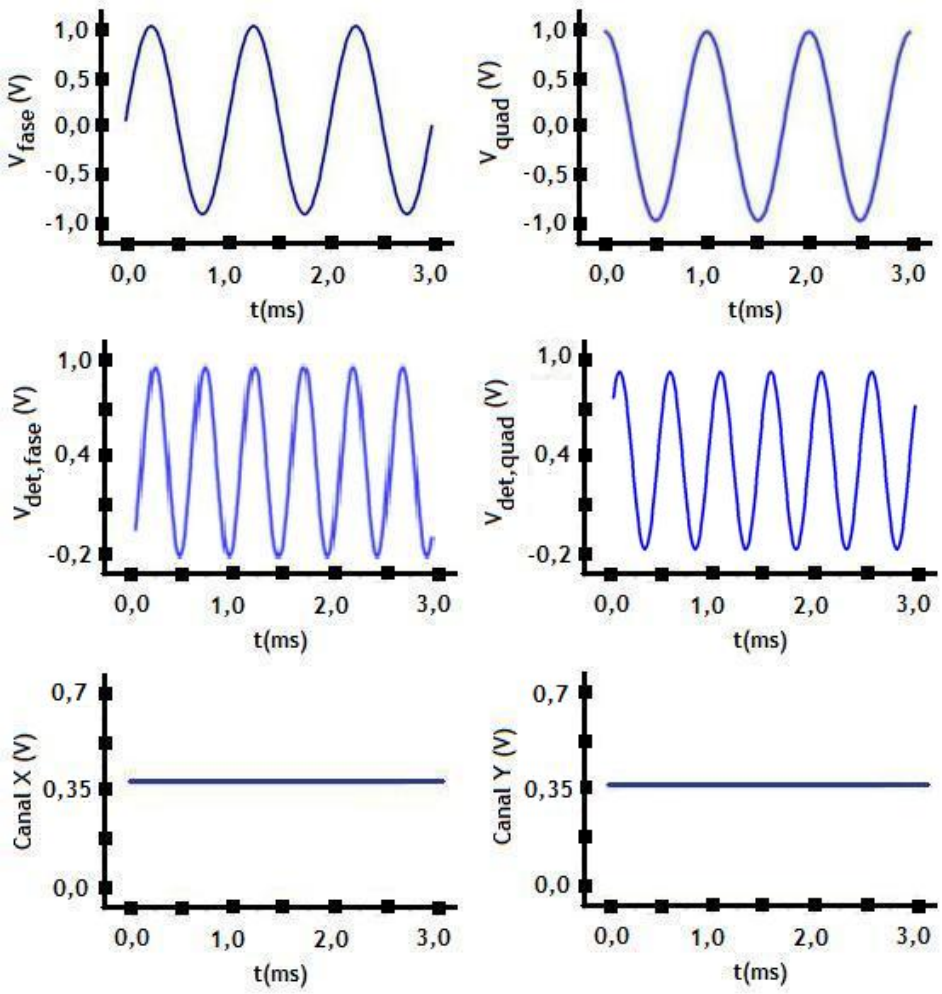


Figura 3.3: Gráficos das tensões presentes no amplificador sensível à fase para um sinal de entrada defasado de 45 graus com relação ao sinal de referência.

3.1.3 Caso 3: Sinal de Entrada em Quadratura com o Sinal de Referência

Na Figura 3.4 são ilustrados os diversos sinais presentes num amplificador sensível à fase, considerando um sinal de entrada de mesma amplitude e defasado de 90 graus com relação ao sinal de referência. O sinal de referência possui amplitude unitária e frequência de 1 kHz.

Neste caso, como o sinal de entrada está em quadratura com o sinal de referência, a tensão de saída do Canal X atinge seu valor mínimo (igual à zero), enquanto a tensão de saída do Canal Y atinge seu valor máximo (igual à metade da amplitude do sinal de entrada).

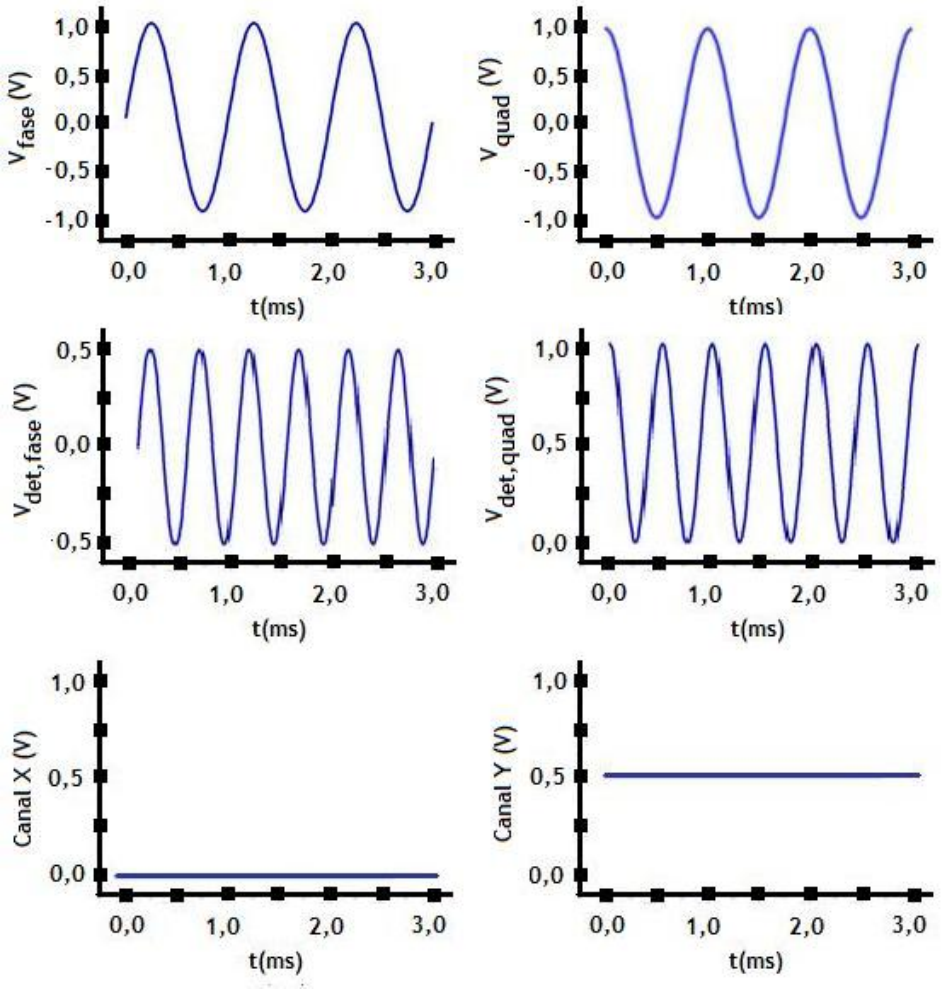


Figura 3.4: Gráficos das tensões presentes no amplificador sensível à fase para um sinal de entrada em quadratura com o sinal de referência.

3.2 Exemplo de Aplicação: Medição de Impedâncias

Na Figura 3.5 é apresentado um diagrama esquemático de um circuito experimental utilizando um amplificador sensível à fase para medição de impedâncias. Nesta figura, V_{teste} é o sinal de excitação gerado pelo amplificador *lock-in*, Y é um dispositivo sob teste, representado por uma admitância, e V_{ent} é o sinal de entrada do amplificador.

Considerando-se que o dispositivo sob teste da Figura 3.5 é uma admitância Y dada por uma condutância G e uma capacitância C, e aplicando-se um sinal V_{teste} cuja amplitude V_{ref}, frequência angular ω e fase ϕ são ajustadas pelo experimentador, obtém-se um sinal V_{ent}, conforme descrito a seguir.

$$V_{teste} = V_{ref} sen(\omega t + \phi), \tag{3.14}$$

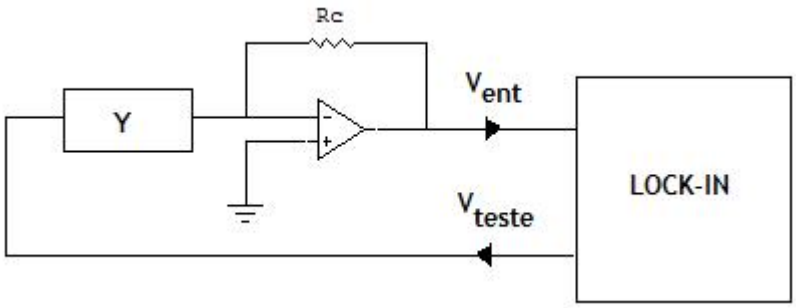


Figura 3.5: Diagrama esquemático de um circuito experimental para medição de impedâncias utilizando um amplificador sensível à fase.

$$Y = G + j\omega C = \frac{1}{R} + j\omega C, \tag{3.15}$$

$$V_{ent} = -R_c I = -R_c Y V_{teste}, \tag{3.16}$$

$$V_{ent} = -R_c V_{ref}[Gsen(\omega t + \phi) + \omega C cos(\omega t + \phi)], \tag{3.17}$$

onde I é a corrente elétrica sobre o resitor R_c; R_c é a resistência do amplificador inversor (conversor corrente-tensão) implementado com o amplificador operacional; e R é a resistência do dispositivo sob teste.

Os sinais de saída dos detectores em fase e em quadratura são dados por

$$V_{det,fase} = -\frac{1}{2}R_c V_{ref}[G(1 - cos(2\omega t + 2\phi)) + \omega C sen(2\omega t + 2\phi)], \tag{3.18}$$

$$V_{det,quad} = -\frac{1}{2}R_c V_{ref}[Gsen(2\omega t + 2\phi)) + \omega C(1 - cos(2\omega t + 2\phi))]. \tag{3.19}$$

Finalmente, os sinais de saída dos canaix X e Y são dados por

$$X = -\frac{1}{2}R_c V_{ref} G, \tag{3.20}$$

$$Y = -\frac{1}{2}R_c V_{ref} \omega C. \tag{3.21}$$

Nota-se que os valores CC das tensões nos canais X e Y são diretamente proporcionais à condutância e capacitância do dispositivo sob teste, respectivamente. Logo, a resistência e a capacitância do dispositivo são dadas por

$$R = -\frac{1}{2}R_c V_{ref}\frac{1}{X}, \tag{3.22}$$

$$C = -\frac{2}{\omega R_c V_{ref}} Y. \tag{3.23}$$

Portanto, uma vez que o sinal de teste é conhecido, pode-se determinar as propriedades elétricas (resistência e capacitância) de um dispositivo sob teste analisando-se a resposta à excitação. A partir do sinal de tensão obtido no canal X determina-se a resistência. Por outro lado, o canal Y informa a capacitância do dispositivo sob investigação.

Na próxima seção será discutida a implementação de um amplificador sensível à fase digital utilizando, inicialmente, um microcomputador com placa de aquisição de dados e MATLAB. Em seguida, será apresentada a implementação do amplificador *lock-in* digital em FPGA.

3.3 Amplificador Sensível à Fase Digital Utilizando Microcomputador com Placa de Aquisição de Dados e MATLAB

Nesta seção será apresentada a implementação de um amplificador *lock-in* digital utilizando um microcomputador com placa de aquisição de dados DAS-20 e MATLAB. Também serão discutidos os resultados obtidos em sua aplicação na medição de impedâncias.

3.3.1 Implementação

Na Figura 3.6 é apresentado o diagrama em blocos do amplificador *lock-in* digital utilizando a placa de aquisição de dados DAS-20 e MATLAB. Com a placa DAS-20, o sinal de teste é gerado e a aquisição da resposta à excitação (sinal de entrada) é realizada. No MATLAB é executado o algoritmo no qual a técnica *lock-in* propriamente dita é implementada.

A DAS-20, desenvolvida pela *Keithley Instruments*, é uma placa de aquisição de dados para computadores IBM/PC e compatíveis. A interface entre o computador e a placa é realizada via barramento ISA, de forma que a DAS-20 pode ser utilizada também em computadores mais antigos. Na implementação do amplificador *lock-in* digital foram utilizados um canal de saída analógico operando de -5 V a +5 V e um canal de entrada analógica operando de -10 V a +10 V.

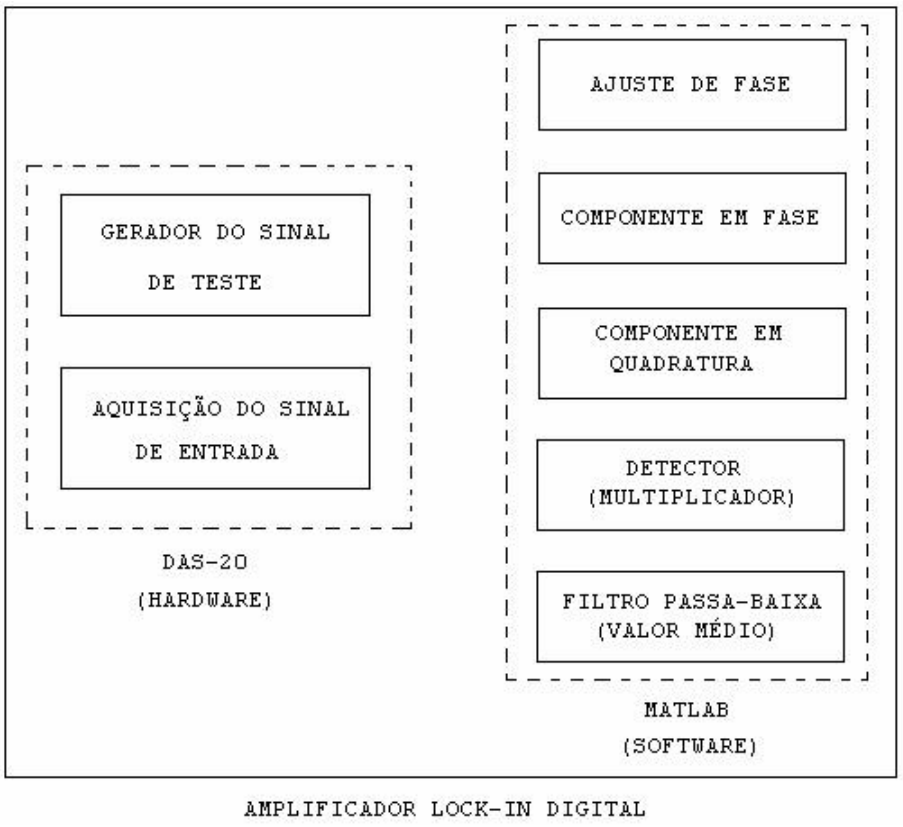


Figura 3.6: Diagrama em blocos do amplificador *lock-in* digital utilizando microcomputador com placa de aquisição de dados e MATLAB.

A placa DAS-20 é empregada na geração do sinal de teste (utilizado para excitar o dispositivo a ser analisado) e na aquisição do sinal de entrada (resposta à excitação). Para realizar o controle da placa foi utilizado um *driver* desenvolvido por *Tony L. Keiser* [24]. Esse *driver* é constituído por uma biblioteca de funções que implementam diversos comandos da DAS-20 e podem ser utilizadas no código de um programa escrito em linguagem C.

O fluxograma do *software* de controle da placa DAS-20, desenvolvido para a implementação do amplificador *lock-in* digital é ilustrado na Figura 3.7.

Ao iniciar a execução do programa, o usuário informa os parâmetros do sinal de teste, o qual será utilizado para excitar o dispositivo sob análise. Nesta etapa, são definidos: amplitude, frequência e fase do sinal de referência. É definido, também, o tempo no qual o dispositivo sob teste será excitado.

O próximo passo é a inicialização da placa DAS-20, que é realizada automaticamente pelo programa. Nesse momento, é realizado um *reset* geral da placa. Em seguida, a frequência de amostragem é definida, as interrupções são configuradas e os

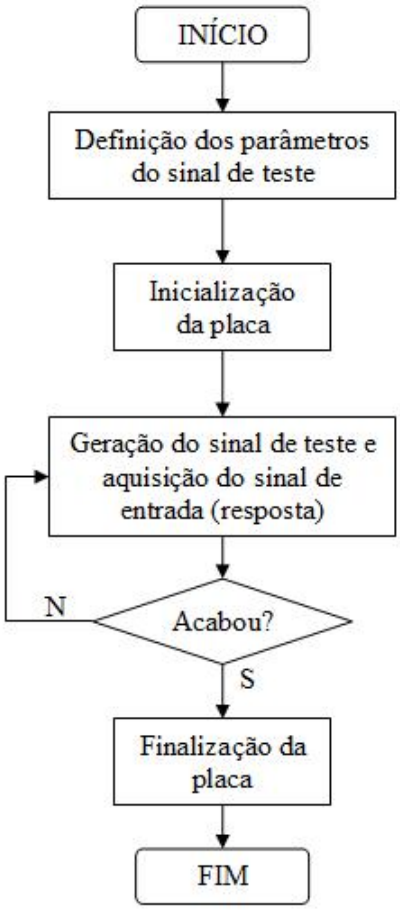


Figura 3.7: Fluxograma do *software* de controle da placa DAS-20.

temporizadores selecionados.

Após o processo de inicialização da placa, são iniciadas a geração do sinal de teste e a aquisição do sinal de entrada (resposta à excitação do dispositivo sob análise). O sinal de teste é gerado durante o intervalo de tempo determinado pelo usuário na definição dos parâmetros. Durante a aquisição dos dados, um arquivo de texto é criado para armazenar as informações obtidas pela placa DAS-20. Esse arquivo é utilizado posteriormente pelo MATLAB na execução do algoritmo desenvolvido para a implementação da técnica *lock-in*.

Ao término do tempo de excitação, é realizado um novo *reset* geral da placa DAS-20 e o programa é encerrado. A tela do principal do programa de aquisição de dados, escrito em linguagem C, é apresentada na Figura 3.8.

Após a aquisição de dados, as operações de multiplicação e cálculo da média (detectores de fase e filtros passa-baixa, respectivamente) são executadas a partir

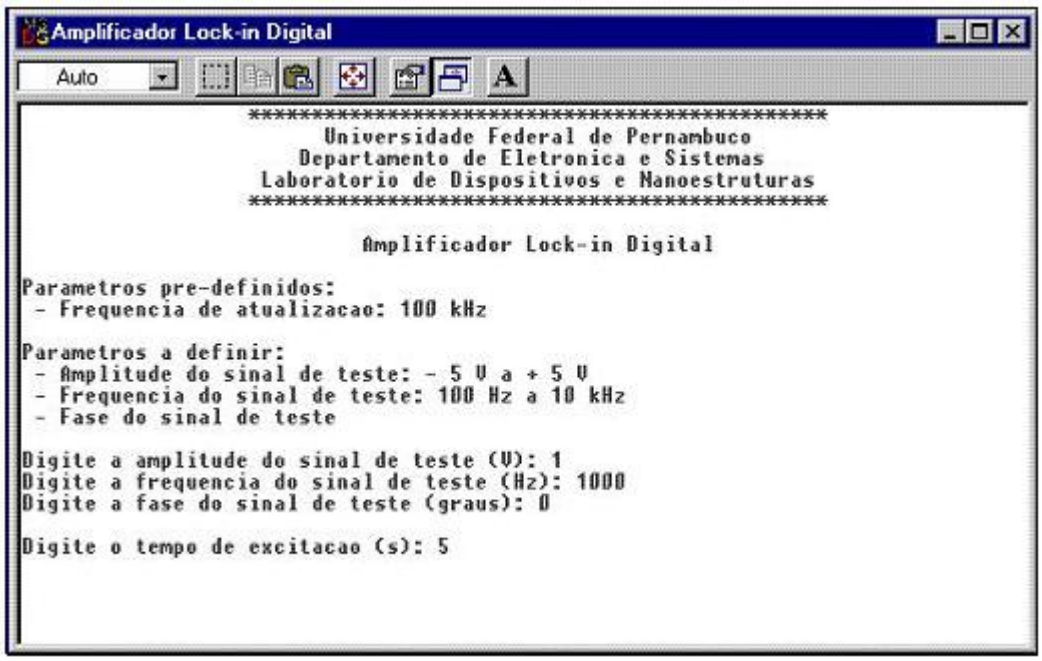


Figura 3.8: Tela principal do programa de aquisição de dados.

do algoritmo implementado no MATLAB. O programa desenvolvido no MATLAB é constituído de dois arquivos: *lockincal.m* e *lockinmed.m*. O arquivo *lockincal.m* é utilizado na calibração do amplificador *lock-in* digital e no ajuste de fase. Essa etapa é necessária para eliminar os erros de fase inerentes ao circuito de medição, externo ao amplificador, causados, por exemplo, por capacitâncias parasitas. Já o arquivo *lockinmed.m* retorna os valores dos canais X e Y, calculando os valores de resistência e capacitância do dispositivo sob teste.

A Figura 3.9 ilustra o fluxograma básico do algoritmo desenvolvido no MATLAB.

Inicialmente, é realizada a calibração do amplificador *lock-in*, a partir do ajuste de fase. Em seguida, o sinal de referência, constituído pelas componentes em fase e em quadratura, é ajustado de acordo com o sinal de excitação aplicado com a placa DAS-20. Vetores são definidos de forma a armazenarem os sinais de referência (componentes em fase e quadratura) e de entrada (obtido do arquivo de texto gerado na aquisição dos dados).

O detector de fase digital é modelado matematicamente como uma operação de multiplicação de vetores (elemento-a-elemento). O nível constante é obtido calculando-se o valor médio do sinal demodulado. Uma vez que o sinal demodulado é discreto, o filtro passa-baixa consiste de uma simples média aritmética dos dados armazenados

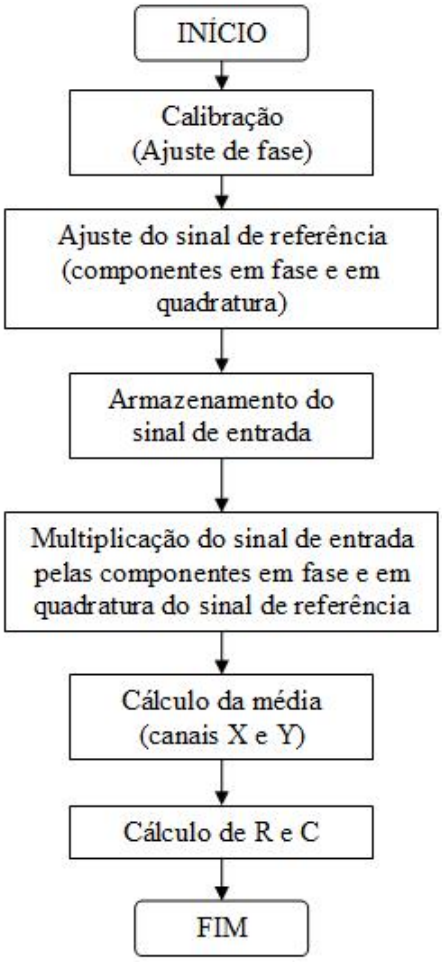


Figura 3.9: Fluxograma do algoritmo desenvolvido no MATLAB.

durante um período do sinal de entrada, produzindo os valores dos canais X e Y. Finalmente, os valores de resistência e capacitância são determinados.

A tela com as saídas produzidas pelo programa em MATLAB é apresentada na Figura 3.10. Os códigos desenvolvidos em linguagem C e MATLAB são apresentados no Apêndice B.

3.3.2 Resultados

Nesta seção serão analisados os resultados obtidos na medição de impedâncias com o amplificador *lock-in* digital utilizando microcomputador com placa de aquisição de dados DAS-20 e MATLAB.

O diagrama esquemático do circuito experimental para medição de impedâncias utilizando o amplificador *lock-in* desenvolvido é apresentado na Figura 3.11. Este

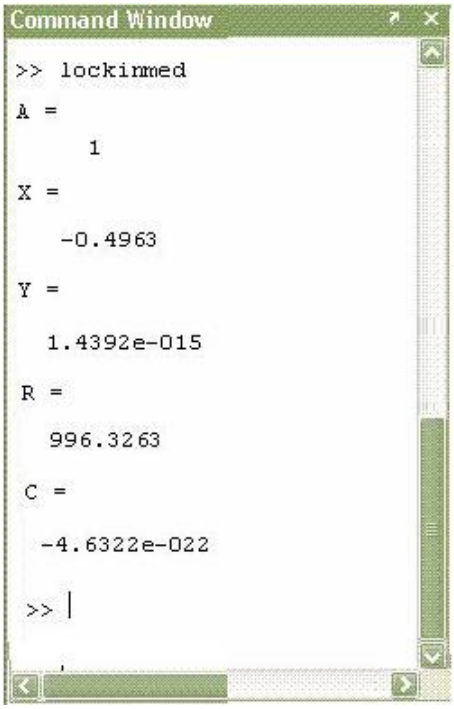


Figura 3.10: Tela do programa desenvolvido em MATLAB para medição de impedâncias com o amplificador *lock-in*.

esquema é baseado no circuito genérico da Figura 3.5, com a inserção de um amplificador operacional na configuração *Buffer*. A função deste amplificador operacional é isolar o circuito de condicionamento do amplificador *lock-in* digital, protegendo a placa de aquisição de dados.

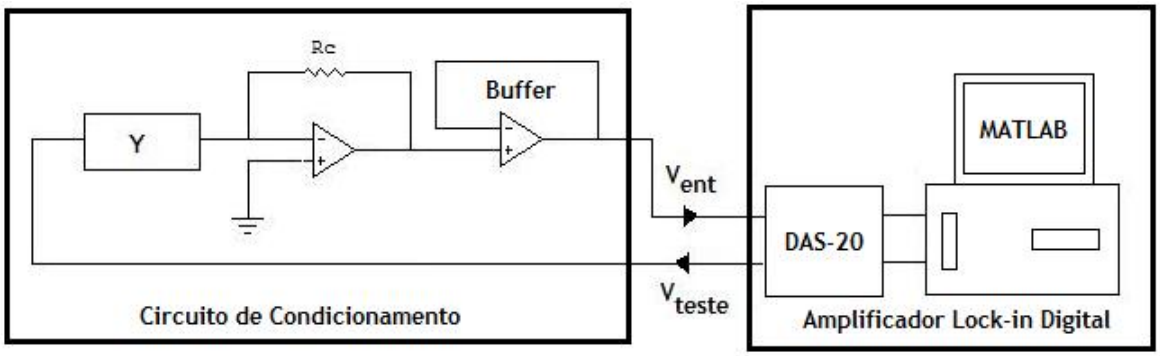


Figura 3.11: Diagrama esquemático de um circuito experimental para medição de impedâncias utilizando o amplificador *lock-in* digital com placa de aquisição de dados e MATLAB.

O circuito experimental montado em laboratório é apresentado na Figura 3.12.

Inicialmente, foi utilizado um sinal de teste, com amplitude igual a 1V e frequência igual a 1kHz. Como dispositivo sob teste foi utilizado um resistor de 1kΩ (valor

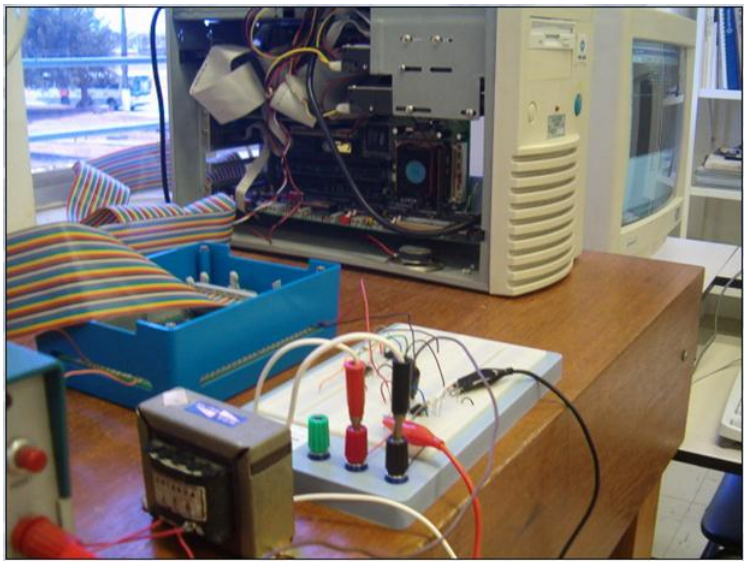


Figura 3.12: Amplificador *lock-in* digital com placa de aquisição de dados e MATLAB utilizado na medição de impedâncias.

nominal) e tolerância de 5%. O valor de resistência medido com um multímetro digital foi de 998Ω. No circuito conversor corrente-tensão, também foi utilizado um resistor $R_c = 1k\Omega$.

Uma vez que o dispositivo sob teste é constituído apenas por um resistor $R = R_c$, os valores teóricos de saída são iguais a $X = -0,5V$ e $Y = 0V$.

Após a excitação do dispositivo sob teste (resistor), aquisição de dados (resposta à excitação) e execução do algoritmo no MATLAB, foram obtidos os valores de saída $X = -0,4963V$ e $Y = 1,439 \times 10^{-15}V$.

Os valores de resistência e capacitância fornecidos pelo programa *lockinmed.m* foram: $R = 996,3263\Omega$ e $C = 4,6322 \times 10^{-22}F$. Portanto, o valor de resistência medido com o amplificador *lock-in* apresentou um desvio de 0,37 % em relação ao valor nominal e 0,16 % em relação ao valor medido com o multímetro. Além disso, o valor de capacitância é aproximadamente zero.

Os resultados obtidos estão resumidos na Tabela 3.1.

A dependência linear do sinal de saída do canal X em função da condutância do dispositivo sob teste é observada na Figura 3.13. Verifica-se, a partir desta figura, que os resultados experimentais obtidos com o amplificador *lock-in* digital utilizando placa de aquisição de dados e MATLAB apresentam um alto grau de concordância com os resultados teóricos.

Tabela 3.1: Resultados obtidos na caracterização de um resistor de 1 kΩ com o amplificador *lock-in*.

	Valor Teórico	Valor Obtido	Desvio
Canal X	-0,5V	-0,4963V	0,7 %
Canal Y	0	1,4392 $\times 10^{-15}$V	-
R	1kΩ	996,3263 Ω	0,37 %
C	0	4,6322 $\times 10^{-22}$ F	-

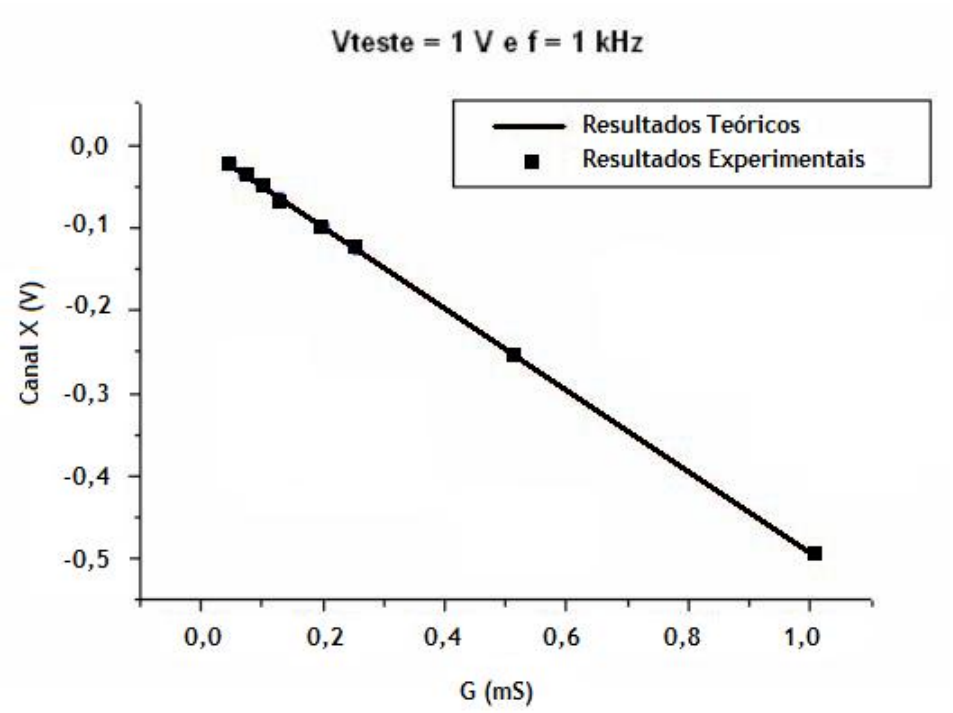


Figura 3.13: Gráfico da tensão de saída do canal X em função da condutância do dispositivo sob teste (teoria e prática).

Este protótipo, entretanto, apresentou limitações na medição de capacitâncias.

Inserindo-se uma capacitância no dispositivo sob teste Y da Figura 3.11, o circuito de condicionamento torna-se um circuito derivador [25]. A tensão de excitação (V_{teste}) é um sinal produzido por um conversor DA da placa DAS-20, apresentando, portanto, descontinuidades. Estas descontinuidades, por sua vez, aumentam à medida que a velocidade de atualização das saídas do conversor DA diminui.

Uma vez que a derivada de uma função pulso produz a função impulso [26], o sinal de excitação, ao passar pelo circuito derivador, gera picos de tensão a cada descontinuidade. Estes picos de tensão impossibilitaram a obtenção de uma resposta à excitação adequada, de forma que não foi possível a medição de capacitâncias.

3.4 Amplificador Sensível à Fase Descrito em VHDL e Implementado em FPGA

Nesta seção será apresentada a descrição VHDL de um amplificador sensível à fase digital e sua implementação em FPGA.

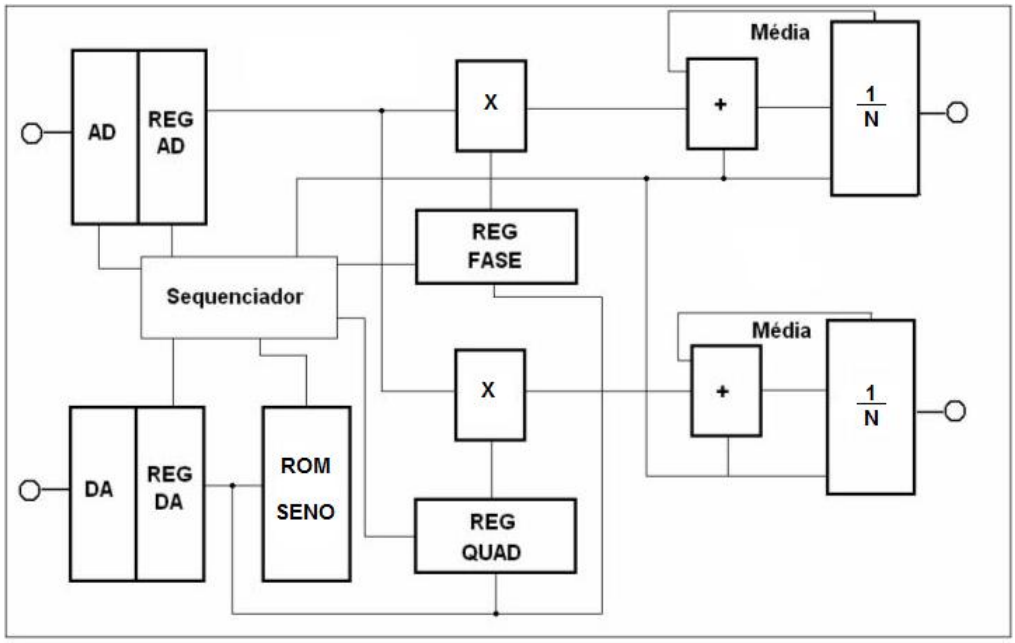


Figura 3.14: Diagrama em blocos do amplificador sensível à fase digital.

O diagrama em blocos do amplificador sensível à fase digital é apresentado na Figura 3.14. Os detectores em fase e em quadratura são representados pelos blocos de multiplicação. Os filtros passa-baixa (cálculo da média aritmética) são representados pelos blocos acumuladores e de divisão por N (número de pontos amostrados num período do sinal de entrada). O sinal de referência é gravado na memória ROM, contendo um período completo da senóide. O sequenciador é o bloco responsável pela execução das operações internas do amplificador *lock-in* digital.

No desenvolvimento deste projeto foi utilizada a estratégia de "dividir para conquistar". Inicialmente, o sistema digital foi fragmentado em vários blocos. Em seguida, cada bloco foi descrito em VHDL, a nível RTL, seguindo uma metodologia de projeto *top-down*. Finalmente, todos os blocos foram interconectados numa mesma entidade principal, utilizando uma descrição estrutural.

Alguns blocos mais complexos, tais como o multiplicador, foram ainda divididos

em sub-blocos, que também foram descritos a nível RTL e em seguida interconectados de forma estrutural.

O código VHDL do amplificador sensível à fase é apresentado no Apêndice A.

3.4.1 Detector de Fase

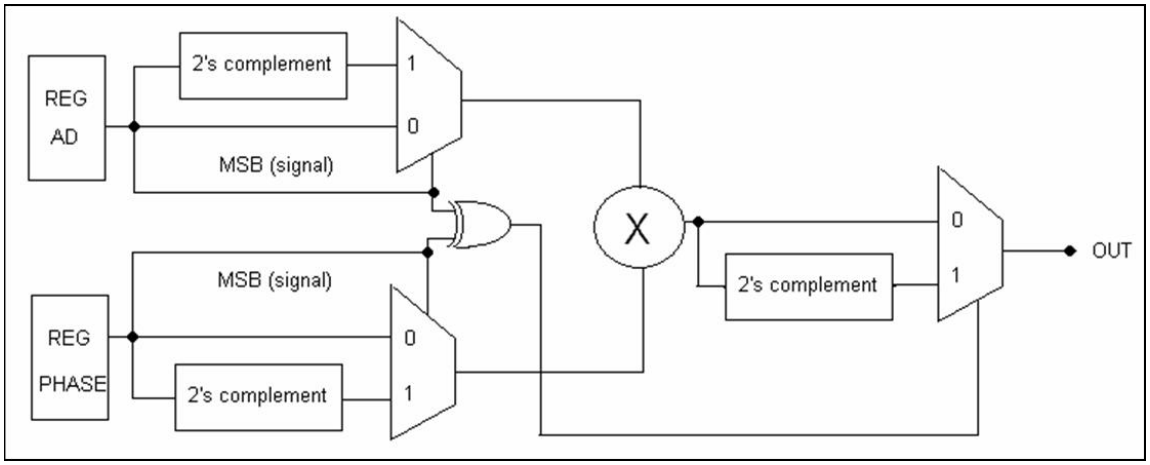


Figura 3.15: Diagrama em blocos do detector de fase.

A Figura 3.15 apresenta o diagrama em blocos do detector de fase. O núcleo do detector de fase consiste de um multiplicador binário de números positivos. Para lidar com números negativos, foram acrescentados blocos para cálculo do complemento a 2 e alguns multiplexadores.

Após a descrição VHDL individual de cada bloco, foi criada uma arquitetura estrutural para o detector de fase, interligando os componentes de cálculo do complemento a dois, os multiplexadores e o multiplicador.

O esquemático RTL de cada bloco do detector de fase gerado pela ferramenta de síntese é apresentado nas Figuras 3.16 a 3.19.

De acordo com o ilustrado na Figura 3.16, o complemento a 2 é calculado invertendo-se os 12 bits da palavra de entrada e somando-se 1 ao resultado. A partir da Figura 3.17, observa-se que o multiplexador 2-1 foi implementado utilizando-se portas lógicas. Por sua vez, o multiplexador 2-1 de 12 bits foi implementado a partir da instanciação de 12 multiplexadores 2-1 de 1 bit, conforme ilustrado na Figura 3.18. Finalmente, na Figura 3.19, pode-se verificar a inferência de um *flip-flop* após o multiplicador binário, de forma a armazenar o último resultado.

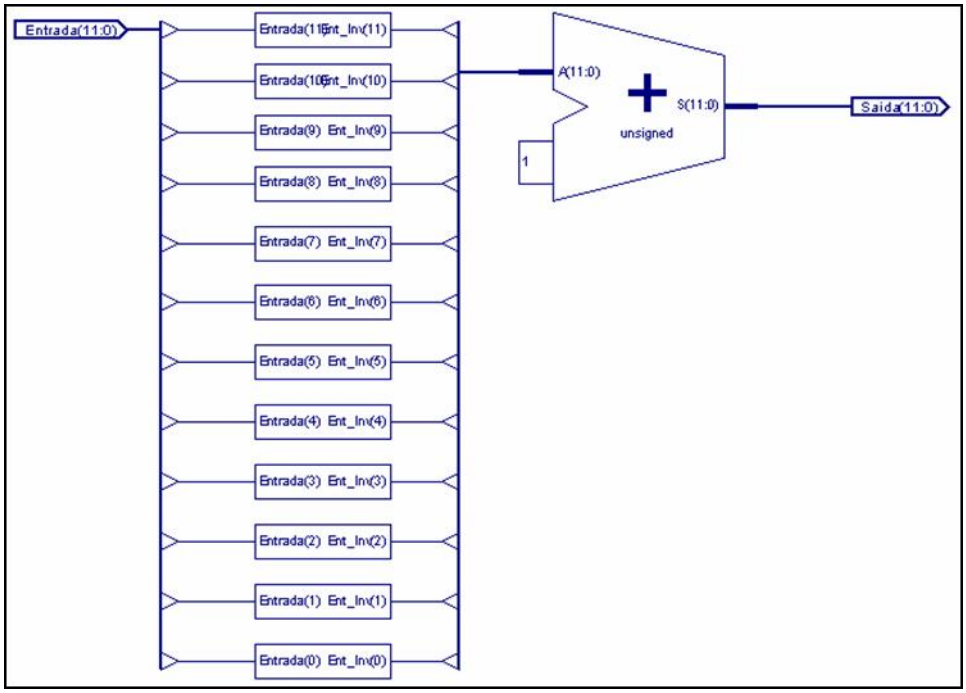


Figura 3.16: Esquemático RTL do bloco de cálculo do complemento a 2.

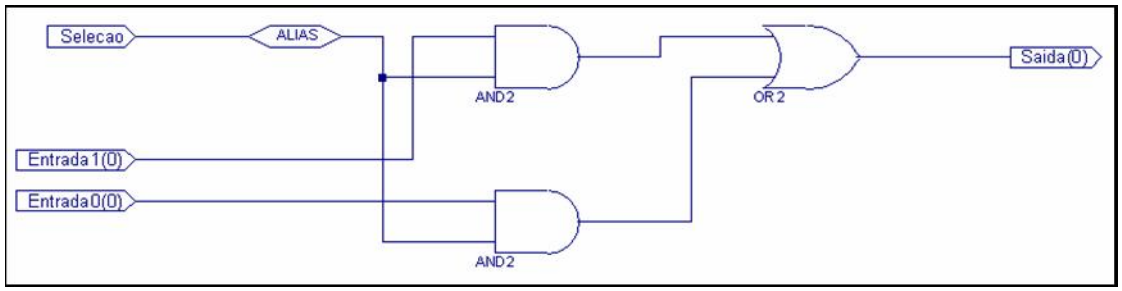


Figura 3.17: Esquemático RTL do bloco multiplexador 2-1 (1 *bit*).

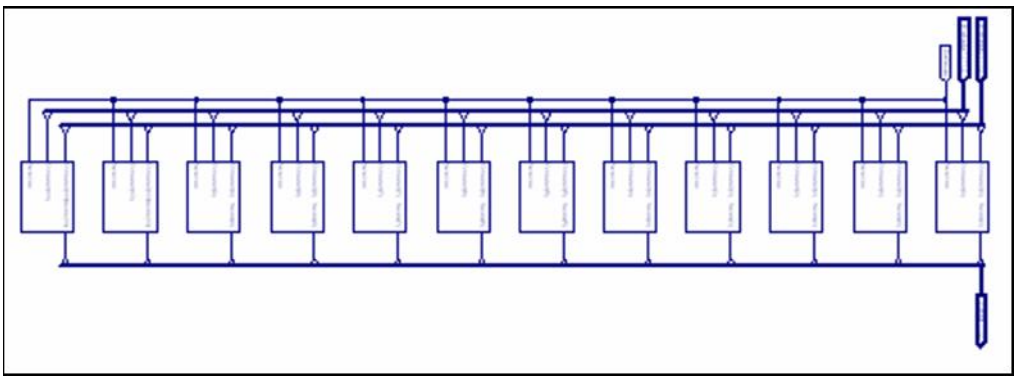


Figura 3.18: Esquemático RTL do bloco multiplexador 2-1 (12 *bits*).

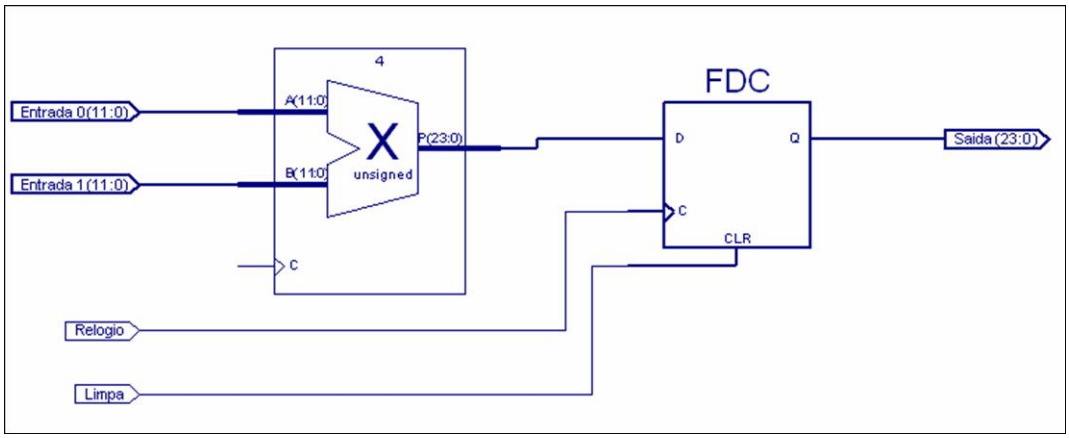


Figura 3.19: Esquemático RTL do bloco multiplicador binário.

3.4.2 Filtro Passa-Baixa

Um filtro passa-baixa de primeira ordem pode ser visto como um circuito para cálculo da média aritmética de N termos. Um circuito para cálculo da média aritmética pode ser implementado com um somador (acumulador) e um divisor. Se o número de termos é uma potência de 2, um registrador de deslocamento pode ser usado para efetuar a divisão por N.

O cálculo da média é realizado a cada período do sinal de referência. Ao final de um período, o registrador de deslocamento é reinicializado, de forma a recomeçar os cálculos. Para manter o último valor calculado nos canais de saída (X e Y), um registrador de *bits* é utilizado como *buffer*.

Os sinais de entrada e de referência são armazenados em registradores de 12 *bits*. Já os sinais demodulados (após os multiplicadores), por sua vez, são armazenados em registradores de 24 *bits*. Para prevenir a ocorrência de um *overflow* no acumulador de saída, foi utilizado um *buffer* (e, consequentemente, um registrador de deslocamento) de 64 *bits*.

O diagrama em blocos do filtro passa-baixa e o esquemático RTL de cada bloco gerado pela ferramenta de síntese estão ilustrados nas Figuras 3.20 a 3.23.

A inferência de um somador de 64 bits é ilustrada na Figura 3.21. A partir da Figura 3.22 observa-se a síntese de uma estrutura com 64 blocos em cascata, onde cada bloco representa um *flip-flop*, implementando o registrador de deslocamento. Já na Figura 3.23, verifica-se a inferência de um registrador de 64 bits, implementando o *buffer*.

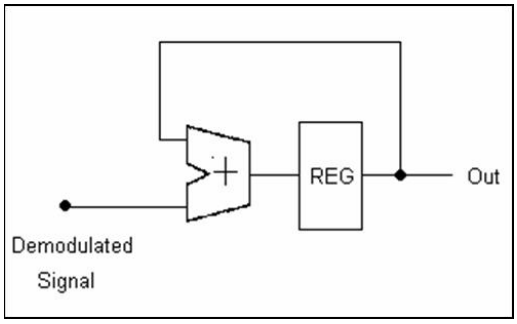


Figura 3.20: Diagrama em blocos do filtro passa-Baixa.

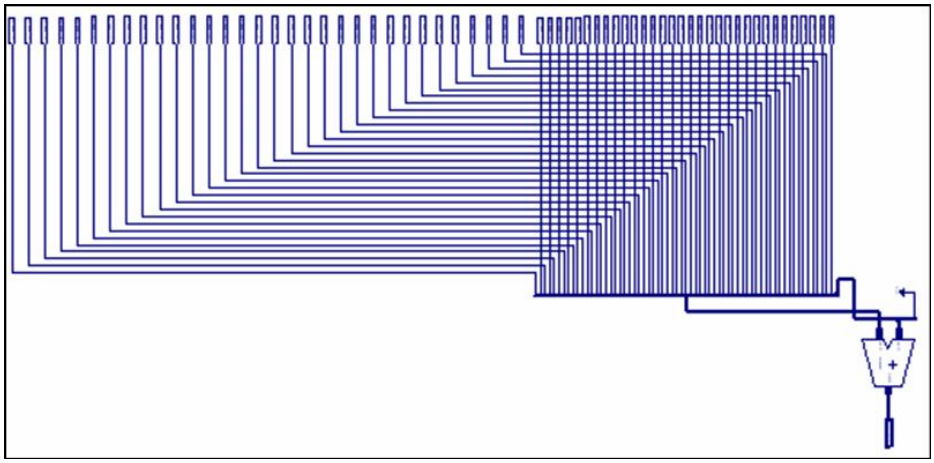


Figura 3.21: Esquemático RTL do bloco somador.

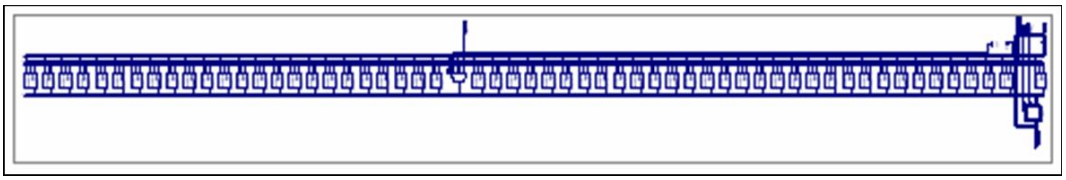


Figura 3.22: Esquemático RTL do bloco registrador de deslocamento.

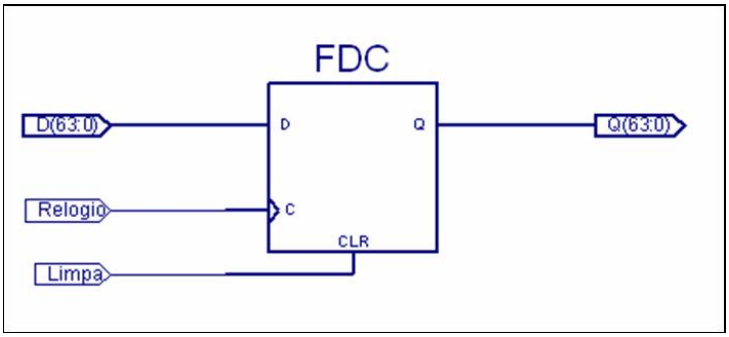


Figura 3.23: Esquemático RTL do bloco registrador *buffer*.

3.4.3 Memória ROM

Para armazenar o sinal de referência, é utilizada uma memória ROM na qual está gravado um período completo do sinal senoidal de referência. Foi considerado, inicialmente, que o sinal de referência possui frequência de 1 kHz, amostrado a uma taxa de 128 kSPS, por conversores de 12 *bits*. Nesse sentido, foi implementada uma memória ROM de 128 x 12 *bits*.

O esquemático RTL da memória ROM gerado pela ferramenta de síntese é ilustrado na Figura 3.24. Nesta figura, cada bloco representa uma célula básica de memória. Estas células básicas de memória são interligadas, formando a estrutura da memória ROM.

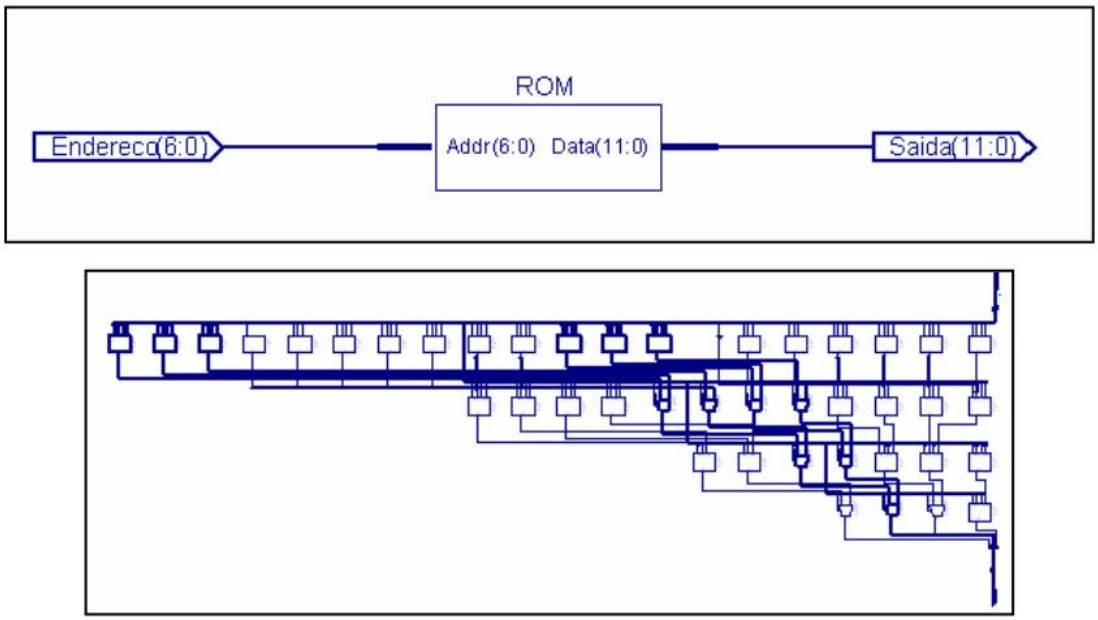


Figura 3.24: Esquemático RTL da memória ROM.

3.4.4 Sequenciador

Para realizar as operações de conversão AD, conversão DA, armazenamento nos registradores, multiplicação, cálculo da média, atualização das saídas X e Y, etc., de forma sincronizada, é necessário um circuito de sequenciamento. Nesse sentido, foi implementada uma máquina de estados finitos, denominada Sequenciador.

Também foram implementados contadores auxiliares, indicando o número de pontos calculados, o número de deslocamentos realizados na divisão e a posição da

memória ROM.

Após a descrição de todos os blocos, uma entidade principal foi criada, realizando as interconexões entre os blocos descritos anteriormente, numa descrição estrutural.

3.4.5 Simulações do Amplificador *Lock-in* Digital em VHDL

Após o desenvolvimento do circuito e a criação do ambiente de teste *(TestBench)*, foram realizadas várias simulações funcionais.

A memória ROM foi preenchida de modo que seus valores correspondessem a uma senóide de amplitude igual a 1V, para o conversor DA de 12 bits escolhido. O conversor DA considerado foi o AD5447 [27], desenvolvido pela *Analog Devices*. Os valores armazenados na ROM são apresentados na Figura 3.25.

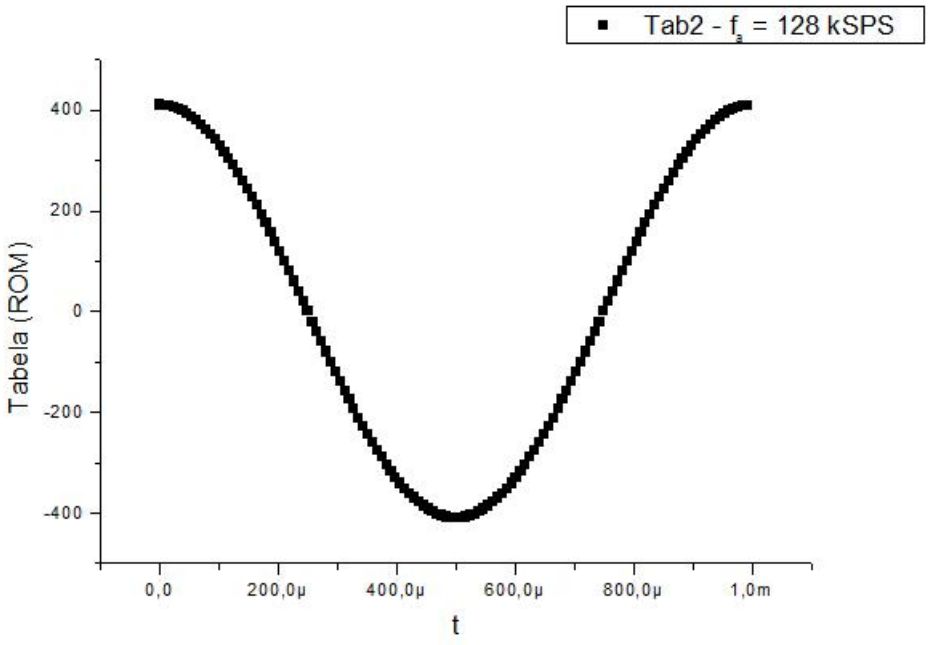


Figura 3.25: Gráfico dos valores armazenados na memória ROM.

Como pode ser observado na Figura 3.25, para $V_{DA} = 1V$(saída do conversor DA), o valor binário armazenado na memória corresponde a 410 (na base decimal). O período do sinal de referência é de 1 ms (frequência de referência de 1 kHz), amostrado a uma taxa de 128 kSPS. Logo, são utilizados 128 pontos na tabela, ou seja, 128 posições de memória.

Nas Figuras 3.26 a 3.28, o funcionamento do amplificador sensível à fase digital é

apresentado, para três situações:

- Sinal de entrada em fase com o sinal de referência;
- Sinal de entrada em quadratura com o sinal de referência;
- Sinal de entrada defasado de 180 graus em relação ao sinal de referência.

Em todas as situações, os sinais de entrada e de referência (ROM) têm a mesma amplitude e frequência.

Consideremos as Equações 3.8 e 3.9 reapresentadas, por comodidade, abaixo:

$$X = \frac{1}{2}ABcos(\theta - \phi), \tag{3.24}$$

$$Y = \frac{1}{2}ABsen(\theta - \phi). \tag{3.25}$$

No primeiro caso, temos que a diferença de fase é zero. Logo, os valores esperados de X e Y são:

$$X = \frac{1}{2}A^2 = \frac{1}{2}(410)^2 = 84050, \tag{3.26}$$

$$Y = 0. \tag{3.27}$$

Como pode ser observado na Figura 3.26, foram obtidos $X = 83992$ e $Y = 0$. Logo, a saída X apresenta um erro de aproximadamente 0,069 %. Este erro é inerente ao algoritmo de divisão utilizado. A divisão consiste no deslocamento para direita de um registrador (divisão por 2). Quando o número armazenado no registrador é par, a divisão é realizada corretamente. Quando o número é ímpar, ocorre uma aproximação levando ao erro observado.

No segundo exemplo, temos que a diferença de fase é de 90 graus. Logo, os valores esperados de X e Y são:

$$X = 0, \tag{3.28}$$

$$Y = \frac{1}{2}A^2 = \frac{1}{2}(410)^2 = 84050. \tag{3.29}$$

Como pode ser observado na Figura 3.27, foram obtidos $X = 0$ e $Y = 83992$. Logo, a saída Y apresenta um erro de aproximadamente 0,069% pelo mesmo motivo discutido anteriormente.

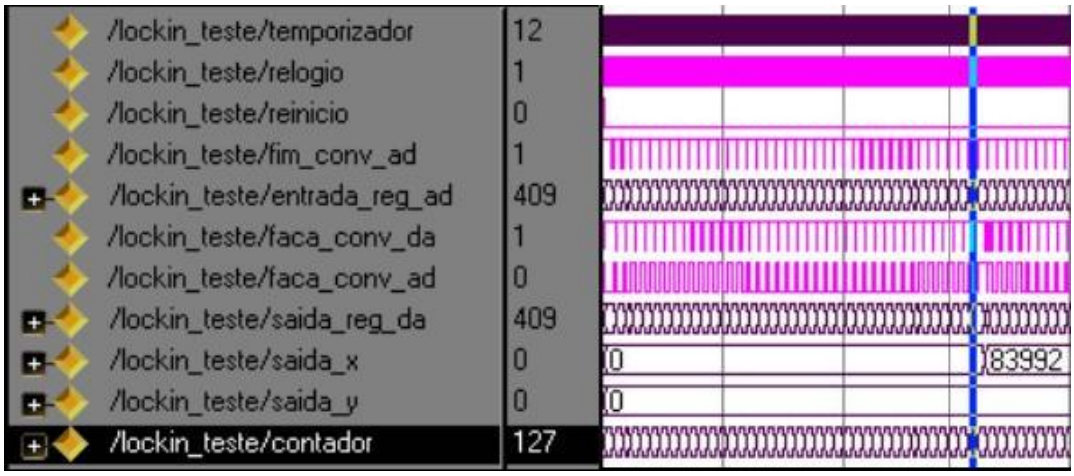


Figura 3.26: Formas de onda obtidas na simulação do amplificador sensível à fase (Sinal de entrada em fase com o sinal de referência).

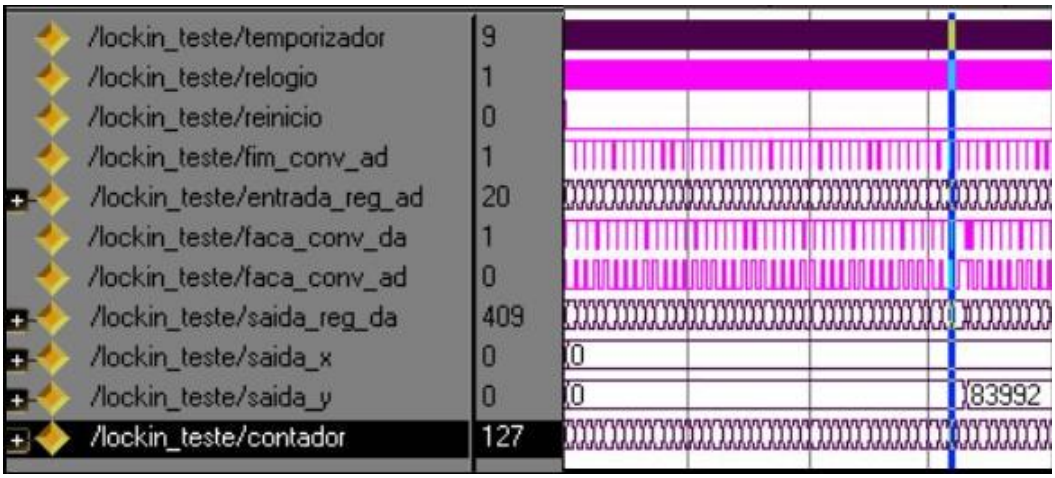


Figura 3.27: Formas de onda obtidas na simulação do amplificador sensível à fase (Sinal de entrada em quadratura com o sinal de referência).

Finalmente, no terceiro caso, temos que a diferença de fase é de 180 graus. Logo, os valores esperados de X e Y são:

$$X = -\frac{1}{2}A^2 = -\frac{1}{2}(410)^2 = -84050, \tag{3.30}$$

$$Y = 0. \tag{3.31}$$

Como pode ser observado na Figura 3.28, foram obtidos $X = -83993$ e $Y = 0$. Novamente a saída X apresenta um erro de aproximadamente 0,069% devido às aproximações realizadas durante a divisão.

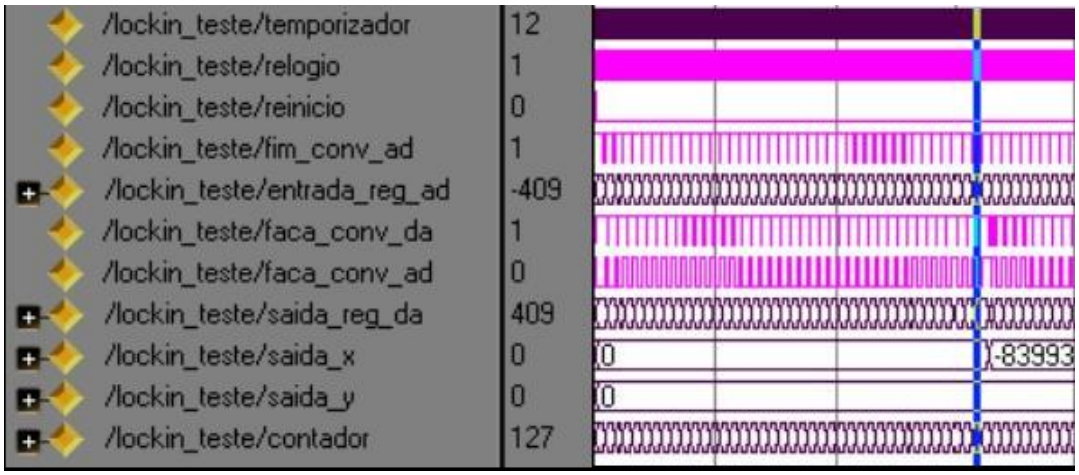


Figura 3.28: Formas de onda obtidas na simulação do amplificador sensível à fase (Sinal de entrada defasado de 180 graus com relação ao sinal de referência).

3.4.6 Síntese do Amplificador *Lock-in* Digital em VHDL

Os resultados obtidos na síntese do amplificador sensível à fase digital descrito em VHDL são apresentados na Tabela 3.2. Neste trabalho foi utilizada a FPGA *Xilinx Spartan 3E XC3S500E -5 FG320*. A partir destes resultados, verifica-se que o amplificador sensível à fase digital ocupa cerca de 6% dos recursos lógicos da FPGA escolhida.

Tabela 3.2: Utilização de recursos lógicos da FPGA após a síntese do *Lock-in*.

	Usado	Disponível	Utilização
Slice Flip-Flops	219	9312	2 %
LUTs de 4 entradas	564	9312	6%
Slices	318	4656	6%

3.4.7 Validação em FPGA do Amplificador *Lock-in* Digital em VHDL

Para validação do amplificador sensível à fase implementado em FPGA, foi utilizada a seguinte metodologia:

- Simular os resultados das medições, gravando numa memória ROM os valores que seriam fornecidos por um conversor AD. Essa memória foi descrita em

VHDL e implementada em FPGA, externamente ao amplificador sensível à fase;

- Realizar o processamento dessas medições com o amplificador sensível à fase;
- Apresentar o resultado desse processamento a partir dos LEDs presentes na placa de desenvolvimento. Uma vez que a placa possui apenas 8 LEDs, o resultado apresentado é o *byte* menos significativo dos canais de saída X e Y.

Na Figura **??** são ilustradas as representações, a partir dos 8 LEDs disponíveis na placa de desenvolvimento, dos *bytes* menos significativos dos sinais de saída dos canais X e Y. Nesse exemplo, a ROM que simula os valores fornecidos pelo conversor AD possui a mesma tabela de valores da ROM que armazena o sinal de referência. Consequentemente, tem-se o caso do sinal de entrada de mesma amplitude, frequência e fase do sinal de referência.

Conforme a Figura 3.26, tem-se $X = 83992_{10} = 10100100000011000_2$ e $Y = 0$. Portanto, as saídas esperadas nos LEDs seriam "00011000"e "00000000", para os canais X e Y, respectivamente. Estes resultados foram obtidos corretamente na placa de desenvolvimento.

3.5 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos na implementação de um amplificador sensível à fase digital utilizando, inicialmente, um microcomputador com placa de aquisição de dados e MATLAB e, em seguida, descrito em VHDL e gravado em FPGA.

Embora o protótipo implementado com a placa DAS-20 e MATLAB tenha apresentado limitações na medição de capacitâncias, o algoritmo de detecção de fase da técnica *lock-in* pôde ser validado. Além disso, este protótipo foi utilizado com êxito na medição de resistências.

Uma alternativa para solucionar a limitação do protótipo na medição de capacitâncias é modificar o circuito de condicionamento, alterando a sua função de transferência e, consequentemente, anulando o efeito derivativo. Outra solução seria utilizar con-

versores DA mais rápidos. Finalmente, pode-se realizar uma filtragem preliminar do sinal de teste, reduzindo as suas descontinuidades.

Uma vez validada a técnica *lock-in*, foi realizada a descrição VHDL de um amplificador sensível à fase digital para implementação em FPGA.

O funcionamento do amplificador *lock-in* digital descrito em VHDL foi verificado com êxito a partir de simulações e de testes realizados após a gravação da FPGA. Entretanto, o circuito final não foi utilizado em medições reais de impedâncias. Para tal, é necessária a integração do amplificador *lock-in* digital em FPGA aos circuitos de conversão AD e DA da placa de desenvolvimento utilizada.

O circuito sintetizado ocupou cerca de 6% dos recursos lógicos da FPGA alvo. Isto significa que vários amplificadores *lock-in* podem ser utilizados em paralelo, num mesmo *chip*. Este é um resultado importante, considerando-se a aplicação deste dispositivo na medição de vazão utilizando a técnica da tomografia por impedância elétrica.

Nesta técnica, são necessárias pelo menos $N(N-1)/2$ medidas para se obter os perfis de condutividade e permissividade da tubulação, onde N é o número de eletrodos. À medida que o número de eletrodos aumenta, aumenta-se o número de medições e, consequentemente, o tempo de resposta do sistema torna-se crítico. A utilização de amplificadores *lock-in* em paralelo torna-se, portanto, uma alternativa de grande interesse.

Capítulo 4

Módulo de Comunicação CAN

Neste capítulo será realizada, inicialmente, uma breve revisão do protocolo de comunicação CAN *(Controller Area Network)*, com ênfase na descrição da camada de enlace de dados. Em seguida, será apresentada a construção de uma rede CAN, na qual cada módulo da rede é implementado utilizando a placa de desenvolvimento comercial SBC28PC, desenvolvida pela *Moditronix Engineering*, e o microntrolador PIC 18F258, desenvolvido pela *Microchip*. Finalmente, serão discutidos a descrição em VHDL e o desenvolvimento em FPGA de um módulo CAN, além da sua integração à rede montada.

4.1 Protocolo de Comunicação CAN

CAN *(Controller Area Network)* é um protocolo de comunicação serial desenvolvido pela empresa alemã Robert BOSCH, em meados dos anos 80 [28]. Foi inicialmente usado na automação de automóveis, com o objetivo de reduzir o cabeamento. Atualmente, o protocolo CAN vem sendo utilizado também em aplicações industriais, em redes de sensores e atuadores, incluindo ambientes intrínsecamente seguros. Além disso, o protocolo CAN também vem sendo implementado em redes sem fio.

O protocolo CAN é padronizado internacionalmente pela ISO 11898. Sua especificação abrange a Camada de Enlace e uma parte da Camada Física, considerando o modelo de referência ISO/OSI [28].

Na descrição da Camada Física, é tratada a forma na qual são transmitidos os *bits*, definindo-se os valores físicos para os *bits* 0 e 1, além dos tipos de cabeamento.

A Camada de Enlace, por sua vez, é subdividida nas camadas de Controle Lógico de Enlace e de Controle de Acesso ao Meio.

Na subcamada de Controle Lógico de Enlace são determinadas as mensagens que devem ser transmitidas no barramento. Também nesta subcamada são definidas quais das mensagens recebidas serão descartadas e quais serão processadas pelo módulo receptor. A descrição desta subcamada está, portanto, mais voltada à aplicação.

O escopo da subcamada de Controle de Acesso ao Meio é o protocolo de transferência das mensagens. Nesta subcamada são realizadas as seguintes funções: o controle dos quadros transmitidos, a execução da arbitragem, a detecção e a correção de erros.

O foco deste trabalho é a Camada de Enlace. Especificamente, o objetivo inicial é a implementação das funções realizadas pela subcamada de Controle de Acesso ao Meio, conforme ilustrado na Figura 4.1.

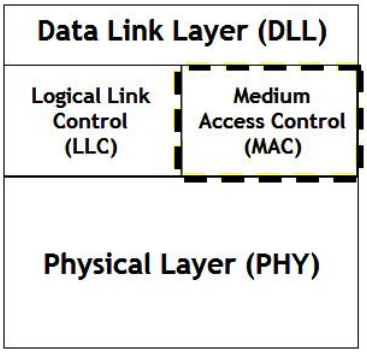


Figura 4.1: Modelo de referência ISO/OSI aplicado ao protocolo CAN.

4.1.1 Camada de Enlace

A Camada de Enlace do protocolo CAN possui as seguintes características principais:

- Mensagens: as mensagens num barramento CAN são enviadas num formato fixo, podendo ter tamanhos diferentes, porém limitados;
- Roteamento de informação: é realizada considerando a estratégia produtor-consumidor. Nenhum módulo no barramento CAN precisa possuir informações sobre a configuração do sistema como, por exemplo, o endereço das estações. A transmissão no protocolo CAN é orientada, portanto, pelo conteúdo da mensagem. Toda mensagem possui um identificador, o qual é único na rede e define

o conteúdo e a prioridade da mensagem. O identificador não indica o destino da mensagem, porém descreve o significado dos dados. Portanto, cada módulo receptor é responsável pela filtragem das mensagens recebidas;

- *Multicast*: todas as mensagens são enviadas para todos os módulos existentes na rede, cabendo a cada módulo decidir se irá ou não utilizar essa informação (modelo de comunicação produtor-consumidor);

- Multimestre: quando o barramento está livre, qualquer unidade conectada pode iniciar a transmissão de uma nova mensagem. Portanto, todos os módulos podem ser tornar mestre num determinado momento e escravo em outro;

- Arbitragem: se dois ou mais módulos iniciam a transmissão de mensagens simultaneamente, o conflito de acesso ao barramento é resolvido a partir de uma arbitragem *bit-a-bit* não destrutiva. Todos os módulos verificam o estado do barramento, analisando se outro módulo está enviando uma mensagem com maior prioridade. Nesse sentido, os módulos comparam os níveis dos *bits* transmitidos e recebidos no barramento. Se esses níveis são iguais, o módulo deve continuar transmitindo. Quando um *bit* recessivo (nível lógico 1) é transmitido e um bit dominante (nível lógico 0) é recebido, o módulo perde a arbitragem e interrompe a transmissão. Após a mensagem de maior prioridade ter sido recebida, o módulo que perdeu a arbitragem reinicia a transmissão, de modo que nenhuma informação é perdida.

O processo de arbitragem numa rede CAN é ilustrado na Figura 4.2. Note que, neste exemplo, o nó 2 de comunicação está transmitindo uma mensagem de maior de prioridade que os nós 1 e 3. O nó 2, portanto, assume o controle do barramento. Os nós 1 e 3, por sua vez, interrompem a transmissão e aguardam até que o barramento esteja novamente livre.

Existem quatro tipos de quadros utilizados no protocolo CAN:

- Quadro de dados: utilizado na transferência de dados de um transmissor para os receptores;

- Quadro remoto: enviado por uma unidade que solicita a transmissão de um quadro de dados com o mesmo identificador;

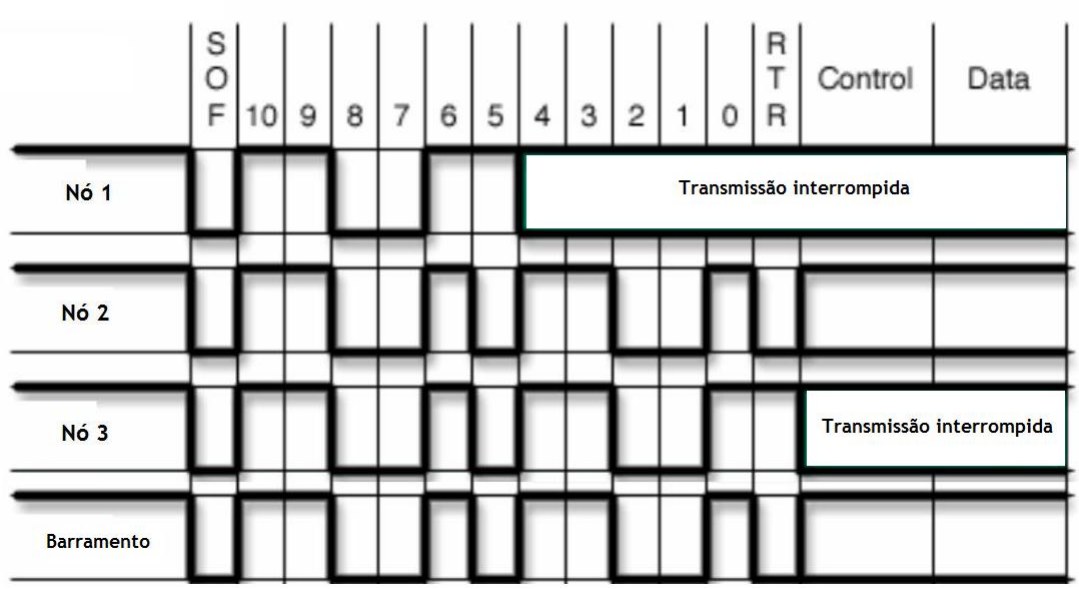


Figura 4.2: Exemplo de arbitragem numa rede CAN.

- Quadro de erro: transmitido por uma unidade que detectou um erro no barramento;
- Quadro de sobrecarga: transmitido para fornecer um retardo extra entre dois quadros de dados (ou remotos) consecutivos.

Neste trabalho é dada ênfase na transmissão e recepção de quadros de dados e remotos.

O protocolo CAN utiliza dois formatos de mensagens que se diferenciam pelo tamanho do campo de arbitragem. O primeiro formato (padrão), denominado CAN 2.0A, possui um identificador de 11 bits. O segundo formato (estendido), denominado CAN 2.0B, possui um identificador de 29 bits. Cada formato é ilustrado nas Figuras 4.3 e 4.4.

	Arbitragem		Controle			Dados	CRC	ACK	Fim	
SOF	IDENTIFIER	RTR	IDE	R0	DLC	DATA	CRC	ACK	EOF	IFS
1	12		6			0 a 64	16	2	7	

Figura 4.3: Quadro padrão com identificador de 11 *bits* (CAN 2.0A).

Cada campo dos quadros das Figuras 4.3 e 4.4 é apresentado a seguir:

	Arbitragem					Controle			Dados	CRC	ACK	Fim	
SOF	11 bits IDENTIFIER	SRR	IDE	18 bits IDENTIFIER	RTR	R1	R0	DLC	DATA	CRC	ACK	EOF	IFS
1	32					6			0 a 64	16	2	7	

Figura 4.4: Quadro estendido com identificador de 29 *bits* (CAN 2.0B).

- SOF *(Start of Frame)*: marca o início dos quadros de dados e remotos. Consiste de um único *bit* dominante (nível lógico 0);
- IDENTIFIER: identificador da mensagem. No formato CAN 2.0A é constituído apenas do identificador padrão de 11 *bits*. Já no formato CAN 2.0B são utilizados mais 18 *bits*, pertencentes ao identificador estendido;
- RTR *(Remote Transmission Request)*: indica se o quadro é de dados (RTR = 0) ou remoto (RTR = 1). Como um *bit* dominante possui maior prioridade em relação a um recessivo, um quadro de dados possui maior prioridade em relação a um quadro remoto de mesmo identificador;
- SRR *(Substitute Remote Request)*: *bit* transmitido apenas no formato estendido (CAN 2.0 B). É sempre enviado como recessivo;
- IDE *(IDentifier Extension)*: indica se a mensagem utiliza o formato padrão (IDE = 0) ou estendido (IDE = 1). Note que uma mensagem no formato padrão possui maior prioridade em relação ao formato estendido;
- R1 e R0: *bits* reservados. Devem ser sempre enviados como *bits* dominantes, embora os módulos receptores aceitem dominante e recessivo em qualquer combinação;
- DLC *(Data Length Code)*: campo formado por quatro *bits* que indicam o número de *bytes* de dados da mensagem. Uma mensagem CAN pode conter até 8 *bytes* de dados (64 *bits*).
- DATA: consiste do dado a ser transmitido. Pode conter de 0 a 64 *bits*;
- CRC *(Cyclical Redundance Check)*: constituído da sequência de CRC calculada (15 *bits*) e de um *bit* delimitador enviado como recessivo;

- ACK *(ACKnowledge)*: formado pelos *bits ACK Slot* e *ACK Delimiter*. *ACK Slot* é sempre enviado como recessivo. Todas as estações que receberam a mensagem corretamente enviam nesse momento um *bit* dominante, validando a mensagem. O *bit ACK Delimiter* é sempre recessivo;

- EOF *(End of Frame)*: sequência formada por 7 *bits* recessivos que indicam o fim de quadro;

- IFS *(Intermission Frame Space)*: número mínimo de *bits* separando mensagens consecutivas.

A detecção de erros em sistemas CAN envolve basicamente os seguintes aspectos:

- Monitoramento: após a escrita de um *bit* dominante, o módulo transmissor verifica o estado do barramento. Se for recebido um *bit* recessivo, significa que houve um erro;

- *Bit Stuffing*: apenas cinco *bits* consecutivos podem ter o mesmo valor (dominante ou recessivo). Caso seja necessário transmitir mais do que cinco *bits* de mesmo valor, o transmissor inserirá um *bit* de valor contrário, o qual será excluído pelo receptor na reconstrução da mensagem;

- Verificação de quadro: os receptores analisam o conteúdo de alguns *bits* da mensagem recebida, os quais não mudam de mensagem para mensagem;

- Reconhecimento de quadro: os receptores escrevem um *bit* dominante no campo de ACK *(Acknowledgement)* da mensagem, em resposta a uma mensagem recebida corretamente.

- CRC *(Cyclic Redundancy Check)*: o transmissor calcula um valor em função dos *bits* da mensagem e o transmite junto com ela. Os receptores recalculam o CRC e verificam se é igual ao recebido.

Para efetuar o cálculo de CRC, é definido um polinômio cujos coeficientes são dados pela sequência formada pelo *bit* de Início de Quadro e pelos campos de Arbitragem, Controle e Dados. Um *Stuff Bit* não é considerado na formação dessa sequência. Este polinômio é dividido pelo seguinte polinômio gerador:

$$X^{15} + X^{14} + X^{10} + X^8 + X^7 + X^4 + X^3 + 1. \quad (4.1)$$

O CRC calculado é, portanto, igual ao resto dessa divisão polinomial [29].

Seja NXTBIT o próximo *bit* da sequência formada pelo *bit* de Início de Quadro até o último *bit* do Campo de Dados. A sequência de CRC, armazenada num registrador de deslocamento de 15 *bits* denominado CRC_RG, pode ser calculada a partir do seguinte algoritmo:

```
CRC_RG = 0;
Repita até que inicie a transmissão do CRC ou ocorra um erro
   CRCNXT = NXTBIT exor CRC_RG(14);
   CRC_RG(14:1) = CRC_RG(13:0);
   CRC_RG(0) = 0;
   Se CRCNXT então
      CRC_RG(14:0) = CRC_RG(14:0) exor (4599hex);
   Fim Se
```

4.1.2 Camada Física

Na Figura 4.5 são apresentados os níveis de tensão num barramento CAN e suas correspondências com as definições de *bits* dominantes e recessivos (níveis lógicos 0 e 1, respectivamente).

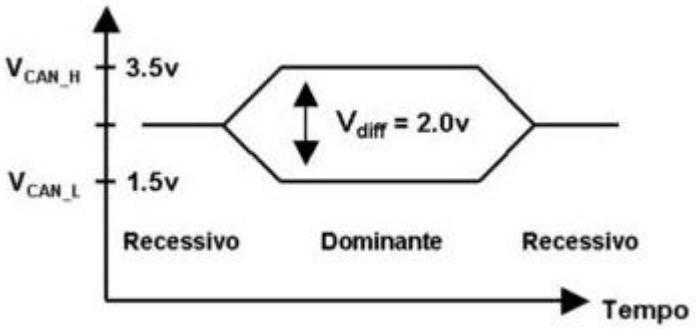


Figura 4.5: Gráfico dos níveis de tensão num barramento CAN.

Para efeitos de temporização e sincronização, um *bit* no protocolo CAN é dividido em quatro segmentos de tempo, ilustrados na Figura 4.6. O comprimento de cada

segmento de tempo é dado por um múltiplo do *time quantum* (T_q). O *time quantum* é uma unidade fixa de tempo, derivada do período do oscilador (sinal de relógio do módulo CAN). A taxa de transmissão nominal dos *bits* no barramento CAN pode ser escolhida na faixa de 125 kbps a 1 Mbps.

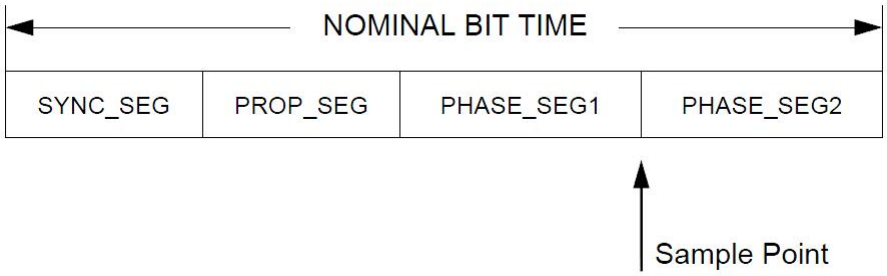


Figura 4.6: Divisão em quatro segmentos de um *bit* no protocolo CAN.

- SYNC_SEG: usado para sincronizar os vários nós no barramento. A ocorrência de uma transição do sinal no barramento é esperada durante este segmento. Possui o comprimento de 1 T_q;
- PROP_SEG: usado para compensar os retardos físicos no barramento. Possui comprimento programável de 1 a 8 T_q;
- PHASE_SEG1: usado para compensar os erros de fase. Possui comprimento programável de 1 a 8 T_q;
- PHASE_SEG2: também usado para compensar os erros de fase. Possui comprimento igual ao máximo entre o PHASE_SEG1 e o tempo de processamento da informação, IPT *(Information Processing Time)*.

O IPT é o segmento de tempo reservado para cálculo do nível do *bit*. Em outras palavras, é o tempo que o módulo CAN leva para detectar se o *bit* lido no barramento possui nível lógico 0 ou nível lógico 1. O IPT inicia no ponto de amostragem do barramento, que ocorre entre o PHASE_SEG1 e o PHASE_SEG2, possuindo comprimento menor ou igual a 2 T_q.

Para compensar os desvios de fase entre os osciladores de cada um dos nós no barramento, cada módulo CAN deve ser capaz de sincronizar-se durante uma transição do sinal no barramento de recessivo para dominante. Existem dois tipos de sincronização:

- Sincronização dura: ocorre sempre durante uma transição de recessivo para dominante no início de quadro;

- Resincronização: durante uma resincronização, o PHSEG1 pode ser aumentado (ou PHSEG2 reduzido). O quanto PHSEG1 é aumentado (ou PHSEG2 reduzido) é definido pelo SJW *(Synchronization Jump Width)*.

Na próxima seção será apresentada a implementação de uma rede CAN utilizando dispositivos comerciais. Em seguida será a discutida a implementação em VHDL de um módulo CAN, considerando os aspectos do protocolo CAN abordados nesta seção introdutória.

4.2 Implementação de uma Rede CAN utilizando Placas SBC28PC e Microcontroladores PIC 18F258

Com o objetivo de realizar um estudo prático do protocolo CAN, foram realizados o projeto, a construção e a montagem de uma rede CAN, composta por dois módulos CAN, utilizando componentes comerciais.

Neste trabalho foi utilizada a placa SBC28PC, desenvolvida pela *Modtronix Engineering*, contendo um microcontrolador PIC 18F258 e um transceptor CAN MCP2551, desenvolvidos pela *Microchip*. O microcontrolador PIC 18F258 já possui um módulo de comunicação CAN integrado. O módulo transceptor CAN é responsável apenas pela conversão dos sinais de 0V e 5V do microcontrolador para os níveis de tensão padronizados pelo protocolo CAN.

Esta rede também foi utilizada na validação do módulo CAN descrito em VHDL e implementado em FPGA, que será apresentado na próxima seção.

Para programação dos microcontroladores PIC foi utilizada a ferramenta de projeto MPLAB 8.36, da *Microchip*. Para gravação e depuração do programa foi utilizado o programador e depurador ICD3, também da *Microchip*.

Para visualização dos sinais no barramento CAN e verificação da comunicação entre dois nós na rede CAN (denominados Nó 0 e Nó 1), foi desenvolvida uma apli-

cação consistindo de dois módulos CAN implementados com duas placas SBC28PC, realizando as seguintes operações:

- O Nó 0 envia um quadro remoto solicitando um dado ao Nó 1;
- O Nó 1 reconhece a solicitação do Nó 0 e envia o dado solicitado (neste exemplo é enviado o caractere ASCII "A");
- O Nó 0, ao receber a resposta do Nó 1, envia o dado recebido ao PC (caractere ASCII "A") através da interface serial RS-232.

Os códigos em linguagem ASSEMBLY utilizados na programação dos microcontroladores PIC 18F258 são disponibilizados no Apêndice C.

A montagem da rede CAN utilizando os módulos implementados com a placa SBC28PC e o microcontrolador PIC 18F258 é apresentada na Figura 4.7. Os dois módulos CAN estão interligados através de um par de fios ao barramento CAN. Um transceptor MCP 2551 extra, montado na matriz de contatos, também foi interligado ao barramento para inserção do módulo CAN implementado em FPGA.

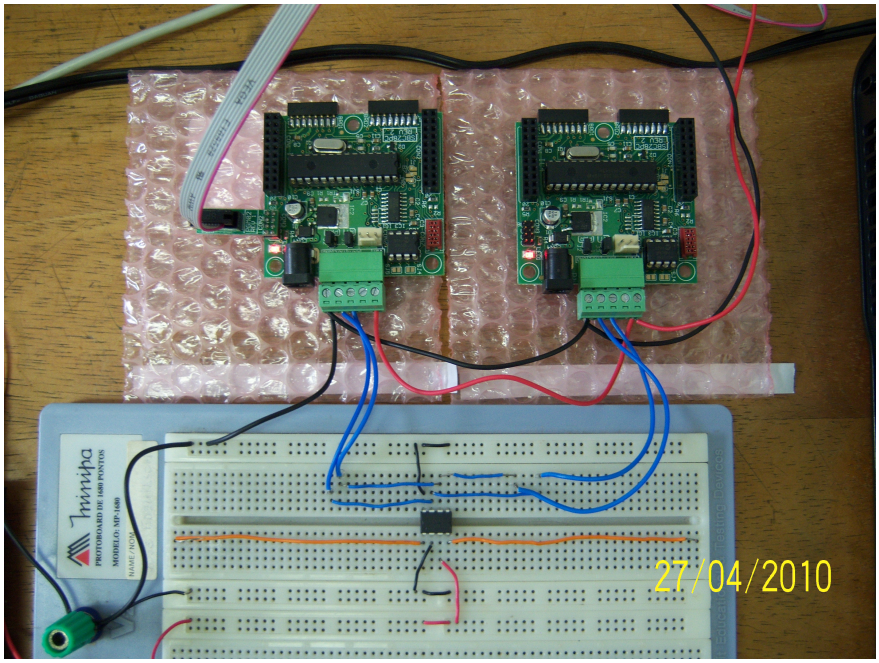


Figura 4.7: Montagem da rede CAN com módulos SBC28PC e microcontroladores PIC 18F258.

O quadro remoto enviado pelo Nó 0 ao barramento CAN, com o Nó 1 desconectado do barramento, é apresentado na Figura 4.8. A desconexão do Nó 1 neste exemplo

visava o não reconhecimento da mensagem transmitida pelo Nó 0, causando retransmissões, pelo Nó 0, desta mensagem, de forma que fosse possível a visualização dos sinais no osciloscópio.

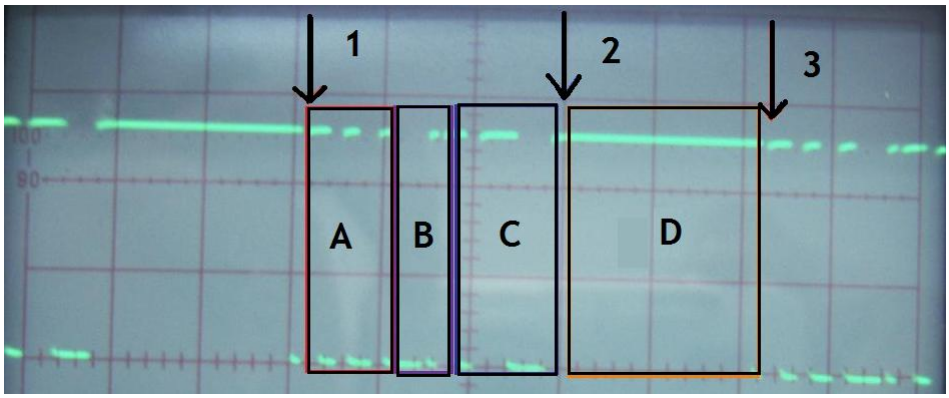


Figura 4.8: Quadro remoto enviado pelo Nó 0 ao barramento CAN com o Nó 1 desconectado.

Na Figura 4.8, as setas de número 1 e 3 indicam o início de quadro. Os retângulos A, B e C representam o identificador de 11 bits, o campo de controle e a sequência de CRC, respectivamente.

Uma vez que o Nó 1 está desconectado do barramento, nenhum módulo irá enviar o reconhecimento da mensagem do Nó 0. Isto é observado no *bit* ACK = 1, indicado pela seta de número 2. Consequentemente, o Nó 0 enviará uma mensagem de erro (sequência de *bits* recessivos), representadada pelo retângulo D e iniciará a retransmissão da mensagem.

Após inserir o Nó 1 na rede, a retransmissão da mensagem CAN pelo Nó 0 foi interrompida, uma vez que a mensagem passou a ser reconhecida. O Nó 1, então, envia do dado solicitado. O Nó 0, por sua vez, recebe a resposta do Nó 1 e envia o campo de dados da mensagem para o PC, via interface serial RS-232. O resultado é apresentado no programa terminal, conforme ilustrado na Figura 4.9.

Na Figura 4.9 são apresentados diversos caracteres ASCII "A". Cada caractere foi recebido após uma reinicialização dos dispositivos presentes na rede CAN. À cada reinicialização, o Nó 0 solicitava um dado e o Nó 1 enviava a resposta (caractere ASCII "A").

O estado dos registradores internos do Nó 0, obtidos com a ferramenta MPLAB e o depurador ICD3 são apresentados na Figura 4.10.

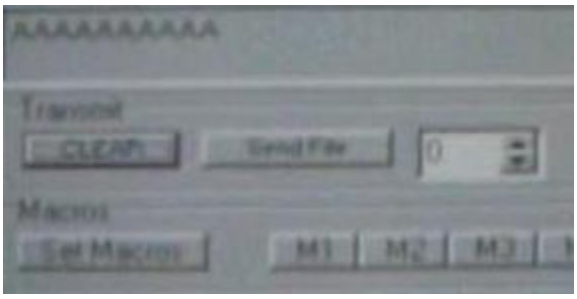


Figura 4.9: Visualização das mensagens transmitidas na rede CAN a partir de um terminal serial no PC.

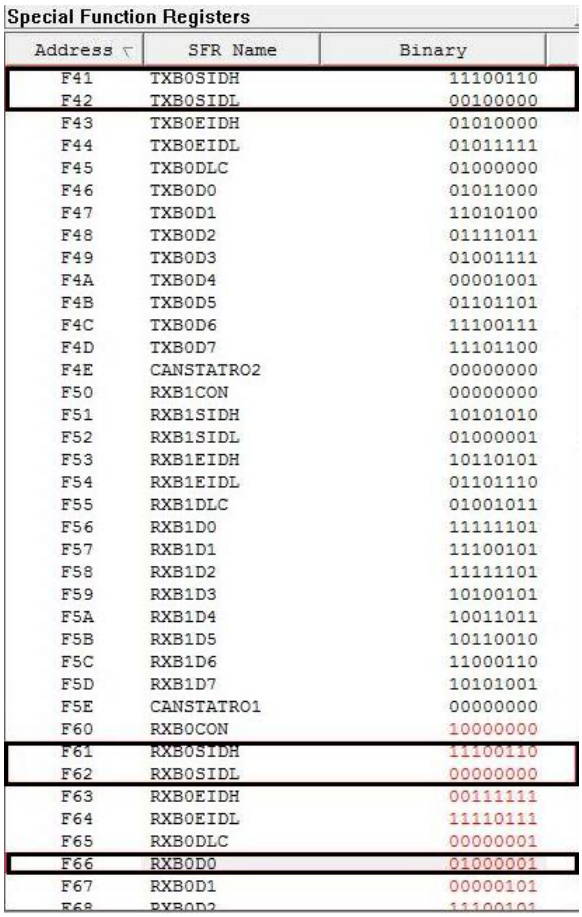
Observa-se, a partir dos endereços F41 e F42, o identificador da mensagem enviada pelo Nó 0: "1110011001". Os endereços F61 e F62 apresentam o identificador da mensagem recebida pelo Nó 0: "1110011000". E, finalmente, o endereço F66 apresenta o dado recebido: "01000001", que representa o caractere "A"no código ASCII.

4.3 Módulo de Comunicação CAN Descrito em VHDL e Implementado em FPGA

O projeto de um módulo de comunicação CAN possui uma complexidade relativamente alta. Por este motivo, foi utilizado novamente o conceito de "dividir para conquistar". Nesse sentido, entidades mais simples foram descritas em VHDL, no estilo RTL, desempenhando as funções básicas do protocolo. Após ser avaliado o funcionamento dessas entidades mais simples, uma nova entidade foi definida, interligando hierarquicamente as entidades básicas, no estilo estrutural.

O diagrama em blocos do módulo CAN é apresentado na Figura 4.11. O módulo é composto por quatro entidades básicas: CAN TX, CAN RX, STUFF HANDLER e CRC. A função de cada entidade é discutida a seguir.

- CAN TX: entidade responsável pelo envio das mensagens;
- CAN RX: entidade responsável pelo empacotamento das mensagens recebidas;
- STUFF HANDLER: entidade responsável pelo gerenciamento do *stuff bit*. Indica às entidades CAN TX e CAN RX a necessidade do envio ou o recebimento de um *stuff bit*;



Special Function Registers

Address	SFR Name	Binary
F41	TXB0SIDH	11100110
F42	TXB0SIDL	00100000
F43	TXB0EIDH	01010000
F44	TXB0EIDL	01011111
F45	TXB0DLC	01000000
F46	TXB0D0	01011000
F47	TXB0D1	11010100
F48	TXB0D2	01111011
F49	TXB0D3	01001111
F4A	TXB0D4	00001001
F4B	TXB0D5	01101101
F4C	TXB0D6	11100111
F4D	TXB0D7	11101100
F4E	CANSTATRO2	00000000
F50	RXB1CON	00000000
F51	RXB1SIDH	10101010
F52	RXB1SIDL	01000001
F53	RXB1EIDH	10110101
F54	RXB1EIDL	01101110
F55	RXB1DLC	01001011
F56	RXB1D0	11111101
F57	RXB1D1	11100101
F58	RXB1D2	11111101
F59	RXB1D3	10100101
F5A	RXB1D4	10011011
F5B	RXB1D5	10110010
F5C	RXB1D6	11000110
F5D	RXB1D7	10101001
F5E	CANSTATRO1	00000000
F60	RXB0CON	10000000
F61	RXB0SIDH	11100110
F62	RXB0SIDL	00000000
F63	RXB0EIDH	00111111
F64	RXB0EIDL	11110111
F65	RXB0DLC	00000001
F66	RXB0D0	01000001
F67	RXB0D1	00000101

Figura 4.10: Visualização do estado dos registradores internos do Nó 0.

- CRC: entidade responsável pelo cálculo da sequência de CRC;
- BIT TIMING 1 e BIT TIMING 2: entidades utilizadas para gerar os sinais de temporização e o ponto de amostragem do barramento CAN.

4.3.1 Simulações do Módulo CAN em VHDL

Esta seção apresenta, inicialmente, os resultados obtidos na simulação das entidades básicas atuando isoladamente. Em seguida, é apresentada a simulação de todos os componentes interligados numa entidade principal.

Na Figura 4.12 são apresentados os resultados obtidos na simulação da entidade BIT TIMING 1. Esta entidade tem a função principal de gerar o ponto de amostragem para a recepção de um *bit* no barramento CAN.

Nesta simulação foram utilizadas as seguintes premissas:

- Frequência do Oscilador *(Clock)*: 50 MHz;

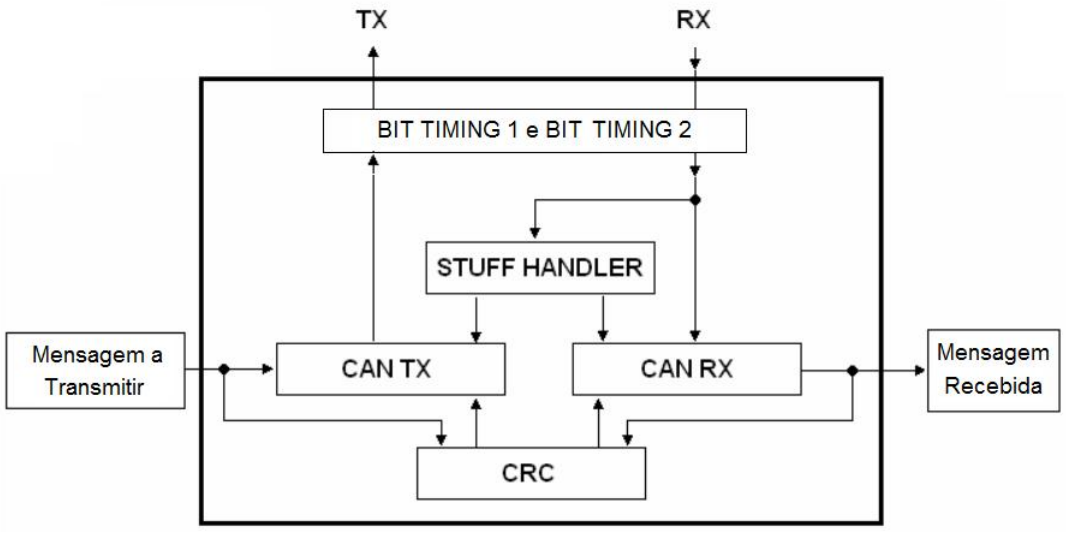


Figura 4.11: Diagrama em blocos do módulo CAN descrito em VHDL.

- *Baud Rate Prescaler* (BRP): 50;
- PROP_SEG (PRSEG): 1 T_q;
- PHASE_SEG1 (PSEG1): 3 T_q;
- PHASE_SEG2 (PSEG2): 3 T_q;
- SJW: 0.

O *time quantum* (T_q) é igual a

$$T_q = \frac{BRP}{Clock} = \frac{50}{50MHz} = 1\mu s. \tag{4.2}$$

O tempo nominal de 1 *bit* é dado por

$$T_{nominal,CAN} = SYNC\_SEG + PRSEG + PSEG1 + PSEG2. \tag{4.3}$$

Logo,

$$T_{nominal,CAN} = (1 + 1 + 3 + 3)T_q = 8T_q = 8\mu s. \tag{4.4}$$

Portanto, a taxa de transmissão é dada por

$$f_{nominal,CAN} = \frac{1}{T_{nominal,CAN}} = \frac{1}{8\mu s} = 125kbps. \tag{4.5}$$

A partir da Figura 4.12, observa-se que o ponto de amostragem *(sample point)* do barramento ocorre entre PHASE_SEG1 e PHASE_SEG2. Além disso, dois pontos de amostragem consecutivos estão separados por 8 T_q ou $1T_{nominal,CAN}$.

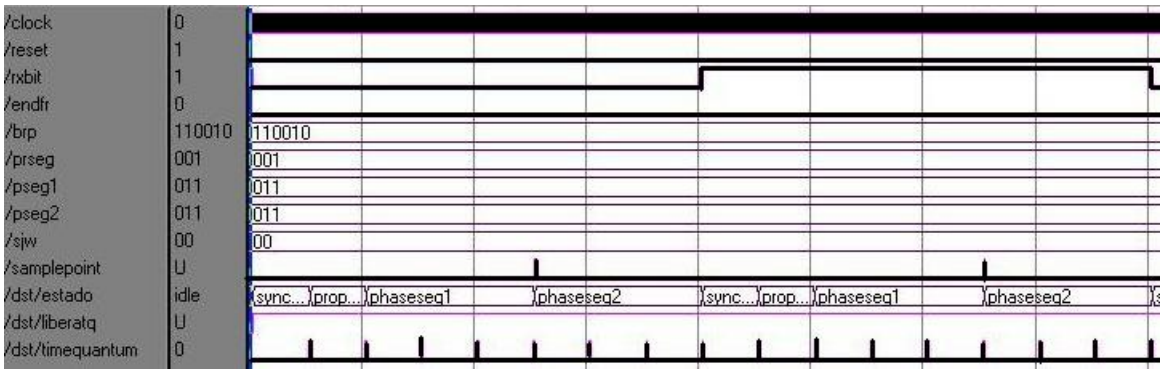


Figura 4.12: Resutados obtidos na simulação da entidade BIT TIMING 1. O ponto de amostragem ocorre entre PHASE_SEG1 e PHASE_SEG2. Dois pontos de amostragem consecutivos estão separados por 8 T_q.

Na Figura 4.13 são apresentados os resultados obtidos na simulação da entidade BIT TIMING 2. Esta entidade tem a função principal de gerar o ponto de transmissão *(txpoint)* para a transmissão de um *bit* no barramento CAN.

Na simulação representada pela Figura 4.13 foram consideradas as mesmas premissas utilizadas anteriormente. Note que dois pontos de transmissão consecutivos estão separados por 8 T_q ou $1T_{nominal,CAN}$.

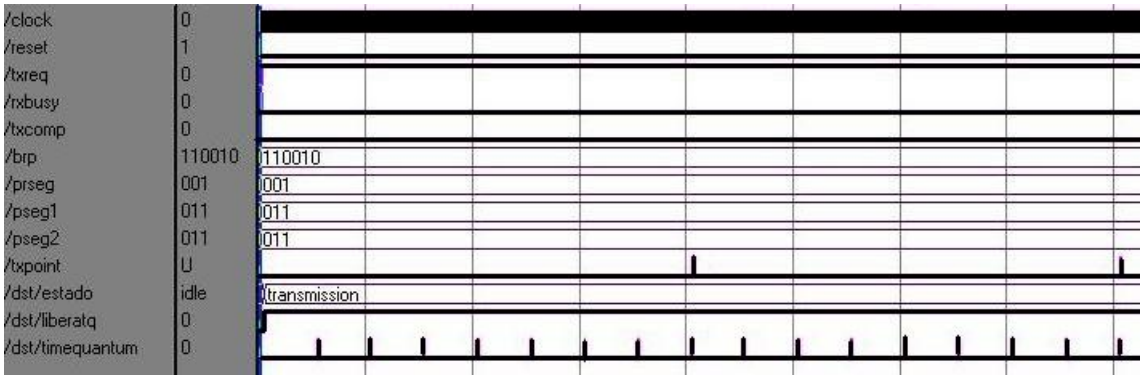


Figura 4.13: Resultados obtidos na simulação da entidade BIT TIMING 2. Dois pontos de transmissão consecutivos estão separados por 8 T_q.

Os resultados obtidos na simulação da entidade STUFF HANDLER são apresentados nas Figuras 4.14 e 4.15.

Conforme ilustrado na Figura 4.14, um *stuff bit* é gerado após a amostragem de cinco *bits* recessivos consecutivos no barramento.

Na Figura 4.15 é ilustrada a ocorrência de um *stuff error*, após a amostragem do sexto *bit* recessivo consecutivo.

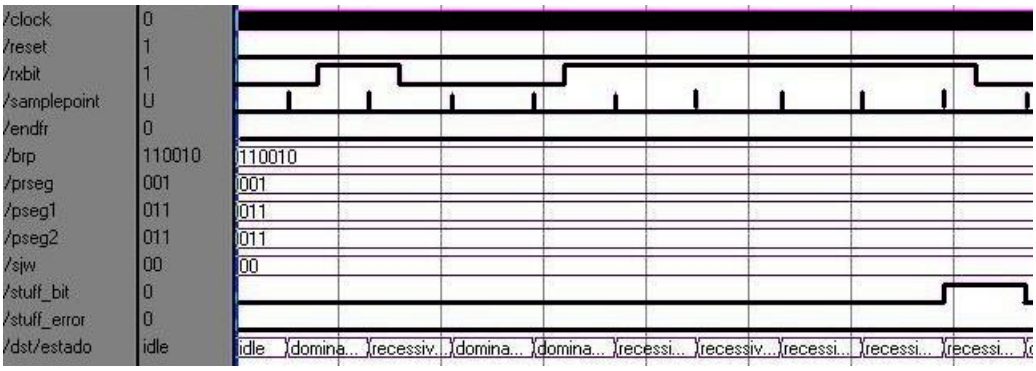


Figura 4.14: Resultados obtidos na simulação da entidade STUFF HANDLER. Um *stuff bit* é gerado após a amostragem de cinco *bits* recessivos consecutivos.

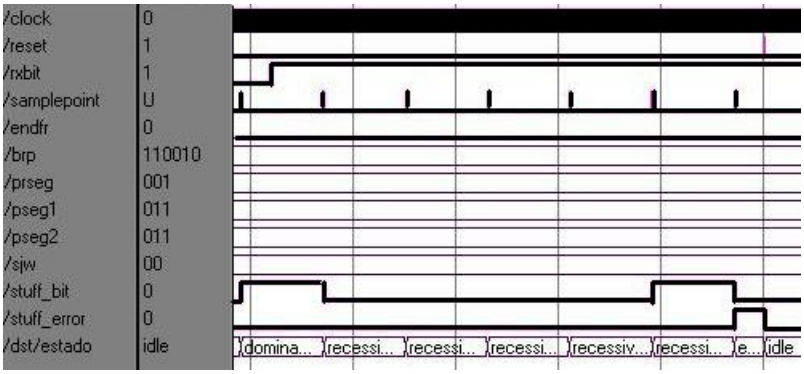


Figura 4.15: Resultados obtidos na simulação da entidade STUFF HANDLER. Um *stuff error* ocorre após a amostragem do sexto *bit* recessivo consecutivo.

Os resultados obtidos na simulação da entidade CRC são apresentados nas Figuras 4.16 e 4.17.

A partir da Figura 4.16 pode-se observar que a sequência de CRC é calculada em cada amostragem do barramento, isto é, após cada *bit* recebido.

O cálculo de CRC é interrompido na ocorrência de um *stuff bit*, conforme ilustrado na Figura 4.17. O cálculo é finalizado quando *crc_stop* é igual a 1.

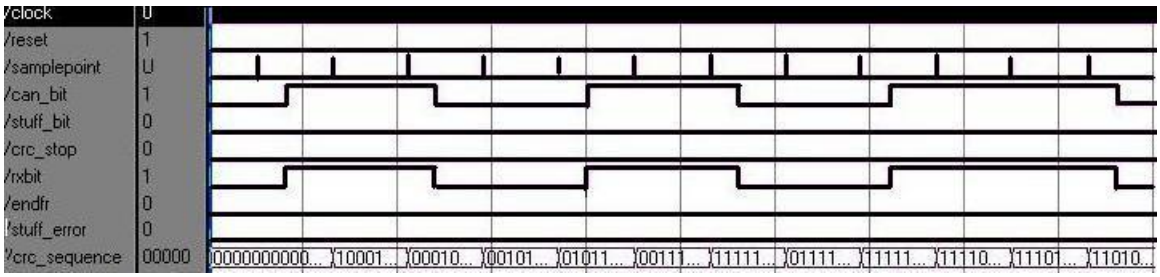


Figura 4.16: Resultados obtidos na simulação da entidade CRC. O cálculo de CRC é realizado após cada *bit* recebido.

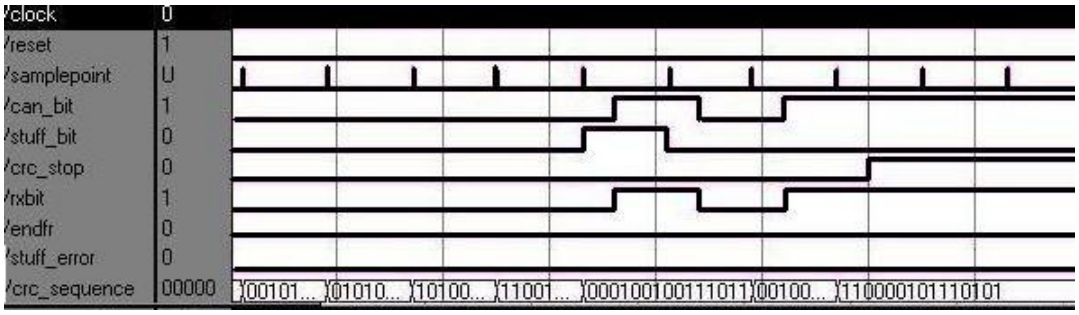


Figura 4.17: Resultados obtidos na simulação da entidade CRC. O cálculo de CRC é interrompido na ocorrência de um *stuff bit*.

Nas Figuras 4.18 e 4.19 são apresentados os resultados obtidos na simulação da entidade CAN RX.

Conforme ilustrado na Figura 4.18, o identificador da mensagem recebida ($id\_rx = 11001100111_2$) é armazenado após doze pontos de amostragem, referentes ao *bit* de início de quadro e aos 11 *bits* do identificador padrão.

O campo de dados da mensagem recebida é composto por 1 *byte* apenas ($msg\_rx = 000111010_2$), conforme ilustrado na Figura 4.19. O CRC calculado é dado por $crc = 010011001101000_2$.

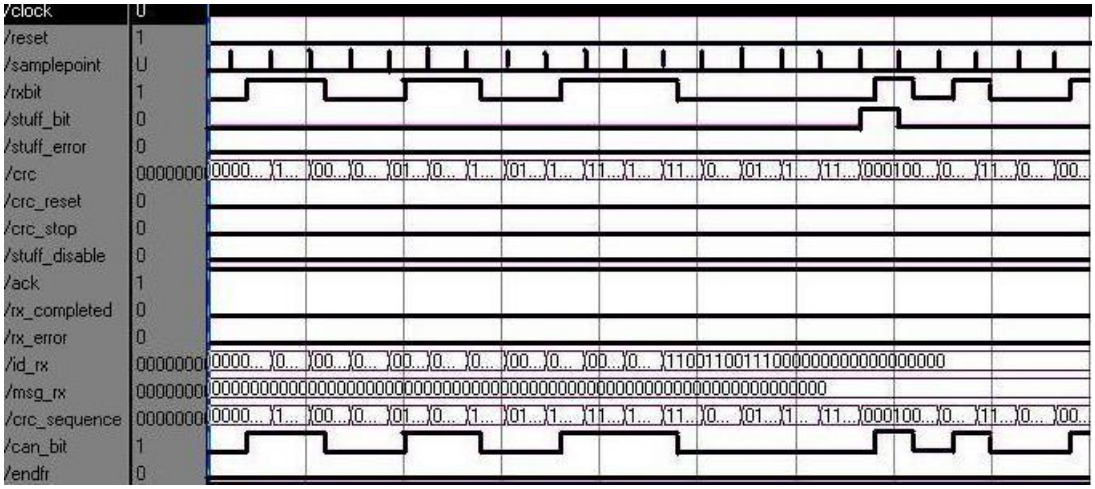


Figura 4.18: Resultados obtidos na simulação da entidade CAN RX. O identificador da mensagem recebida ($id\_rx = 11001100111_2$) é armazenado após doze pontos de amostragem.

Finalmente, os resultados obtidos na simulação da entidade CAN TX são apresentados nas Figuras 4.20, 4.21 e 4.22. Nesta simulação, foram instanciados um módulo transmissor, contendo a entidade CAN TX e um módulo receptor, contendo uma entidade CAN RX. Tanto o transmissor quanto o receptor também possuiam

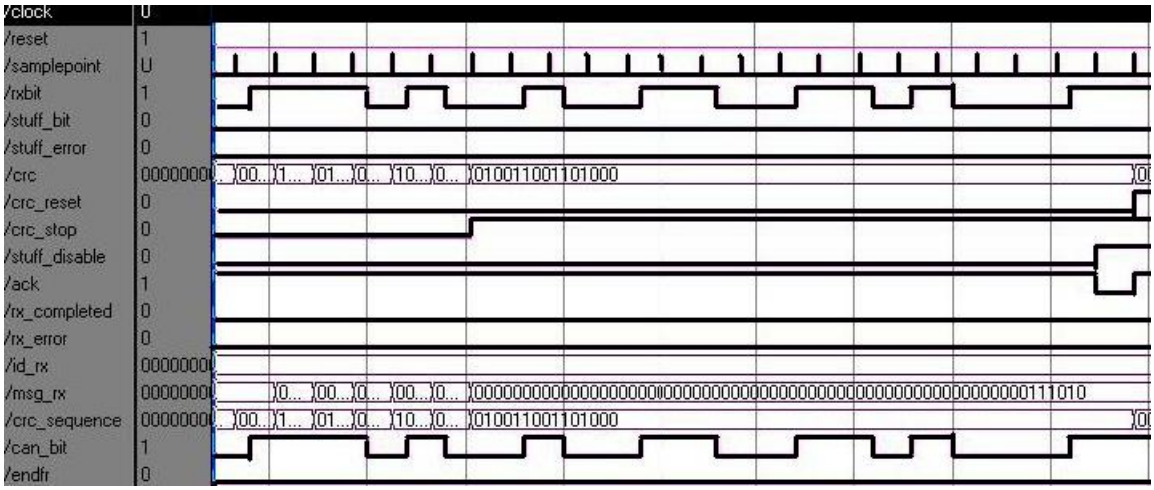


Figura 4.19: Resultados obtidos na simulação da entidade CAN RX. Mensagem recebida: $msg\_rx = 000111010_2$. CRC calculado: $crc = 010011001101000_2$.

componentes formados pelas entidades BIT TIMING 1, BIT TIMING 2, STUFF HANDLER e CRC.

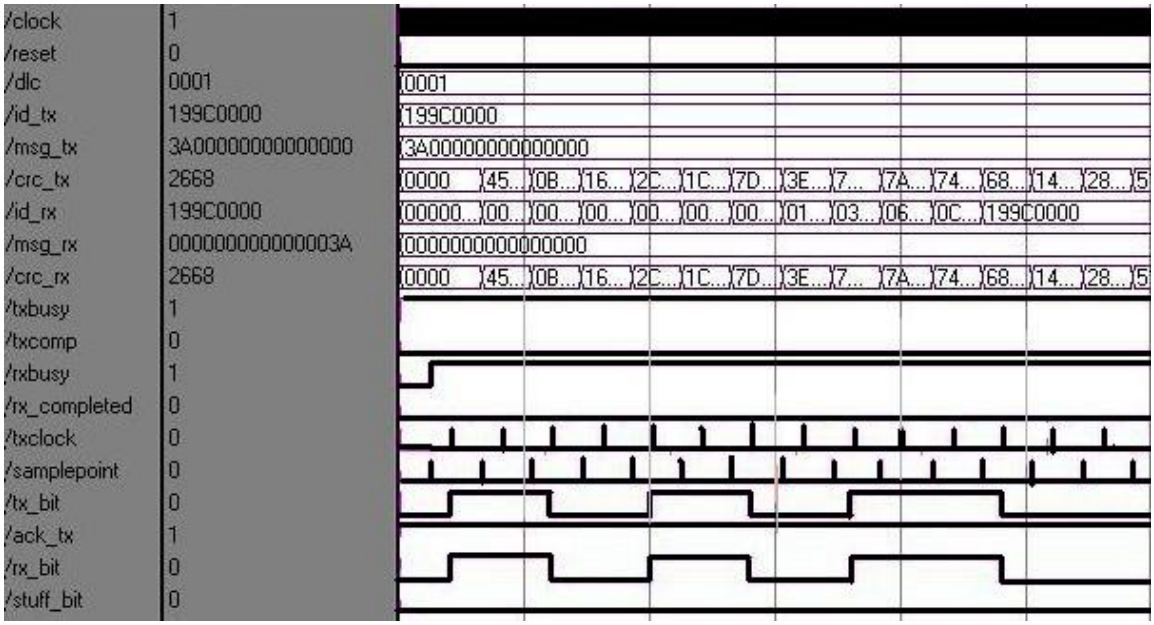


Figura 4.20: Resultados obtidos na simulação da entidade CAN TX. Identificador a ser transmitido: $id\_tx = 199C000_{16}$. Dado a ser transmitido: $msg\_tx = 3A_{16}$. CRC a ser transmitido: $crc\_tx = 2668_{16}$.

A mensagem a ser transmitida possui identificador $id\_tx = 199C000_{16}$ e campo de dados composto por um *byte*, $msg\_tx = 3A_{16}$. O CRC calculado pelo módulo transmissor é $crc\_tx = 2668_{16}$, conforme ilustrado na Figura 4.20.

Na Figura 4.21 pode ser observado que o identificador, o campo de dados e o CRC

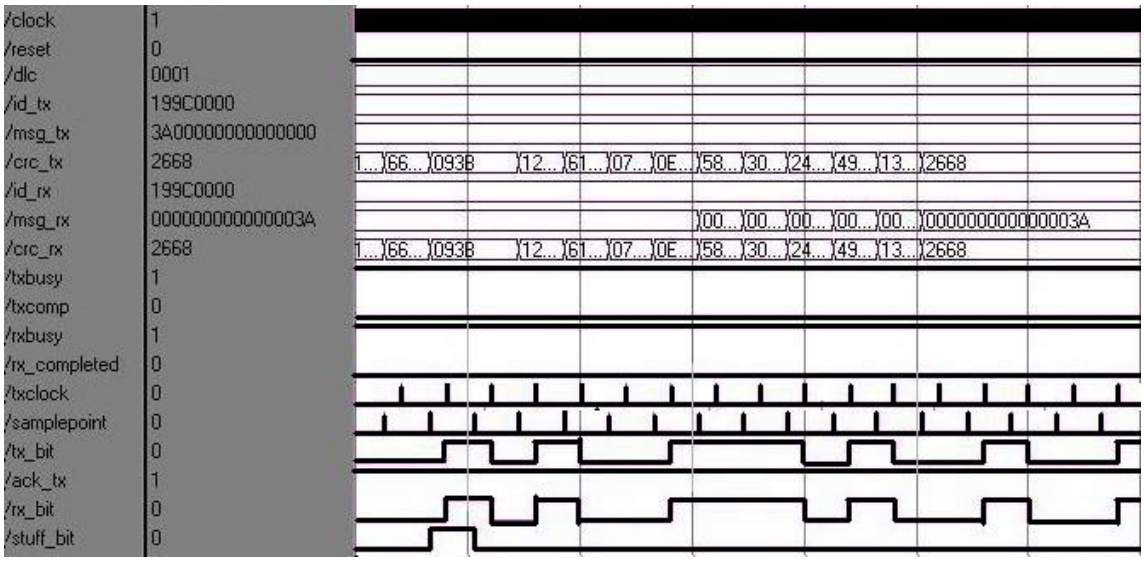


Figura 4.21: Resultados obtidos na simulação da entidade CAN TX. Identificador recebido: $id\_tx = 199C000_{16}$. Dado recebido: $msg\_tx = 3A_{16}$. CRC recebido: $crc\_tx = 2668_{16}$.

recebidos são iguais aos transmitidos. Além disso, conforme ilustrado na Figura 4.22, um *bit* de reconhecimento é enviado, $ack\_tx = 0$, validando a transmissão.

Após a validação de todos os blocos individuais, uma entidade principal foi criada, interligando os componentes de forma estrutural. Em seguida, foi realizada a simulação de uma rede CAN contendo três nós de comunicação. Os resultados desta simulação são apresentados nas Figuras 4.23 e 4.24.

Inicialmente, o Nó 0 envia um quadro remoto solicitando um dado do Nó 1, conforme ilustrado na Figura 4.23. Tanto o Nó 1 quanto o Nó 2 validam a mensagem transmitida pelo Nó 0, enviando um *bit* de reconhecimento (ACK = 0). Em seguida, o Nó 1 envia o dado solicitado.

Ao receber a resposta do Nó 1, o Nó 0 envia um quadro remoto solicitando um dado ao Nó 2, conforme ilustrado na Figura 4.24. Novamente, tanto o Nó 1 quanto o Nó 2 validam a mensagem do Nó 0, enviando um *bit* de reconhecimento (ACK = 0). Finalmente, o Nó 2 envia o dado solicitado.

4.3.2 Síntese do Módulo CAN em VHDL

Os resultados obtidos na síntese do módulo de comunicação CAN são apresentados a seguir. Neste trabalho foi utilizada a FPGA *Xilinx Spartan-3E XC3S500E -5 FG320*.

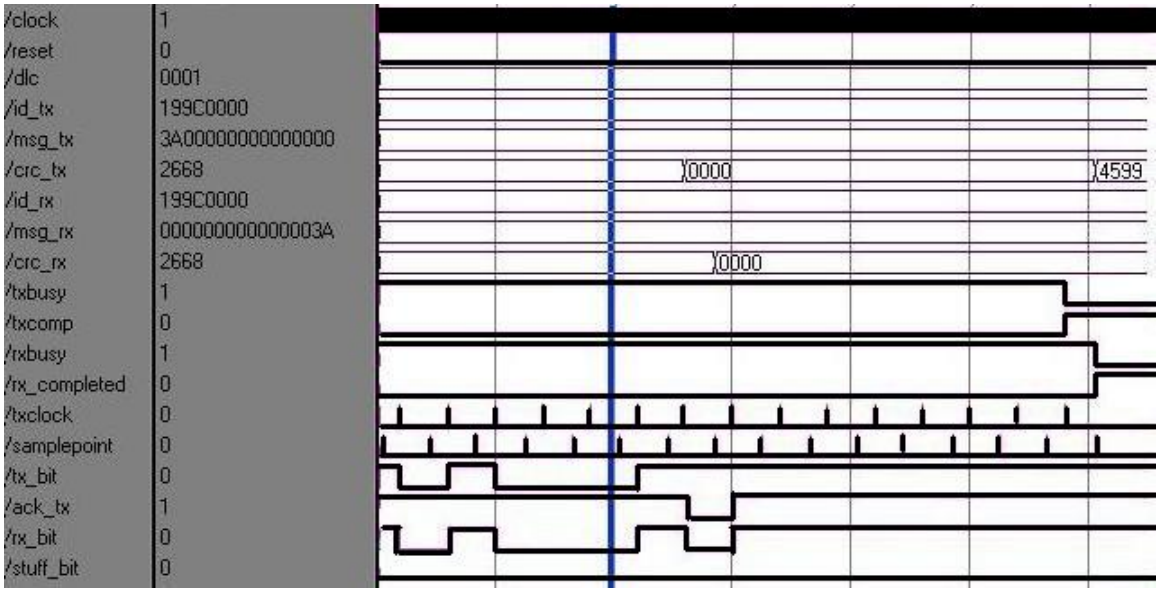


Figura 4.22: Resultados obtidos na simulação da entidade CAN TX. Um *bit* de reconhecimento é enviado, $ack\_tx = 0$, validando a transmissão.

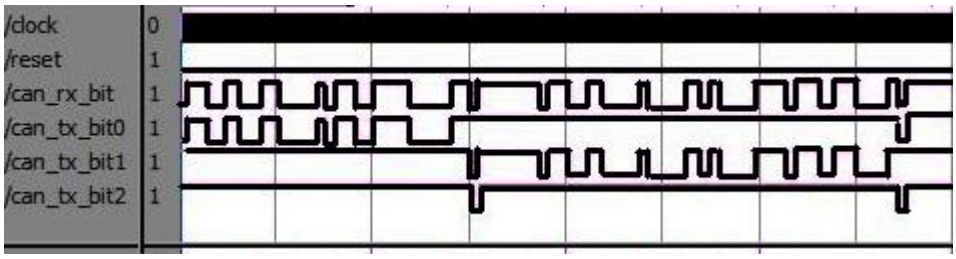


Figura 4.23: Resultados obtidos na simulação de uma rede CAN com três nós. O Nó 0 envia um quadro remoto solicitando um dado do Nó 1. Em seguida, o Nó 1 envia o dado solicitado.

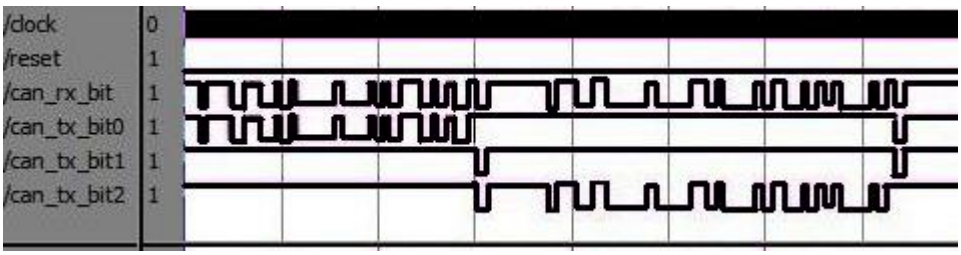


Figura 4.24: Resultados obtidos na simulação de uma rede CAN com três Nós. O Nó 0 envia um quadro remoto solicitando um dado do Nó 2. Em seguida, o Nó 2 envia o dado solicitado.

Tabela 4.1: Utilização de recursos lógicos da FPGA após a síntese do módulo CAN em VHDL.

	Usado	Disponível	Utilização
Slice Flip-Flops	246	9312	2%
LUTs de 4 entradas	507	9312	5%
Slices	271	4656	5%

Tabela 4.2: Comparativo entre as entidades descritas no módulo de comunicação CAN implementado neste trabalho (A) e as entidades correspondentes no Módulo *HurriCANe* (B), desenvolvido pela ESA *(European Space Agency)*. O critério de avaliação usado foi a utilização lógica do dispositivo.

	Slices		*Slice FFs*		*LUTs*	
	A	B	A	B	A	B
BIT TIMING 1	31	31	22	22	58	56
STUFF HANDLER	9	6	13	6	15	9
CRC	9	21	15	31	8	27
CAN RX	141	264	131	258	258	202
CAN TX	203	99	144	113	394	188
TOTAL	393	421	325	430	733	482

De acordo com a Tabela 4.1, o módulo de comunicação CAN desenvolvido neste trabalho utilizou apenas cerca de 5% dos recursos lógicos da FPGA escolhida.

No sentido de avaliar a alocação de recursos no projeto desenvolvido, foi realizado um comparativo, ilustrado na Tabela 4.2, entre as entidades descritas no módulo de comunicação CAN implementado neste trabalho, o qual será integrado futuramente à interface de rede MARIA (Módulo de Acesso à Rede para Instrumentação Avançada), e as entidades correspondentes no Módulo *HurriCANe* [47], desenvolvido pela ESA *(European Space Agency)*. O critério de avaliação usado foi a utilização lógica do dispositivo. O código VHDL do Módulo *HurriCANe*, no passado disponibilizado gratuitamente, não estava mais à disposição durante a execução deste trabalho, de forma que os resultados da síntese deste módulo foram obtidos na referência [49].

Como pode ser observado na Tabela 4.2, a entidade BIT TIMING 1 do módulo desenvolvido neste trabalho utiliza praticamente a mesma quantidade de recursos lógicos da entidade correspondente no módulo *HurriCANe*, com uma pequena desvantagem no número de LUTs *(Look-Up Tables)*. A entidade BIT TIMING 2 não possui cor-

respondente no módulo *HurriCANe*.

As entidades STUFF HANDLER e CAN TX desenvolvidas neste trabalho utilizam um número maior de recursos lógicos, comparado ao módulo *HurriCANe*. Em compensação, as entidades CRC e CAN RX utilizam uma quantidade de recursos lógicos significativamente menor, comparadas às entidades correspondentes no módulo *HurriCANe*.

De modo geral, neste trabalho foi utilizada uma menor quantidade de *Slices* e *Slices Flip-Flops*. Em contrapartida, o módulo *HurriCANe* utiliza um menor número de LUTs de 4 entradas.

4.3.3 Validação em FPGA do Módulo CAN em VHDL

Para validação do módulo CAN descrito em VHDL e implementado em FPGA, foi utilizada a rede CAN formada pelos módulos SBC28PC e microcontroladores PIC 18F258.

Na aplicação desenvolvida para teste do dispositivo, o módulo CAN implementado em FPGA é responsável por enviar um *byte* de dados para um nó da rede implementado com o microcontrolador PIC através do barramento CAN. Este nó, por sua vez, envia o *byte* de dados recebido pelo PIC para um PC via porta serial (RS-232).

A montagem experimental do circuito é apresentada na Figura 4.25.

Nesse exemplo, foi utilizado um quadro de dados com identificador padrão representado, em binário, por "11001100111". O dado transmitido é representado em binário por "00111010". Os resultados obtidos são apresentados na Figura 4.26.

Na Figura 4.26, as setas 1 e 3 indicam o início de um novo quadro de dados. Já a seta 2 representa o *bit* de reconhecimento, ACK = 0, gerado pelos módulos PIC. Por sua vez, cada retângulo representa um campo da mensagem, conforme descrito a seguir:

- A: identificador da mensagem;
- B: campo de controle;
- C: campo de dados;
- D: campo de CRC;

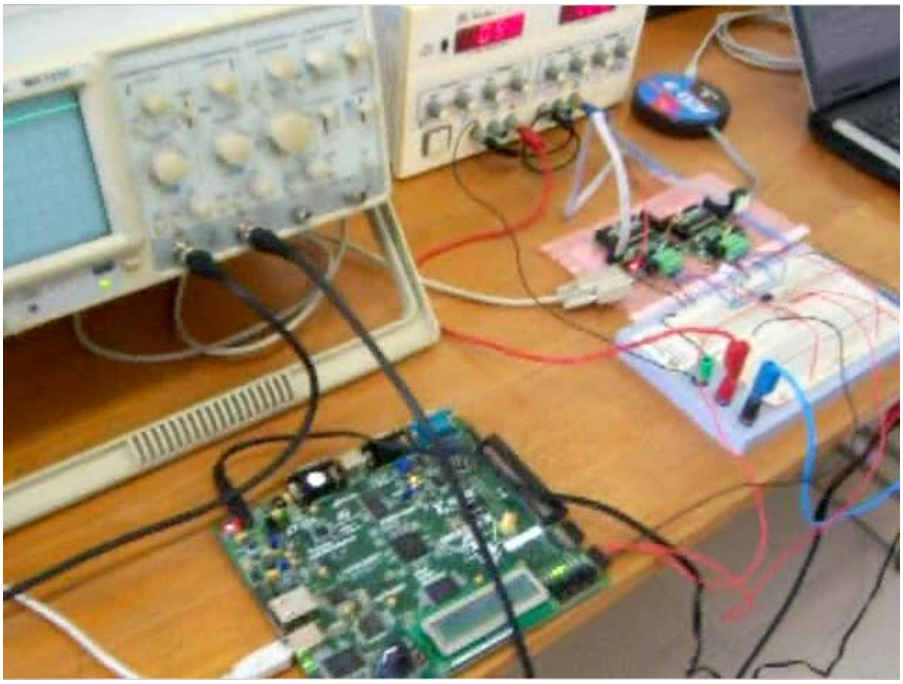


Figura 4.25: Montagem experimental da rede CAN com placas SBC28PC e microcontroladores PIC 18F258, incluindo o módulo CAN implementado em FPGA.

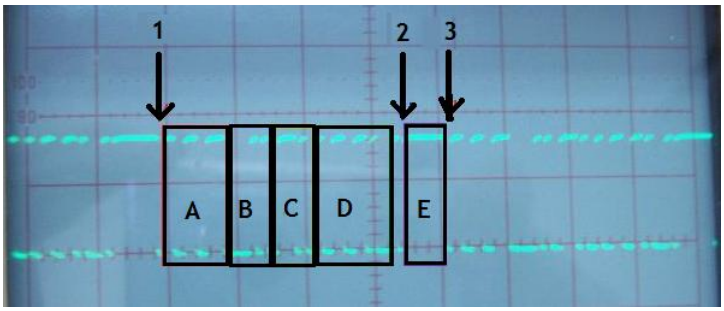


Figura 4.26: Forma de onda obtida com o osciloscópio de um quadro de dados após implementação do módulo CAN em FPGA.

- E: fim da mensagem.

4.4 Implementação em VHDL de Sensor Inteligente com Módulo CAN e Amplificador Sensível à Fase

Após descrição em VHDL, exaustivas simulações, síntese, implementação e verificação do funcionamento de cada bloco em FPGA, foi realizada a integração do amplificador sensível à fase digital, apresentado no Capítulo 3, ao módulo de comunicação CAN, abordado neste capítulo. Os resultados obtidos são discutidos nesta seção.

4.4.1 Síntese do Sensor Inteligente Implementado em FPGA

Os resultados da síntese do sensor inteligente são apresentados na Tabela 4.3. Como pode ser observado, o sensor utilizou cerca de 14% dos recursos lógicos da FPGA.

Tabela 4.3: Utilização de recursos lógicos da FPGA após a síntese do sensor inteligente em VHDL.

	Usado	Disponível	Utilização
Slice Flip-Flops	502	9312	5%
LUTs de 4 entradas	1177	9312	12%
Slices	652	4656	14%

4.4.2 Validação do Sensor Inteligente Implementado em FPGA

Para validação do sensor inteligente implementado em FPGA, foi criada uma aplicação na qual os resultados das medições efetuadas pelo amplificador sensível à fase são enviados através da rede de comunicação CAN.

Conforme discutido no Capítulo 3, não foi possível realizar medições reais com o amplificador *lock-in* digital em FPGA, uma vez que a integração deste dispositivo aos conversores AD e DA da placa de desenvolvimento não foi efetuada. Portanto, para simular o sinal de entrada do amplificador sensível à fase, os valores que seriam fornecidos por um conversor AD foram gravados numa memória ROM, descrita em VHDL e implementada na FPGA, externa ao sensor.

Em seguida, os valores gravados nesta memória ROM são processados pelo amplificador sensível à fase e os *bytes* menos significativos das palavras de 64 *bits* correspondentes às saídas X e Y do amplificador *lock-in* são enviados através da rede CAN para um módulo implementado com a placa SBC28PC e o microcontrolador PIC 18F258. Este módulo, por sua vez, envia os resultados obtidos para um PC através da porta serial. Finalmente, os resultados são exibidos no programa Terminal.

Na Figura 4.27 são apresentados os valores dos *bytes* menos significativos dos sinais de saída dos canais X e Y. Neste caso, o sinal de entrada possui mesma amplitude, frequência e fase do sinal de referência.

Conforme discutido no Capítulo 3, as saídas esperadas são $X = 83992_{10} = 10100100000011000_2$ e $Y = 0$. Consequentemente, os valores dos *bytes* menos sig-

nificativos dos sinais de saída dos canais X e Y são "00011000"e "00000000", respectivamente.

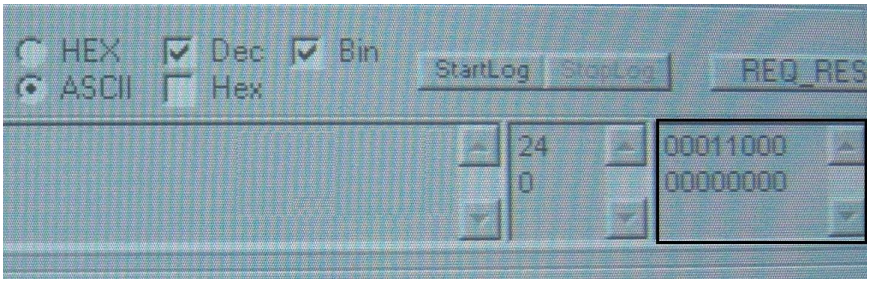


Figura 4.27: Representação, a partir do programa Terminal, dos *bytes* menos significativos dos sinais de saída dos canais X e Y: "00011000"e "00000000", respectivamente.

4.5 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos na implementação de um módulo de comunicação CAN em VHDL e na integração deste módulo ao amplificador sensível à fase digital, abordado no Capítulo 3. Também foi discutida a implementação de uma rede CAN utilizando módulos construídos com placas de desenvolvimento SBC28PC e microcontroladores PIC.

O funcionamento do módulo CAN descrito em VHDL foi validado a partir de simulações e de testes realizados após a gravação da FPGA. Em particular, foi observado a partir da inserção deste módulo à rede CAN implementada, que este dispositivo é capaz de se comunicar com outros módulos CAN, desenvolvidos por fabricantes comerciais.

O circuito sintetizado ocupou cerca de 5% dos recursos lógicos da FPGA alvo. Comparado ao módulo *HurriCANe*, desenvolvido pela ESA *(European Space Agency)*, foi observado que, de modo geral, neste trabalho foi utilizada uma menor quantidade de *Slices* e *Slices Flip-Flops*. Em compensação, o módulo *HurriCANe* utiliza um menor número de LUTs de 4 entradas.

A integração do módulo CAN ao amplificador sensível à fase digital foi realizada com sucesso. Além disso, foram ocupados apenas 14% dos recursos lógicos da FPGA. Isto torna possível a futura integração, numa mesma FPGA, destes módulos ao microcontrolador LAMPIÃO e ao módulo de acesso à rede MARIA.

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Conclusões

Este trabalho é uma contribuição para o projeto e o desenvolvimento de sensores inteligentes, na qual é apresentada a arquitetura de um sensor inteligente integrado baseada na família de padrões IEEE 1451. Em particular, neste trabalho são discutidas a descrição em VHDL e a implementação em FPGA dos circuitos de medição e condicionamento de sinais (amplificador sensível à fase) e de comunicação (módulo CAN).

As simulações realizadas e os resultados experimentais obtidos após a implementação em FPGA mostram que o amplificador sensível à fase desenvolvido funciona conforme esperado e pode, portanto, ser aplicado na medição e no condicionamento de sinais, em especial na medição de impedâncias, desde que seja realizada a integração do amplificador *lock-in* digital aos circuitos de conversão AD e DA. Além disso, uma vez que o circuito sintetizado ocupou apenas 6% dos recursos lógicos da FPGA, vários amplificadores podem ser utilizados em paralelo, no mesmo *chip*. Consequentemente, aumenta-se o desempenho das medições num equipamento de tomografia por impedância elétrica para medição de vazão de fluxos multifásicos, uma vez que a medição das tensões em vários eletrodos podem ser feitas paralelamente, reduzindo o tempo de resposta do sistema de medição.

Do mesmo modo, as simulações realizadas e os resultados experimentais obtidos mostram que o módulo CAN, descrito em VHDL e implementado em FPGA, é capaz

de se comunicar em rede com outros módulos CAN, desenvolvidos por fabricantes comerciais. O circuito sintetizado ocupou cerca de apenas 5% dos recursos lógicos da FPGA alvo. O comparativo entre o módulo CAN desenvolvido e o módulo *HurriCANe* demonstra que as descrições em VHDL das entidades STUFF HANDLER e CAN TX podem ser melhoradas, a fim de atingir uma menor utilização de recursos lógicos. Em compensação, as entidades CRC e CAN RX utilizam uma quantidade de recursos lógicos significativamente menor, comparadas às entidades correspondentes no módulo *HurriCANe*.

Finalmente, a integração do módulo CAN ao amplificador sensível à fase digital foi realizada com sucesso, sendo ocupados apenas 14% dos recursos lógicos da FPGA, o que possibilita a futura integração destes módulos ao microcontrolador LAMPIÃO e ao módulo de acesso à rede MARIA, numa mesma FPGA.

Na elaboração deste trabalho foram abordados diversos assuntos das áreas de microeletrônica, projeto de circuitos integrados, sensores, instrumentação, controle de processos, automação industrial, redes de comunicação, entre outros. Esta visão multidisciplinar possibilita que o conhecimento adquirido durante este trabalho possa ser usado em outras implementações.

5.2 Trabalhos Futuros

Algumas sugestões para continuidade do trabalho realizado são apresentadas a seguir:

- Verificar o desempenho do amplificador sensível à fase desenvolvido em FPGA na medição de impedâncias e estimar seus limites de operação: faixa de medição, resolução, etc.;
- Verificar o desempenho do módulo de comunicação CAN desenvolvido em FPGA numa rede com maior número de nós, sujeito a maior tráfego de mensagens e seguindo padrões de certificação internacionais;
- Verificar o desempenho do sensor inteligente proposto aplicado à tomografia por impedância elétrica;
- Implementar o amplificador sensível à fase, o módulo de comunicação CAN, o microcontrolador LAMPIÃO e a interface de rede MARIA numa mesma FPGA;

- Integrar o amplificador sensível à fase, o módulo de comunicação CAN, o microcontrolador LAMPIÃO e a interface de rede MARIA em ASIC.

Apêndice A

Códigos VHDL

Neste apêndice estão listados os códigos VHDL do sensor inteligente desenvolvido e de seus subcircuitos.

A.1 Sensor Inteligente

```
--------------------------------------------------------------------------
-- UFPE  - Universidade Federal de Pernambuco
-- PPGEE - Programa de Pós-Graduação em Engenharia Elétrica
-- LDN   - Laboratório de Dispositivos e Nanoestruturas
--------------------------------------------------------------------------
-- PROJETO     : Sensor Inteligente (Smart Transducer Interface Module)
-- SUBPROJETO  : STIM
-- DESCRIÇÃO   : Módulo principal da arquitetura estrutural do STIM Sensor
-- VERSÃO      : 0.0
-- CRIADO      : 20/05/2010
-- MODIFICADO  : 20/05/2010
-- SIMULADO    : 20/05/2010
-- SINTETIZADO : 20/05/2010
-- IMPLEMENTADO: 20/05/2010
-- TESTADO     : 20/05/2010
-- PROJETISTA  : Eng.  José  E. O. Reges
-- ORIENTADOR  : Prof. Edval J. P. Santos
--------------------------------------------------------------------------
-- COMENTÁRIOS :
-- [1] Módulo principal da arquitetura estrutural do STIM Sensor
-- CONTROLE DE VERSÃO
-- [1] Versão 0.0 - 20/05/2010 - Versão inicial.
--------------------------------------------------------------------------

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity STIM is
generic (N1: integer := 12 ;
         N2: integer := 64);
port (Clock     : in  std_logic;
Reset     : in  std_logic;
LOCK_IN_Canal  : in  std_logic;
CAN_Tx_Req     : in  std_logic;
CAN_Rx_Bit  : in  std_logic;
CAN_Tx_Bit  : out std_logic;
LOCK_IN_Saida  : out std_logic_vector (7 downto 0));
end STIM;

architecture Estrutural of STIM is

-- Declaração dos componentes

-- Módulo CAN

component CAN_Module
port (Clock    : in  std_logic;
Reset    : in  std_logic;
CAN_Rx_Bit : in  std_logic;
CAN_Tx_Req : in  std_logic;
CAN_RTR    : in  std_logic;
CAN_IDE    : in  std_logic;
CAN_DLC    : in  std_logic_vector( 3 downto 0);
CAN_BRP    : in  std_logic_vector( 5 downto 0);
CAN_PRSEG  : in  std_logic_vector( 2 downto 0);
CAN_PSEG1  : in  std_logic_vector( 2 downto 0);
CAN_PSEG2  : in  std_logic_vector( 2 downto 0);
CAN_SJW    : in  std_logic_vector( 1 downto 0);
CAN_ID_Tx  : in  std_logic_vector(28 downto 0);
CAN_MSG_Tx : in  std_logic_vector(63 downto 0);
CAN_ID_Rx  : out std_logic_vector(28 downto 0);
CAN_MSG_Rx : out std_logic_vector(63 downto 0);
CAN_Tx_Busy: out std_logic;
CAN_Tx_Comp: out std_logic;
CAN_Rx_Busy: out std_logic;
CAN_Rx_Comp: out std_logic;
CAN_Tx_Bit : out std_logic);
end component;

-- Amplificador Lock-in
```

```
component Lockin
generic (N1: integer := 12 ;
         N2: integer := 64);
port (Relogio       : in  std_logic                          ;
      Reinicio      : in  std_logic                          ;
Fim_Conv_AD   : in  std_logic                          ;
Entrada_REG_AD: in  std_logic_vector (N1-1 downto 0) ;
Faca_Conv_DA  : out std_logic                          ;
Faca_Conv_AD  : out std_logic                          ;
Saida_REG_DA  : out std_logic_vector (N1-1 downto 0) ;
Saida_X       : out std_logic_vector (N2-1 downto 0) ;
Saida_Y       : out std_logic_vector (N2-1 downto 0));
end component;

-- Simula Conversor AD

component ROM is
port (Endereco : in  STD_LOGIC_VECTOR ( 6 downto 0 );
Saida  : out STD_LOGIC_VECTOR ( 11 downto 0 ));
end component;

-- Declaração dos sinais internos

-- Módulo CAN

signal CAN_RTR    : std_logic := '0';
signal CAN_IDE    : std_logic := '0';
signal CAN_DLC    : std_logic_vector( 3 downto 0) := (others => '0');
signal CAN_BRP    : std_logic_vector( 5 downto 0) := (others => '0');
signal CAN_PRSEG  : std_logic_vector( 2 downto 0) := (others => '0');
signal CAN_PSEG1  : std_logic_vector( 2 downto 0) := (others => '0');
signal CAN_PSEG2  : std_logic_vector( 2 downto 0) := (others => '0');
signal CAN_SJW    : std_logic_vector( 1 downto 0) := (others => '0');
signal CAN_ID_Tx  : std_logic_vector(28 downto 0) := (others => '0');
signal CAN_MSG_Tx : std_logic_vector(63 downto 0) := (others => '0');
signal CAN_ID_Rx  : std_logic_vector(28 downto 0) := (others => '0');
signal CAN_MSG_Rx : std_logic_vector(63 downto 0) := (others => '0');
signal CAN_Tx_Busy: std_logic := '0';
signal CAN_Tx_Comp: std_logic := '0';
signal CAN_Rx_Busy: std_logic := '0';
signal CAN_Rx_Comp: std_logic := '0';

-- Amplificador Lock-in

signal Fim_Conv_AD   : std_logic := '1';
signal Entrada_REG_AD: std_logic_vector (N1-1 downto 0);
```

```
signal Faca_Conv_DA  : std_logic                       ;
signal Faca_Conv_AD  : std_logic                       ;
signal Saida_REG_DA  : std_logic_vector (N1-1 downto 0);
signal Saida_X       : std_logic_vector (N2-1 downto 0);
signal Saida_Y       : std_logic_vector (N2-1 downto 0);

-- Simula Conversor AD

signal Temporizador  : integer range 0 to 19;
signal contador      : std_logic_vector (6 downto 0);

begin

-- Interligação dos componentes

-- Módulo CAN

X2: CAN_Module port map
(
Clock     => Clock,
Reset     => Reset,
CAN_Rx_Bit  => CAN_Rx_Bit,
CAN_Tx_Req  => CAN_Tx_Req,
CAN_RTR     => CAN_RTR,
CAN_IDE => CAN_IDE,
CAN_DLC     => CAN_DLC,
CAN_BRP     => CAN_BRP,
CAN_PRSEG   => CAN_PRSEG,
CAN_PSEG1   => CAN_PSEG1,
CAN_PSEG2   => CAN_PSEG2,
CAN_SJW     => CAN_SJW,
CAN_ID_Tx   => CAN_ID_Tx,
CAN_MSG_Tx  => CAN_MSG_Tx,
CAN_ID_Rx   => CAN_ID_Rx,
CAN_MSG_Rx  => CAN_MSG_Rx,
CAN_Tx_Busy => CAN_Tx_Busy,
CAN_Tx_Comp => CAN_Tx_Comp,
CAN_Rx_Busy => CAN_Rx_Busy,
CAN_Rx_Comp => CAN_Rx_Comp,
CAN_Tx_Bit  => CAN_Tx_Bit
);

-- Amplificador Lock-in

X1: Lockin port map (Clock,
                     Reset,
Fim_Conv_AD,
```

```
Entrada_REG_AD,
Faca_Conv_DA,
Faca_Conv_AD,
Saida_REG_DA,
Saida_X,
Saida_Y);

-- Simula Conversor AD

X0: ROM port map (contador, Entrada_REG_AD);

-- Lógica de Controle do STIM

-- Escolhe o canal de saída do Lock-in que será exibido
nos LEDs da Placa e enviado pelo CAN

process(LOCK_IN_Canal, Saida_X, Saida_Y)
begin
if LOCK_IN_Canal = '0' then
LOCK_IN_Saida (7 downto 0) <= Saida_X (7 downto 0);
CAN_MSG_Tx(63 downto 56)   <= Saida_X (7 downto 0);
else
LOCK_IN_Saida (7 downto 0) <= Saida_Y (7 downto 0);
CAN_MSG_Tx(63 downto 56)   <= Saida_Y (7 downto 0);
end if;
end process;

--CAN_MSG_Tx(63 downto 56)   <= "11001110";

-- Simula fim de conversão AD

process (Clock)
   begin
       if Reset = '1' then
          Temporizador <= 0;
          Fim_Conv_AD  <= '1';
       elsif Clock'event and Clock = '1' then
           if Temporizador = 19 then
              Temporizador <= 0;
              Fim_Conv_AD  <= '0';
           else
              Temporizador <= Temporizador + 1;
              Fim_Conv_AD  <= '1';
           end if;
       end if;
   end process;
```

```
-- Incrementa endereço da ROM que simula sinal de saida do
conversor AD

process(Faca_Conv_AD, Reset)
begin
    if Reset = '1' then
        contador <= (others => '0');
    elsif Faca_Conv_AD'event and Faca_Conv_AD = '1' then
        if contador = "1111111" then
            contador <= (others => '0');
        else
           contador <= contador+1;
        end if;
    end if;
end process;

-- Configura os campos da mensagem a ser transmitida

CAN_RTR <= '0';   -- Transmissão de Quadro de Dados
CAN_IDE <= '0';   -- Transmissão de Identificador Padrão
CAN_DLC <= "0001"; -- Transmissão de 01 Byte de Dados

-- Configura os segmentos de tempo de um bit CAN

-- SyncSeg   = 1 Tq
-- PropSeg   = 1 Tq
-- PhaseSeg1 = 3 Tq
-- PhaseSeg2 = 3 Tq
-- SJW       = 0 Tq

-- Nominal Bit Rate = 125 kbps
-- Nominal Bit Time = 8 us
-- TimeQuantum (Tq) = 1 us
-- Baud Rate (BRP)  = 50

CAN_BRP   <= "110010";
CAN_PRSEG <= "001";
CAN_PSEG1 <= "011";
CAN_PSEG2 <= "011";
CAN_SJW  <= "00" ;

-- Configura mensagem a ser transmitida

CAN_ID_Tx (28 downto 18) <= "11100110000";

end Estrutural;
```

A.2 Amplificador Sensível à Fase

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Lockin is
generic (N1: integer := 12 ;
         N2: integer := 64);
port (Relogio       : in  std_logic                          ;
      Reinicio      : in  std_logic                          ;
Fim_Conv_DA   : in  std_logic                          ;
Fim_Conv_AD   : in  std_logic                          ;
Entrada_REG_AD: in  std_logic_vector (N1-1 downto 0) ;
Faca_Conv_DA  : out std_logic                          ;
Faca_Conv_AD  : out std_logic                          ;
Saida_REG_DA  : out std_logic_vector (N1-1 downto 0) ;
Saida_X       : out std_logic_vector (N2-1 downto 0) ;
Saida_Y       : out std_logic_vector (N2-1 downto 0));
end Lockin;

architecture Estrutural of Lockin is

component Sequenciador is
port (Relogio          : in  std_logic;
      Reinicio         : in  std_logic;
Fim_Conv_DA      : in  std_logic;
Fim_Conv_AD      : in  std_logic;
Limpe_Registradores: out std_logic;
Carregue_REG_DA  : out std_logic;
Faca_Conv_DA     : out std_logic;
Faca_Conv_AD     : out std_logic;
Carregue_REG_AD  : out std_logic;
Carregue_REG_FASE: out std_logic;
Carregue_REG_QUAD: out std_logic;
Multiplique      : out std_logic;
Some             : out std_logic;
Desloque_Soma    : out std_logic;
Carregue_X_Y     : out std_logic;
Endereco         : out std_logic_vector (6 downto 0));
end component;

component ROM is
  port (
    Endereco : in STD_LOGIC_VECTOR ( 6 downto 0 );
    Saida : out STD_LOGIC_VECTOR ( 11 downto 0 ));
end component;
```

```
component Registrador is
generic (N: integer := 12);
port    (Relogio: in  std_logic;
         Limpa  : in  std_logic;
         Entrada: in  std_logic_vector (N-1 downto 0) ;
Saida  : out std_logic_vector (N-1 downto 0));
end component;

component Detetor_Fase is
generic (N    : integer := 12 ;
         N2   : integer := 24);
port (Relogio : in  std_logic;
      Limpa   : in  std_logic;
      Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (2*N-1 downto 0));
end component;

component Filtro_PB is
    generic (N1: integer := 24 ;
             N2: integer := 64);
    port (Relogio       : in  std_logic;
          Limpa_Parcial : in  std_logic;
          Limpa         : in  std_logic;
 Some          : in  std_logic;
 Desloque      : in  std_logic;
 Carregue_Saida: in  std_logic;
          Entrada       : in  std_logic_vector (N1-1 downto 0);
          Saida         : out std_logic_vector (N2-1 downto 0));
end component;

signal Limpe_Registradores, Carregue_REG_DA, Carregue_REG_AD,
Carregue_REG_FASE, Carregue_REG_QUAD,
 Multiplique, Some, Desloque_Soma, Carregue_X_Y : std_logic;

signal Saida_ROM, Saida_REG_AD, Saida_REG_FASE,
Saida_REG_QUAD: std_logic_vector (N1-1 downto 0);
signal Saida_Detector_X, Saida_Detector_Y:
 std_logic_vector (2*N1-1 downto 0);

signal Endereco               : std_logic_vector (6 downto 0);

begin


REG_DA: Registrador generic map (N1) port map (Carregue_REG_DA,
```

```
                                                Reinicio       ,
  Saida_ROM      ,
  Saida_REG_DA  );

REG_AD: Registrador generic map (N1) port map (Carregue_REG_AD,
                                                Reinicio       ,
  Entrada_REG_AD ,
  Saida_REG_AD  );

REG_FASE: Registrador generic map (N1) port map (Carregue_REG_FASE,
                                                  Reinicio         ,
    Saida_ROM        ,
    Saida_REG_FASE  );

REG_QUAD: Registrador generic map (N1) port map (Carregue_REG_QUAD,
                                                  Reinicio         ,
    Saida_ROM        ,
    Saida_REG_QUAD  );

SEQUENCIADOR0: Sequenciador port map (Relogio,
                                      Reinicio,
  Fim_Conv_DA,
  Fim_Conv_AD,
  Limpe_Registradores,
  Carregue_REG_DA,
  Faca_Conv_DA,
  Faca_Conv_AD,
  Carregue_REG_AD,
  Carregue_REG_FASE,
  Carregue_REG_QUAD,
  Multiplique,
  Some,
  Desloque_Soma,
  Carregue_X_Y,
  Endereco);

ROM0: ROM port map (Endereco, Saida_ROM);

DETECTOR_X: Detetor_Fase port map (Multiplique,
                                   Reinicio,
Saida_REG_AD,
Saida_REG_FASE,
Saida_Detector_X);

DETECTOR_Y: Detetor_Fase port map (Multiplique,
                                   Reinicio,
Saida_REG_AD,
```

```
Saida_REG_QUAD,
Saida_Detector_Y);

FILTRO_X: Filtro_PB port map (Relogio,
                              Limpe_Registradores,
                              Reinicio,
Some,
Desloque_Soma,
Carregue_X_Y,
Saida_Detector_X,
Saida_X);

FILTRO_Y: Filtro_PB port map (Relogio,
                              Limpe_Registradores,
                              Reinicio,
Some,
Desloque_Soma,
Carregue_X_Y,
Saida_Detector_Y,
Saida_Y);

end Estrutural;
```

A.2.1 Sequenciador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Sequenciador is
port (Relogio           : in  std_logic;
      Reinicio          : in  std_logic;
Fim_Conv_DA : in std_logic;
Fim_Conv_AD       : in  std_logic;
Limpe_Registradores : out std_logic;
Carregue_REG_DA   : out std_logic;
Faca_Conv_DA      : out std_logic;
Faca_Conv_AD      : out std_logic;
Carregue_REG_AD   : out std_logic;
Carregue_REG_FASE : out std_logic;
Carregue_REG_QUAD : out std_logic;
Multiplique       : out std_logic;
Some              : out std_logic;
Desloque_Soma     : out std_logic;
Carregue_X_Y      : out std_logic;
Endereco          : out std_logic_vector (6 downto 0));
```

```
end Sequenciador;

architecture RTL of Sequenciador is

-- Definição dos estados da máquina seqüencial. A função de cada
-- estado está explícita pelo rótulo utilizado.

type estados is (Reiniciar        ,
                 Carregar_REG_DA  ,
                 Fazer_Conv_DA    ,
  Fazer_Conv_AD    ,
  Carregar_REG_AD  ,
                Carregar_REG_FASE,
                 Carregar_REG_QUAD,
                 Multiplicar      ,
                 Somar            ,
                 Deslocar_Soma    ,
                 Carregar_X_Y    );

signal estado           : estados                       ;
signal contador_7       : std_logic_vector (2 downto 0);
signal contador_128     : std_logic_vector (6 downto 0);
signal contador_endereco: std_logic_vector (6 downto 0);
signal end_reg_fase     : std_logic_vector (6 downto 0);
signal end_reg_quad     : std_logic_vector (6 downto 0);


signal contador_amostragem: std_logic_vector (8 downto 0);
signal Tamostragem : std_logic;

begin

-- Relógio de amostragem

process (Relogio, Reinicio)
begin
if Reinicio = '1' then
contador_amostragem <= (others => '0');
Tamostragem <= '0';
elsif Relogio'event and Relogio = '1' then
if contador_amostragem = "110000111" then -- 391
contador_amostragem <= (others => '0');
Tamostragem <= '1';
else
contador_amostragem <= contador_amostragem+1;
Tamostragem <= '0';
end if;
```

```
end if;
end process;

-- Início da Transição de Estados

process (Relogio, Reinicio)
begin
if Reinicio = '1' then
estado <= Reiniciar;
elsif Relogio'event and Relogio = '1' then
case estado is
when Reiniciar         =>
if Tamostragem = '1' then
estado <= Carregar_REG_DA  ;
end if;
when Carregar_REG_DA   => estado <= Fazer_Conv_DA    ;
when Fazer_Conv_DA     => estado <= Fazer_Conv_AD    ;
if Fim_Conv_DA = '1' then
estado <= Fazer_Conv_AD;
end if;
when Fazer_Conv_AD     =>
if Fim_Conv_AD = '0' then
estado <= Carregar_REG_AD;
end if;
when Carregar_REG_AD   => estado <= Carregar_REG_FASE;
when Carregar_REG_FASE => estado <= Carregar_REG_QUAD;
when Carregar_REG_QUAD => estado <= Multiplicar      ;
when Multiplicar       => estado <= Somar            ;
when Somar             =>
if contador_128 = "1111111" then
estado <= Deslocar_Soma;
else
estado <= Carregar_REG_DA;
end if;
when Deslocar_Soma     =>
if contador_7 = "111" then -- 111
estado <= Carregar_X_Y  ;
end if;
when Carregar_X_Y      => estado <= Reiniciar; -- Carregar_REG_DA  ;
end case;
end if;
end process;

-- Fim da Transição de Estados

-- Início das Equações de Saída
```

```
process (estado)
begin
case estado is
when Reiniciar         =>
Limpe_Registradores <= '1';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= (others => '0');
when Carregar_REG_DA   =>
Limpe_Registradores <= '0';
Carregue_REG_DA     <= '1';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Fazer_Conv_DA     =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '1';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Fazer_Conv_AD     =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '0';
```

```
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Carregar_REG_AD   =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '1';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= end_reg_fase;
when Carregar_REG_FASE =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '1';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= end_reg_quad;
when Carregar_REG_QUAD =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '1';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
```

```
when Multiplicar       =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '1';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Somar             =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '1';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Deslocar_Soma     =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '1';
Carregue_X_Y        <= '0';
Endereco            <= contador_endereco;
when Carregar_X_Y      =>
   Limpe_Registradores <= '0';
Carregue_REG_DA     <= '0';
Faca_Conv_DA        <= '0';
Faca_Conv_AD        <= '1';
Carregue_REG_AD     <= '0';
Carregue_REG_FASE   <= '0';
Carregue_REG_QUAD   <= '0';
```

```
Multiplique         <= '0';
Some                <= '0';
Desloque_Soma       <= '0';
Carregue_X_Y        <= '1';
Endereco            <= contador_endereco;
end case;
end process;

-- Fim das Equações de Saída

-- Início do Contador Mod(7)

-- Usado para realizar o deslocamento da soma para direita 7 vezes,
-- isto é, dividir o total acumulado por 128 (número de pontos).

process (Relogio, Reinicio)
begin
if Reinicio = '1' then
contador_7 <= (others => '0');
elsif Relogio'event and Relogio = '1' then
if estado = Deslocar_Soma then
if contador_7 = "111" then
contador_7 <=  (others => '0');
else
contador_7 <= contador_7 + 1;
end if;
end if;
end if;
end process;

-- Fim do Contador Mod(7)

-- Início do Contador Mod(128)

-- Usado para contar o número de pontos que foram gerados,
            adquiridos e tratados.

process (Relogio, Reinicio)
begin
if Reinicio = '1' then
contador_128 <= (others => '0');
elsif Relogio'event and Relogio = '1' then
if estado = Somar then
if contador_128 = "1111111" then
contador_128 <=  (others => '0');
else
contador_128 <= contador_128 + 1;
```

```
end if;
end if;
end if;
end process;

-- Fim do Contador Mod(128)

-- Início do Contador de Endereços

-- Usado para fornecer o endereço da memória ROM.
-- Como o deslocador de fase ainda não foi implementado,
o Contador de Endereços
-- apresenta sempre o mesmo valor do Contador mod128.
Quando o deslocador de fa-
-- se for implementado, o Contador de Endereços terá um
offset relativo ao erro
-- de fase inerente ao circuito de medição para
calibração do LOCK-IN.

process (Relogio, Reinicio)
begin
if Reinicio = '1' then
contador_endereco <= (others => '0');
elsif Relogio'event and Relogio = '1' then
if estado = Multiplicar then
if contador_endereco = "1111111" then
contador_endereco <=  (others => '0');
else
contador_endereco <= contador_endereco + 1;
end if;
end if;
end if;
end process;

-- Fim do Contador de Endereços

-- Início da Lógica de Endereços

-- Lógica combinacional que calcula os endereços da ROM que serão carregados
-- em REG_FASE e REG_QUAD. O endereço carregado em REG_FASE é dado pelo pró-
-- prio Contador de Endereços. Já o endereço carregado em REG_QUAD apresenta
-- um deslocamento de 32 posições de memória, relativos aos 90 graus de fase.

-- OBS: quando o endereço de REG_FASE = 96, o endereço de REG_QUAD será igual
-- a 96 + 32 = 128 mod 128 = 0. Portanto, a partir do endereço 96, o desloca-
-- mento (negativo) será de 96 posições de memória.
```

```
process (contador_endereco)
begin
if contador_endereco > "1011111" then
end_reg_quad <= contador_endereco - 96;
else
end_reg_quad <= contador_endereco + 32;
end if;
end process;

end_reg_fase <= contador_endereco;

-- Fim da Lógica de Endereços

end RTL;
```

A.2.2 ROM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ROM is
    generic (i    : integer := 7                       ;
             j    : integer := 12)                     ;
    port (Endereco: in  std_logic_vector (i-1 downto 0);
          Saida   : out std_logic_vector (j-1 downto 0));
end ROM;

architecture RTL of ROM is
    type arranjo_memoria is array (0 to 127) of integer range -2048 to 2047;
    constant dados: arranjo_memoria := (410,
                                        409,
 408,
 405,
 402,
 398,
 392,
 386,
 379,
 370,
 361,
 352,
 341,
 329,
 317,
```

```
304,
290,
275,
260,
244,
228,
211,
193,
175,
157,
138,
119,
100,
 80,
 60,
 40,
 20,
    0,
-20,
-40,
-60,
-80,
-100,
-119,
-138,
-157,
-175,
-193,
-211,
-228,
-244,
-260,
-275,
-290,
-304,
-317,
-329,
-341,
-352,
-361,
-370,
-379,
-386,
-392,
-398,
-402,
-405,
```

```
-408,
-409,
-410,
-409,
-408,
-405,
-402,
-398,
-392,
-386,
-379,
-370,
-361,
-352,
-341,
-329,
-317,
-304,
-290,
-275,
-260,
-244,
-228,
-211,
-193,
-175,
-157,
-138,
-119,
-100,
-80,
-60,
-40,
-20,
0,
20,
40,
60,
80,
100,
119,
138,
157,
175,
193,
211,
228,
```

```
 244,
 260,
 275,
 290,
 304,
 317,
 329,
 341,
 352,
 361,
 370,
 379,
 386,
 392,
 398,
 402,
 405,
 408,
 409);

begin

Saida <= conv_std_logic_vector(dados(conv_integer(Endereco(i-1 downto 0))),12);

end RTL;
```

A.2.3 Registrador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Registrador is
generic (N: integer := 12);
port    (Relogio: in  std_logic;
         Limpa  : in  std_logic;
         Entrada: in  std_logic_vector (N-1 downto 0) ;
Saida  : out std_logic_vector (N-1 downto 0));
end Registrador;

architecture RTL of Registrador is

begin

process (Relogio, Limpa)
```

```
begin
if Limpa = '1' then
Saida <= (others => '0');
elsif Relogio'event and Relogio = '1' then
Saida <= Entrada;
end if;
end process;

end RTL;
```

A.2.4 Detector de Fase

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Detetor_Fase is
generic (N    : integer := 12 ;
         N2   : integer := 24);
port (Relogio : in  std_logic;
      Limpa   : in  std_logic;
      Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (2*N-1 downto 0));
end Detetor_Fase;

architecture Estrutural of Detetor_Fase is

component Complemento2 is
generic (N   : integer := 12);
port (Entrada: in  std_logic_vector (N-1 downto 0);
      Saida  : out std_logic_vector (N-1 downto 0));
end component;

component Multiplexador is
generic (N    : integer := 12);
port (Selecao : in  std_logic;
         Entrada0: in  std_logic_vector (N-1 downto 0);
         Entrada1: in  std_logic_vector (N-1 downto 0);
         Saida   : out std_logic_vector (N-1 downto 0));
end component;

component Multiplicador is
generic (N    : integer := 12);
port (Relogio : in  std_logic;
```

```
      Limpa   : in  std_logic;
      Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (2*N-1 downto 0));
end component;

signal fio                : std_logic;
signal cmpl0 , cmpl1      : std_logic_vector (N-1 downto 0);
signal opr0  , opr1       : std_logic_vector (N-1 downto 0);
signal result, cmplresult : std_logic_vector (2*N-1 downto 0);

begin

    fio <= Entrada0(N-1) xor Entrada1(N-1);

X0: Complemento2 port map (Entrada0, cmpl0);

X1: Complemento2 port map (Entrada1, cmpl1);

X2: Multiplexador port map (Entrada0(N-1),
                            Entrada0     ,
                            cmpl0        ,
                            opr0        );

X3: Multiplexador port map (Entrada1(N-1),
                            Entrada1     ,
                            cmpl1        ,
                            opr1        );

X4: Multiplicador port map (Relogio,
                            Limpa  ,
                            opr0   ,
                            opr1   ,
                            result );

X5: Complemento2 generic map (N2)  port map (result, cmplresult);

X6: Multiplexador generic map (N2) port map (fio,
                                             result,
                                             cmplresult,
                                             Saida);

end Estrutural;
```

A.2.5 Complemento2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Complemento2 is
generic (N   : integer := 12);
port (Entrada: in  std_logic_vector (N-1 downto 0);
      Saida  : out std_logic_vector (N-1 downto 0));
end Complemento2;

architecture RTL of Complemento2 is

signal Ent_Inv : std_logic_vector (N-1 downto 0);

begin

Ent_Inv <= not Entrada;
Saida   <= Ent_Inv+1;

end RTL;
```

A.2.6 Multiplexador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Multiplexador is
generic (N    : integer := 12);
port (Selecao : in  std_logic;
        Entrada0: in  std_logic_vector (N-1 downto 0);
Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (N-1 downto 0));
end Multiplexador;

architecture RTL of Multiplexador is

begin

Saida <= Entrada0 when Selecao = '0' else
         Entrada1;

end RTL;
```

A.2.7 Multiplicador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Multiplicador is
generic (N    : integer := 12);
port (Relogio : in  std_logic;
      Limpa   : in  std_logic;
      Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (2*N-1 downto 0));
end Multiplicador;

architecture RTL of Multiplicador is

begin

process (Relogio, Limpa)
begin
   if (Limpa = '1') then
      Saida <= (others => '0');
   elsif (Relogio'event and Relogio = '1') then
   Saida <= Entrada0 * Entrada1;
   end if;
end process;

end RTL;
```

A.2.8 Filtro Passa-Baixa

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Filtro_PB is
    generic (N1: integer := 24 ;
             N2: integer := 64);
    port (Relogio       : in  std_logic;
          Limpa_Parcial : in  std_logic;
 Limpa         : in  std_logic;
 Some          : in  std_logic;
```

```
 Desloque      : in  std_logic;
 Carregue_Saida: in std_logic;
          Entrada       : in  std_logic_vector (N1-1 downto 0);
          Saida         : out std_logic_vector (N2-1 downto 0));
end Filtro_PB;

architecture Estrutural of Filtro_PB is

component Somador is
generic (N    : integer := 64);
port (Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (N   downto 0));
end component;

component Reg_Desl is
generic (N: integer := 64);
port    (Relogio : in  std_logic;
         Limpa   : in  std_logic;
Desloque: in  std_logic;
Entrada : in  std_logic_vector (N-1 downto 0) ;
Saida   : out std_logic_vector (N-1 downto 0));
end component;

component Registrador is
generic (N    : integer := 12);
port (Relogio : in  std_logic;
      Limpa   : in  std_logic;
Entrada : in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (N-1 downto 0));
end component;

signal ent : std_logic_vector (N2-1 downto 0);
signal fio1: std_logic_vector (N2   downto 0);
signal fio2: std_logic                      ;
signal sai : std_logic_vector (N2-1 downto 0);


begin

-- Início Ajuste do sinal demodulado (24 -> 64 bits)

   process (Entrada)
   begin
      if Entrada(N1-1) = '1' then
         ent (N2-1 downto N1) <= (others => '1');
      else
```

```
         ent (N2-1 downto N1) <= (others => '0');
      end if;
   end process;

   ent (N1-1 downto 0)  <= Entrada;

-- Fim Ajuste do sinal demodulado (24 -> 64 bits)

-- Início Componentes

   X0: Somador generic map (N2) port map (ent  ,
                                          sai  ,
                                          fio1);

fio2 <= (Relogio and Desloque) or Some;

   X1: Reg_Desl generic map (N2) port map (fio2               ,
                                           Limpa_Parcial      ,
 Desloque           ,
                                           fio1(N2-1 downto 0),
                                           sai               );

X2: Registrador generic map (N2) port map (Carregue_Saida,
                                           Limpa         ,
 sai           ,
 Saida        );

end Estrutural;
```

A.2.9 Somador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Somador is
generic (N    : integer := 64);
port (Entrada0: in  std_logic_vector (N-1 downto 0);
      Entrada1: in  std_logic_vector (N-1 downto 0);
Saida   : out std_logic_vector (N   downto 0));
end Somador;

architecture RTL of Somador is

begin
```

```
Saida <= ('0' & Entrada0) + ('0' & Entrada1);

end RTL;
```

A.2.10 Registrador de Deslocamento

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Reg_Desl is
generic (N: integer := 64);
port    (Relogio : in  std_logic;
         Limpa   : in  std_logic;
Desloque: in  std_logic;
Entrada : in  std_logic_vector (N-1 downto 0) ;
Saida   : out std_logic_vector (N-1 downto 0));
end Reg_Desl;

architecture RTL of Reg_Desl is

signal Saida_aux : std_logic_vector (N-1 downto 0);

begin

Saida <= Saida_aux;

process (Relogio, Limpa)
begin
if Limpa = '1' then
Saida_aux <= (others => '0');
elsif Relogio'event and Relogio = '1' then
if Desloque = '1' then
if Saida_aux(N-1) = '1' then
Saida_aux(N-1) <= '1';
Saida_aux(N-2 downto 0) <= Saida_aux(N-1 downto 1);
else
Saida_aux(N-1) <= '0';
Saida_aux(N-2 downto 0) <= Saida_aux(N-1 downto 1);
end if;
else
Saida_aux <= Entrada;
end if;
end if;
end process;
```

```
end RTL;
```

A.3 Módulo CAN

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity CAN_Module is
port (Clock    : in  std_logic;
Reset    : in  std_logic;
CAN_Rx_Bit : in  std_logic;
CAN_Tx_Req : in  std_logic;
CAN_RTR    : in  std_logic;
CAN_IDE    : in  std_logic;
CAN_DLC    : in  std_logic_vector( 3 downto 0);
CAN_BRP    : in  std_logic_vector( 5 downto 0);
CAN_PRSEG  : in  std_logic_vector( 2 downto 0);
CAN_PSEG1  : in  std_logic_vector( 2 downto 0);
CAN_PSEG2  : in  std_logic_vector( 2 downto 0);
CAN_SJW    : in  std_logic_vector( 1 downto 0);
CAN_ID_Tx  : in  std_logic_vector(28 downto 0);
CAN_MSG_Tx : in  std_logic_vector(63 downto 0);
CAN_ID_Rx  : out std_logic_vector(28 downto 0);
CAN_MSG_Rx : out std_logic_vector(63 downto 0);
CAN_Tx_Busy: out std_logic;
CAN_Tx_Comp: out std_logic;
CAN_Rx_Busy: out std_logic;
CAN_Rx_Comp: out std_logic;
CAN_Tx_Bit : out std_logic);
end CAN_Module;

architecture Estrutural of CAN_Module is

-- Declaração dos Componentes

component CAN_Tx
port(
Clock  : in  std_logic;
Reset  : in  std_logic;
TxClock  : in  std_logic;
SamplePoint  : in  std_logic;
RTR_Tx  : in  std_logic;
```

```
IDE_Tx  : in  std_logic;
DLC  : in  std_logic_vector( 3 downto 0);
ID_Tx  : in  std_logic_vector(28 downto 0);
MSG_Tx  : in  std_logic_vector(63 downto 0);
CRC  : in  std_logic_vector(14 downto 0);
Rx_Bit  : in  std_logic;
Rx_Error  : in  std_logic;
Stuff_Bit  : in  std_logic;
Stuff_Error  : in  std_logic;
CRC_Reset  : out std_logic;
CRC_Stop  : out std_logic;
LostArbitration: out std_logic;
Tx_Bit  : out std_logic;
TxBusy  : out std_logic;
TxComp  : out std_logic;
Tx_Error  : out std_logic
);
end component;

component CAN_Rx
port(
Clock   : in  std_logic;
Reset   : in  std_logic;
SamplePoint  : in  std_logic;
ACKPoint     : in  std_logic;
TxBusy       : in  std_logic;
RxBit   : in  std_logic;
Stuff_Bit   : in  std_logic;
Stuff_Error  : in  std_logic;
CRC   : in  std_logic_vector(14 downto 0);
CRC_Reset   : out std_logic;
CRC_Stop    : out std_logic;
Stuff_Disable: out std_logic;
ACK_Tx  : out std_logic;
RxBusy       : out std_logic;
Rx_Completed : out std_logic;
Rx_Error     : out std_logic;
ID_Rx   : out std_logic_vector(28 downto 0);
MSG_Rx   : out std_logic_vector(63 downto 0)
);
end component;

component CAN_CRC
port(
Clock       : in  std_logic;
Reset       : in  std_logic;
SamplePoint : in  std_logic;
```

```
CAN_Bit     : in  std_logic;
Stuff_Bit   : in  std_logic;
CRC_Stop    : in  std_logic;
CRC_Sequence: out std_logic_vector(14 downto 0)
);
end component;

component CAN_Bit_Stuffing
port(
Clock      : in  std_logic;
Reset      : in  std_logic;
RxBit      : in  std_logic;
SamplePoint: in  std_logic;
Stuff_Bit  : out std_logic;
Stuff_Error: out std_logic
);
end component;

component CAN_Bit_Timing2
port(
Clock  : in  std_logic;
Reset  : in  std_logic;
TxReq  : in  std_logic;
RxBusy : in  std_logic;
TxComp : in  std_logic;
BRP    : in  std_logic_vector(5 downto 0);
PRSEG  : in  std_logic_vector(2 downto 0);
PSEG1  : in  std_logic_vector(2 downto 0);
PSEG2  : in  std_logic_vector(2 downto 0);
ResetTx: out std_logic;
TxPoint: out std_logic
);
end component;

component CAN_Bit_Timing
port(
Clock    : in std_logic;
Reset    : in std_logic;
RxBit    : in std_logic;
EndFr    : in std_logic;
BRP      : in std_logic_vector(5 downto 0);
PRSEG    : in std_logic_vector(2 downto 0);
PSEG1    : in std_logic_vector(2 downto 0);
PSEG2    : in std_logic_vector(2 downto 0);
SJW      : in std_logic_vector(1 downto 0);
ACKPoint   : out std_logic;
SamplePoint: out std_logic
```

```
);
end component;

-- Declaração dos Sinais Internos

signal TxClock  :  std_logic := '0';
signal SamplePoint  :  std_logic := '0';
signal RTR_Tx  :  std_logic := '0';
signal IDE_Tx  :  std_logic := '0';
signal DLC :  std_logic_vector( 3 downto 0) := (others=>'0');
signal ID_Tx :  std_logic_vector(28 downto 0) := (others=>'0');
signal MSG_Tx :  std_logic_vector(63 downto 0) := (others=>'0');
signal CRC_Tx :  std_logic_vector(14 downto 0) := (others=>'0');
signal Rx_Bit  :  std_logic := '0';
signal Rx_Error    :  std_logic := '0';
signal Stuff_Bit  :  std_logic := '0';
signal Stuff_Error  :  std_logic := '0';
signal CRC_Reset_Tx  :  std_logic := '0';
signal CRC_Stop_Tx :  std_logic := '0';
signal LostArbitration :  std_logic := '0';
signal Tx_Bit  :  std_logic := '0';
signal TxBusy  :  std_logic := '0';
signal TxComp  :  std_logic := '0';
signal Tx_Error  :  std_logic := '0';
signal ACKPoint       :  std_logic := '0';
signal CRC_Rx        :  std_logic_vector(14 downto 0) := (others=>'0');
signal CRC_Reset_Rx   :  std_logic := '0';
signal CRC_Stop_Rx    :  std_logic := '0';
signal Stuff_Disable  :  std_logic := '0';
signal ACK_Tx   :  std_logic := '0';
signal RxBusy  :  std_logic := '0';
signal Rx_Completed :  std_logic := '0';
signal ID_Rx    :  std_logic_vector(28 downto 0) := (others=>'0');
signal MSG_Rx    :  std_logic_vector(63 downto 0) := (others=>'0');
signal CAN_CRC_Reset_Tx     :  std_logic := '0';
signal CAN_CRC_Reset_Rx     :  std_logic := '0';
signal CAN_Bit_Stuffing_Reset :  std_logic := '0';
signal CAN_Bit_Timing2_Reset  :  std_logic := '0';
signal TxReq :  std_logic := '0';
signal BRP          :  std_logic_vector(5 downto 0) := (others=>'0');
signal PRSEG        :  std_logic_vector(2 downto 0) := (others=>'0');
signal PSEG1        :  std_logic_vector(2 downto 0) := (others=>'0');
signal PSEG2        :  std_logic_vector(2 downto 0) := (others=>'0');
signal SJW          :  std_logic_vector(1 downto 0) := (others=>'0');
signal ResetTx               :  std_logic;
signal CAN_Tx_Reset          :  std_logic;
```

```
begin

-- Equações de Saída

CAN_ID_Rx   <= ID_Rx;
CAN_MSG_Rx  <= MSG_Rx;
CAN_Tx_Busy <= TxBusy;
CAN_Tx_Comp <= TxComp;
CAN_Rx_Busy <= RxBusy;
CAN_Rx_Comp <= Rx_Completed;
CAN_Tx_Bit  <= Tx_Bit and ACK_Tx;

-- Sinais Internos

Rx_Bit                 <= CAN_Rx_Bit;
TxReq                  <= CAN_Tx_Req;
RTR_Tx    <= CAN_RTR;
IDE_Tx       <= CAN_IDE;
DLC                    <= CAN_DLC;
BRP    <= CAN_BRP;
PRSEG    <= CAN_PRSEG;
PSEG1    <= CAN_PSEG1;
PSEG2    <= CAN_PSEG2;
SJW   <= CAN_SJW;
ID_Tx     <= CAN_ID_Tx;
MSG_Tx    <= CAN_MSG_Tx;
CAN_CRC_Reset_Tx   <= Reset or CRC_Reset_Tx or (not TxBusy);
CAN_CRC_Reset_Rx   <= Reset or CRC_Reset_Rx;
CAN_Bit_Stuffing_Reset <= Reset or Stuff_Disable;
CAN_Bit_Timing2_Reset  <= Reset or LostArbitration;
CAN_Tx_Reset           <= Reset or ResetTx;

-- Interligação dos componentes

-- Módulo de transmissão CAN

X6: CAN_Tx port map
(
Clock           => Clock,
Reset           => CAN_Tx_Reset,
TxClock         => TxClock,
SamplePoint     => SamplePoint,
RTR_Tx          => RTR_Tx,
IDE_Tx          => IDE_Tx,
DLC             => DLC,
ID_Tx           => ID_Tx,
MSG_Tx          => MSG_Tx,
```

```
CRC             => CRC_Tx,
Rx_Bit          => Rx_Bit,
Rx_Error        => Rx_Error,
Stuff_Bit       => Stuff_Bit,
Stuff_Error     => Stuff_Error,
CRC_Reset       => CRC_Reset_Tx,
CRC_Stop        => CRC_Stop_Tx,
LostArbitration => LostArbitration,
Tx_Bit          => Tx_Bit,
TxBusy          => TxBusy,
TxComp          => TxComp,
Tx_Error        => Tx_Error
);

-- Módulo de recepção CAN

X5: CAN_Rx port map
(
Clock    => Clock,
Reset    => Reset,
SamplePoint   => SamplePoint,
ACKPoint      => ACKPoint,
TxBusy        => TxBusy,
RxBit    => Rx_Bit,
Stuff_Bit    => Stuff_Bit,
Stuff_Error   => Stuff_Error,
CRC    => CRC_Rx,
CRC_Reset    => CRC_Reset_Rx,
CRC_Stop    => CRC_Stop_Rx,
Stuff_Disable => Stuff_Disable,
ACK_Tx   => ACK_Tx,
RxBusy        => RxBusy,
Rx_Completed  => Rx_Completed,
Rx_Error    => Rx_Error,
ID_Rx    => ID_Rx,
MSG_Rx    => MSG_Rx
);

-- Módulo de cálculo de CRC para CAN_Tx

X4: CAN_CRC port map
(
Clock         => Clock,
Reset    => CAN_CRC_Reset_Tx,
SamplePoint   => SamplePoint,
CAN_Bit       => Tx_Bit,
Stuff_Bit     => Stuff_Bit,
```

```
CRC_Stop      => CRC_Stop_Tx,
CRC_Sequence  => CRC_Tx
);

-- Módulo de cálculo de CRC para CAN_Rx

X3: CAN_CRC port map
(
Clock         => Clock,
Reset    => CAN_CRC_Reset_Rx,
SamplePoint   => SamplePoint,
CAN_Bit       => Rx_Bit,
Stuff_Bit     => Stuff_Bit,
CRC_Stop      => CRC_Stop_Rx,
CRC_Sequence  => CRC_Rx
);

-- Módulo detector e gerador de "Stuff Bit"

X2: CAN_Bit_Stuffing port map
   (
Clock       => Clock,
Reset  => CAN_Bit_Stuffing_Reset,
RxBit       => Rx_Bit,
SamplePoint => SamplePoint,
Stuff_Bit   => Stuff_Bit,
Stuff_Error => Stuff_Error
);

-- Módulo de sincronização 2

X1: CAN_Bit_Timing2 port map
(
Clock  => Clock,
Reset  => CAN_Bit_Timing2_Reset,
TxReq  => TxReq,
RxBusy => RxBusy,
TxComp => TxComp,
BRP    => BRP,
PRSEG  => PRSEG,
PSEG1  => PSEG1,
PSEG2  => PSEG2,
ResetTx=> ResetTx,
TxPoint=> TxClock
);

-- Módulo de sincronização
```

```
X0: CAN_Bit_Timing port map
(
Clock  => Clock,
Reset  => Reset,
RxBit  => Rx_Bit,
EndFr  => Rx_Completed,
BRP    => BRP,
PRSEG  => PRSEG,
PSEG1  => PSEG1,
PSEG2  => PSEG2,
SJW    => SJW,
ACKPoint  => ACKPoint,
SamplePoint => SamplePoint
);

end Estrutural;
```

A.3.1 CAN TX

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CAN_Tx is
port (Clock       : in  std_logic ;
      Reset          : in  std_logic ;
      TxClock       : in  std_logic ;
SamplePoint    : in  std_logic ;
--      ErrorPassive   : in  std_logic ;
--BusOff         : in  std_logic ;
      RTR_Tx         : in  std_logic ;
IDE_Tx         : in  std_logic ;
DLC            : in  std_logic_vector ( 3 downto 0) ;
ID_Tx          : in  std_logic_vector (28 downto 0) ;
      MSG_Tx       : in  std_logic_vector (63 downto 0) ;
      CRC            : in  std_logic_vector (14 downto 0) ;
Rx_Bit       : in  std_logic ;
Rx_Error       : in  std_logic ;
Stuff_Bit      : in  std_logic ;
      Stuff_Error    : in  std_logic ;
Tx_Bit         : out std_logic ;
LostArbitration: out std_logic ;
      CRC_Reset      : out std_logic ;
      CRC_Stop       : out std_logic ;
TxBusy         : out std_logic ;
TxComp         : out std_logic ;
```

```
Tx_Error       : out std_logic);
end CAN_Tx;

architecture RTL of CAN_Tx is

-- Máquina sequencial de estados

type states is (StartOfFrame, BaseID, IDEBit, ExtendedID, R1Bit,
CtrlField, DataField, CRCField, ACKSlotBit,
ACKDelimitBit, EndOfFrame, ErrorFrame);
signal state              : states;

-- Sinais auxiliares

signal busmonitor         : std_logic;
signal biterror           : std_logic;
signal counter            : std_logic_vector ( 6 downto 0);

-- Entradas

signal rxbit              : std_logic;
signal rtr                : std_logic;
signal ide                : std_logic;
signal controlfield       : std_logic_vector ( 4 downto 0);
signal msgid              : std_logic_vector (28 downto 0);
signal msg               : std_logic_vector (63 downto 0);
signal crcsequence        : std_logic_vector (14 downto 0);

-- Saídas

signal txbit              : std_logic;
signal lostarb            : std_logic;

begin

LostArbitration <= lostarb;
Tx_Bit          <= txbit;

-- Processo que monitora o barramento (amostra o bit recebido)

process (Clock, Reset)
begin
if Reset = '1' then
rxbit <= '1';
elsif (Clock'event and Clock = '1') then
if SamplePoint = '1' then
rxbit <= Rx_Bit;
```

```
end if;
end if;
end process;

-- Processo que compara os bits transmitido e recebido

process (Clock, Reset)
begin
if Reset = '1' then
lostarb  <= '0';
biterror <= '0';
elsif (Clock'event and Clock = '1') then
if TxClock = '1' and busmonitor = '1' then
if txbit = '1' and rxbit = '0' then
lostarb  <= '1';
biterror <= '0';
elsif txbit = '0' and rxbit = '1' then
lostarb  <= '0';
biterror <= '1';
else
lostarb  <= '0';
biterror <= '0';
end if;
end if;
end if;
end process;

-- Controlador de transmissão de mensagem CAN

process (Clock, Reset)
begin
   if Reset = '1' then -- Reset assíncrono
-- Máquina de estados
state        <=  StartOfFrame;
-- Sinais auxiliares
busmonitor   <= '0';
counter      <= (others => '0');
-- Entradas
rtr          <= '0';
ide          <= '0';
controlfield <= (others => '0');
msgid        <= (others => '0');
msg          <= (others => '0');
crcsequence  <= (others => '0');
-- Saídas
CRC_Reset    <= '0';
CRC_Stop  <= '0';
```

```
      txbit        <= '1';
TxBusy       <= '0';
TxComp       <= '0';
Tx_Error     <= '0';
elsif (Clock'event and Clock = '1') then
      if TxClock = '1' then
         if biterror = '0' and Rx_Error = '0' and Stuff_Error = '0' then
            case state is
               when StartOfFrame =>
state        <= BaseID;
busmonitor   <= '1';
counter      <= (others => '0');
rtr          <= RTR_Tx;
ide          <= IDE_Tx;
controlfield <= '0' & DLC;
msgid        <= ID_Tx;
msg          <= MSG_Tx;
crcsequence  <= (others => '0');
CRC_Reset    <= '0';
CRC_Stop  <= '0';
txbit        <= '0';
TxBusy       <= '1';
TxComp       <= '0';
Tx_Error     <= '0';
               when BaseID =>
         busmonitor   <= '1';
CRC_Reset    <= '0';
CRC_Stop  <= '0';
TxBusy       <= '1';
TxComp       <= '0';
Tx_Error     <= '0';
if lostarb = '1' then
state   <= StartOfFrame;
busmonitor <= '0';
TxBusy     <= '0';
elsif counter = "0001011" then
                     if ide = '0' and Stuff_Bit = '0' then
txbit   <= RTR;
                        state   <= IDEBit;
counter <= (others => '0');
         elsif ide = '1' and Stuff_Bit = '0' then
                        txbit   <= '1'; -- SRR
state   <= IDEBit;
counter <= (others => '0');
         else
                        txbit   <= not txbit;
state   <= BaseID;
```

```
end if;
                  else
                     state <= BaseID;
                     if Stuff_Bit = '0' then
            txbit   <= msgid(28);
counter <= counter+1;
                        msgid(28 downto 18) <= msgid(27 downto 18) & msg(28);
else
txbit   <= not txbit;
                     end if;
end if;
               when IDEBit =>
busmonitor <= '1';
CRC_Reset  <= '0';
CRC_Stop   <= '0';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if lostarb = '1' then
state      <= StartOfFrame;
busmonitor <= '0';
TxBusy     <= '0';
elsif ide = '1' and Stuff_Bit = '0' then
txbit <= IDE;
state <= ExtendedID;
      elsif ide = '0' and Stuff_Bit = '0' then
txbit <= IDE;
state <= CtrlField;
      else
txbit <= not txbit;
state <= IDEBit;
end if;
               when ExtendedID =>
busmonitor <= '1';
CRC_Reset  <= '0';
CRC_Stop   <= '0';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if lostarb = '1' then
state      <= StartOfFrame;
busmonitor <= '0';
TxBusy     <= '0';
elsif counter = "0010010" then
            if Stuff_Bit = '0' then
txbit   <= RTR;
                        state   <= R1Bit;
```

```
counter <= (others => '0');
                    else
txbit   <= not txbit;
state   <= ExtendedID;
end if;
         else
            state <= ExtendedID;
                    if Stuff_Bit = '0' then
                       txbit   <= msgid(17);
counter <= counter+1;
            msgid(17 downto 0) <= msgid(16 downto 0) & msgid(17);
else
txbit   <= not txbit;
end if;
end if;
when R1Bit =>
busmonitor <= '1';
CRC_Reset  <= '0';
CRC_Stop   <= '0';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if Stuff_Bit = '0' then
txbit <= '0'; -- R1
               state <= CtrlField;
                  else
txbit <= not txbit;
state <= R1Bit;
         end if;
when CtrlField =>
busmonitor <= '1';
CRC_Reset  <= '0';
CRC_Stop   <= '0';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if counter = "0000100" then -- Transmitindo DLC0
            if rtr = '1' and Stuff_Bit = '0' then
txbit   <= controlfield(4); -- DLC0
            state   <= CRCField;
counter <= (others => '0');
  controlfield (4 downto 0) <= controlfield(3 downto 0) & controlfield(4);
elsif rtr = '0' and Stuff_Bit = '0' then
txbit   <= controlfield(4); -- DLC0
            state   <= DataField;
counter <= (others => '0');
  controlfield (4 downto 0) <= controlfield(3 downto 0) & controlfield(4);
```

```
else
txbit <= not txbit;
state <= CtrlField;
end if;
                  else
            state <= CtrlField;
            if Stuff_Bit = '0' then
            txbit   <= controlfield(4);
counter <= counter+1;
  controlfield (4 downto 0) <= controlfield(3 downto 0) & controlfield(4);
else
txbit <= not txbit;
end if;
                  end if;
when DataField =>
busmonitor <= '1';
CRC_Reset  <= '0';
CRC_Stop   <= '0';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if controlfield(3 downto 0) > "1000" and Stuff_Bit = '0' then
                     controlfield(3 downto 0) <= "1000";
end if;
if counter = (controlfield(3 downto 0)&"000"-1) then
if Stuff_Bit = '0' then -- Transmitindo último bit de dados
txbit   <= msg(63);
state   <= CRCField;
counter <= (others => '0');
msg(63 downto 0) <= msg(62 downto 0) & msg(63);
else
txbit  <= not txbit;
state  <= DataField;
end if;
                  else
state     <= DataField;
                     if Stuff_Bit = '0' then
txbit    <= msg(63);
counter  <= counter+1;
            msg(63 downto 0) <= msg(62 downto 0) & msg(63);
else
txbit  <= not txbit;
end if;
                  end if;
               when CRCField =>
busmonitor <= '1';
CRC_Reset  <= '0';
```

```
CRC_Stop   <= '1';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if counter = "0001111" and Stuff_Bit = '0' then
txbit   <= '1';
state   <= ACKSlotBit;
counter <= (others => '0');
elsif counter = "0000000" and Stuff_Bit = '0' then
state       <= CRCField;
counter     <= counter+1;
txbit       <= CRC(14);
crcsequence <= CRC (13 downto 0) & CRC(14);
else
state       <= CRCField;
                    if Stuff_Bit = '0' then
txbit   <= crcsequence(14);
counter <= counter+1;
crcsequence(14 downto 0) <= crcsequence(13 downto 0) & crcsequence(14);
else
txbit <= not txbit;
end if;
end if;
         when ACKSlotBit =>
state      <= ACKDelimitBit;
busmonitor <= '0';
CRC_Reset  <= '1';
CRC_Stop   <= '1';
txbit      <= '1';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
when ACKDelimitBit =>
         state      <= EndOfFrame;
busmonitor <= '0';
CRC_Reset  <= '1';
CRC_Stop   <= '1';
txbit      <= '1';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
if rxbit = '0' then -- ACK recebido
state <= EndOfFrame;
else
Tx_Error <= '1';
state    <= ErrorFrame;
end if;
```

```
               when EndOfFrame =>
state      <= EndOfFrame;
busmonitor <= '0';
CRC_Reset  <= '1';
CRC_Stop   <= '1';
txbit      <= '1';
TxBusy     <= '1';
TxComp     <= '0';
Tx_Error   <= '0';
         if counter = "0000111" then
state    <= StartOfFrame;
busmonitor <= '0';
counter    <= (others => '0');
crcsequence<= (others => '0');
CRC_Reset  <= '0';
CRC_Stop   <= '0';
txbit      <= '1';
TxBusy     <= '0';
TxComp     <= '1';
Tx_Error   <= '0';
else
counter <= counter+1;
end if;
         when ErrorFrame =>
-- Enviar quadro de erro ativo ou passivo
            end case;
      else
            state <= ErrorFrame;
      end if;
   end if;
   end if;
end process;

end RTL;
```

A.3.2 CAN RX

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CAN_Rx is
port (Clock       : in  std_logic ;
      Reset          : in  std_logic ;
      SamplePoint    : in  std_logic ;
ACKPoint       : in  std_logic ;
```

```
      TxBusy          : in  std_logic ;
RxBit          : in  std_logic ;
      Stuff_Bit       : in  std_logic ;
      Stuff_Error     : in  std_logic ;
      CRC             : in  std_logic_vector (14 downto 0) ;
      CRC_Reset       : out std_logic ;
      CRC_Stop        : out std_logic ;
      Stuff_Disable   : out std_logic ;
      ACK_Tx          : out std_logic ;
RxBusy         : out std_logic ;
      Rx_Completed    : out std_logic ;
      Rx_Error        : out std_logic ;
      ID_Rx           : out std_logic_vector (28 downto 0) ;
      MSG_Rx        : out std_logic_vector (63 downto 0));
end CAN_Rx;

architecture RTL of CAN_Rx is

type estados is (StartOfFrame, BaseID, IDEBit, ExtendedID,
RemFrCtrlField, DataFrCtrlField, DataField,
CRCField, ACKSlotBit, ACKDelimitBit, EndOfFrame, qerror, R1Bit);

signal estado        : estados;
signal ACK           : std_logic;
signal contador      : std_logic_vector (6  downto 0);
signal campo_controle: std_logic_vector (5  downto 0);
signal crc_recebido  : std_logic_vector (14 downto 0);
signal MSG_ID        : std_logic_vector (28 downto 0);
signal MSG          : std_logic_vector (63 downto 0);

begin

ID_Rx  <= MSG_ID;
MSG_Rx <= MSG;

-- Processo que gera o Bit ACK

process (Clock, Reset)
begin
if Reset = '1' then
ACK_Tx <= '1';
elsif (Clock'event and Clock = '1') then
if ACKPoint = '1' then
if ACK = '0' then
ACK_Tx <= '0';
else
ACK_Tx <= '1';
```

```
end if;
end if;
end if;
end process;

-- Controlador de recepção de mensagem CAN

process (Clock, Reset)
begin
   if Reset = '1' then   -- Reset assíncrono
   estado            <=  StartOfFrame;
         CRC_Reset         <= '0';
CRC_Stop <= '0';
      Stuff_Disable     <= '0';
      ACK          <= '1';
RxBusy            <= '0';
      Rx_Completed      <= '0';
      Rx_Error          <= '0';
MSG_ID            <= (others => '0');
         MSG               <= (others => '0');
contador <= (others => '0');
      campo_controle    <= (others => '0');
      crc_recebido      <= (others => '0');
   elsif (Clock'event and Clock = '1') then
      if SamplePoint = '1' then
         if Stuff_Error = '0' then
            case estado is
               when StartOfFrame =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '0';
Rx_Completed   <= '0';
Rx_Error       <= '0';
MSG_ID         <= (others => '0');
MSG            <= (others => '0');
contador <= (others => '0');
campo_controle    <= (others => '0');
crc_recebido      <= (others => '0');
                  if RxBit = '0' then
         estado       <= BaseID;
RxBusy         <= '1';
else
estado <= qerror;
end if;
               when BaseID =>
```

```
         CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
if contador = "0001011" then
                     if RxBit = '1' and Stuff_Bit = '0' then
                        estado   <= IDEBit;
contador <= (others => '0');
         elsif RxBit = '0' and Stuff_Bit = '0' then
                        estado   <= DataFrCtrlField;
contador <= (others => '0');
         else
                        estado   <= BaseID;
end if;
                  else
                     estado <= BaseID;
                     if Stuff_Bit = '0' then
            contador <= contador+1;
                        MSG_ID(28 downto 18) <= MSG_ID(27 downto 18) & RxBit;
                     end if;
end if;
               when IDEBit =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
if RxBit = '1' and Stuff_Bit = '0' then
estado <= ExtendedID;
      elsif RxBit = '0' and Stuff_Bit = '0' then
estado <= RemFrCtrlField;
      else
estado <= IDEBit;
end if;
               when ExtendedID =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
```

```
if contador = "0010010" then
            if RxBit = '1' and Stuff_Bit = '0' then
                        estado   <= R1Bit;
contador <= (others => '0');
                     elsif RxBit = '0' and Stuff_Bit = '0' then
                        estado   <= DataFrCtrlField;
contador <= (others => '0');
                     else
estado   <= ExtendedID;
end if;
         else
            estado <= ExtendedID;
                     if Stuff_Bit = '0' then
                        contador <= contador+1;
            MSG_ID(17 downto 0) <= MSG_ID(16 downto 0) & RxBit;
end if;
end if;
      when RemFrCtrlField =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
if contador = "0000100" then
            if Stuff_Bit = '0' then
            estado   <= CRCField;
contador <= (others => '0');
               campo_controle (5 downto 0) <= campo_controle(4 downto 0) & RxBit;
CRC_Stop <= '1';
end if;
                  else
            estado <= RemFrCtrlField;
            if Stuff_Bit = '0' then
            contador <= contador+1;
            campo_controle (5 downto 0) <= campo_controle(4 downto 0) & RxBit;
end if;
                  end if;
      when DataFrCtrlField =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
```

```
                    if contador = "0000101" then
if Stuff_Bit = '0' then
estado   <= DataField;
contador <= (others => '0');
campo_controle (5 downto 0) <= campo_controle(4 downto 0) & RxBit;
end if;
else
estado <= DataFrCtrlField;
if Stuff_Bit = '0' then
contador <= contador+1;
campo_controle (5 downto 0) <= campo_controle(4 downto 0) & RxBit;
end if;
end if;
               when DataField =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
if campo_controle(3 downto 0) > "1000" and Stuff_Bit = '0' then
                    campo_controle(3 downto 0) <= "1000";
end if;
         if contador = (campo_controle(3 downto 0)&"000"-1) then
if Stuff_Bit = '0' then
estado   <= CRCField;
contador <= (others => '0');
MSG(63 downto 0) <= MSG(62 downto 0) & RxBit;
CRC_Stop <= '1';
end if;
                  else
estado    <= DataField;
                    if Stuff_Bit = '0' then
contador  <= contador+1;
            MSG(63 downto 0) <= MSG(62 downto 0) & RxBit;
end if;
                  end if;
               when CRCField =>
CRC_Reset      <= '0';
CRC_Stop       <= '1';
Stuff_Disable  <= '0';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
                    if contador = "0001111" and Stuff_Bit = '0' then
```

```
Stuff_Disable <= '1';
if TxBusy = '0' then
if crc_recebido = CRC then
ACK      <= '0';
estado   <= ACKSlotBit;  => ack_slot
contador <= (others => '0');
else
Rx_Error <= '1';
estado   <= qerror;
end if;
else
estado   <= ACKSlotBit;
contador <= (others => '0');
end if;
else
estado <= CRCField;
                    if Stuff_Bit = '0' then
contador  <= contador+1;
crc_recebido(14 downto 0) <= crc_recebido(13 downto 0) & RxBit;
end if;
end if;
         when ACKSlotBit =>
         CRC_Reset      <= '1';
CRC_Stop       <= '1';
Stuff_Disable  <= '1';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
             estado            <= ACKDelimitBit;
               when ACKDelimitBit =>
CRC_Reset      <= '1';
CRC_Stop       <= '1';
Stuff_Disable  <= '1';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
         estado            <= EndOfFrame;
               when EndOfFrame =>
CRC_Reset      <= '1';
CRC_Stop       <= '1';
Stuff_Disable  <= '1';
ACK            <= '1';
RxBusy            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
```

```
estado            <= EndOfFrame;
         if contador = "0000110" then
if RxBit = '1' then
RxBusy <= '0';
Rx_Completed <= '1';
Stuff_Disable <= '0';
estado <= StartOfFrame;
else
estado <= qerror;
end if;
else
if RxBit = '1' then
contador <= contador+1;
else
estado <= qerror;
end if;
                  end if;
               when R1Bit =>
CRC_Reset      <= '0';
CRC_Stop       <= '0';
Stuff_Disable  <= '0';
ACK            <= '1';
Rx_Completed   <= '0';
Rx_Error       <= '0';
         if Stuff_Bit = '0' then
               estado <= RemFrCtrlField;
                  else
estado <= R1Bit;
         end if;
         when qerror =>
         Stuff_Disable   <= '1';
                  Rx_Error     <= '1';
            end case;
      else
            estado <= qerror;
      end if;
   end if;
   end if;
end process;

end RTL;
```

A.3.3 CRC

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;

entity CAN_CRC is
port (Clock    : in  std_logic;
Reset       : in  std_logic;
   SamplePoint : in  std_logic;
CAN_Bit     : in  std_logic;
      Stuff_Bit   : in  std_logic;
   CRC_Stop    : in  std_logic;
   CRC_Sequence: out std_logic_vector (14 downto 0));
end CAN_CRC;

architecture Comportamental of CAN_CRC is

signal NXTBIT: std_logic;

begin

NXTBIT <= CAN_Bit;

-- Processo que calcula a sequência de CRC

process (Clock, Reset)
variable CRCNXT : std_logic;
variable CRC_RG : std_logic_vector (14 downto 0);
begin
   if Reset = '1' then
      CRCNXT      := '1';
   CRC_RG      := (others => '0');
   CRC_Sequence <= (others => '0');
   elsif (Clock'event and Clock = '1') then
      if CRC_Stop = '0' then
      if SamplePoint = '1' then
            if Stuff_Bit = '0' then
                  CRCNXT := NXTBIT xor CRC_RG(14);
               CRC_RG(14 downto 1) := CRC_RG(13 downto 0);
               CRC_RG(0) := '0';
               if CRCNXT = '1' then
               CRC_RG(14 downto 0) := CRC_RG(14 downto 0) xor "100010110011001";
               end if;
      CRC_Sequence <= CRC_RG;
            end if;
         end if;
end if;
     end if;
end process;
```

```
end Comportamental;
```

A.3.4 STUFF HANDLER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CAN_Bit_Stuffing is
port (Clock      : in  std_logic ;
      Reset      : in  std_logic ;
      RxBit      : in  std_logic ;
SamplePoint: in  std_logic ;
Stuff_Bit  : out std_logic ;
Stuff_Error: out std_logic);
--      Stuff_Rx   : out std_logic ;
--      Stuff_Tx   : out std_logic ;
end CAN_Bit_Stuffing;

architecture RTL of CAN_Bit_Stuffing is

type estados is (Idle, Dominant1, Dominant2, Dominant3,
Dominant4, Dominant5, Recessive1,
Recessive2, Recessive3, Recessive4,
Recessive5, Error);

signal estado: estados;

begin

-- Processo que realiza o controle de "Bit Stuffing"

process (Clock, Reset)
begin
if Reset = '1' then
   Stuff_Bit   <= '0';
Stuff_Error <= '0';
   estado      <= Idle;
   elsif (Clock'event and Clock = '1') then
      if SamplePoint = '1' then
         case estado is
            when Idle =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant1;
```

```
               else
                  Stuff_Error <= '1';
estado <= Error;  resetado por CAN_Tx.
               end if;
            when Dominant1 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant2;
               else
                  estado <= Recessive1;
               end if;
            when Dominant2 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant3;
               else
                  estado <= Recessive1;
               end if;
            when Dominant3 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant4;
               else
                  estado <= Recessive1;
               end if;
            when Dominant4 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
Stuff_Bit <= '1';
estado <= Dominant5;
               else
                  estado <= Recessive1;
               end if;
            when Dominant5 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
if RxBit = '0' then
                  Stuff_Error <= '1';
estado <= Error;
else
estado <= Recessive1;
end if;
            when Recessive1 =>
```

```
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant1;
               else
                  estado <= Recessive2;
               end if;
            when Recessive2 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant1;
               else
                  estado <= Recessive3;
               end if;
            when Recessive3 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant1;
               else
                  estado <= Recessive4;
               end if;
            when Recessive4 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
               if RxBit = '0' then
                  estado <= Dominant1;
               else
Stuff_Bit <= '1';
                  estado <= Recessive5;
               end if;
            when Recessive5 =>
Stuff_Bit   <= '0';
Stuff_Error <= '0';
if RxBit = '0' then
estado <= Dominant1;
else
Stuff_Error <= '1';
estado <= Error;
end if;
when Error =>
Stuff_Bit   <= '0';
Stuff_Error <= '1';
estado <= Error;
         end case;
         end if;
```

```
   end if;
end process;
end RTL;
```

A.3.5 BIT TIMING 1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CAN_Bit_Timing is
port (Clock: in std_logic;
Reset: in std_logic;
RxBit: in std_logic;
EndFr: in std_logic;
BRP  : in std_logic_vector(5 downto 0);
PRSEG: in std_logic_vector(2 downto 0);
PSEG1: in std_logic_vector(2 downto 0);
PSEG2: in std_logic_vector(2 downto 0);
SJW  : in std_logic_vector(1 downto 0);
ACKPoint   : out std_logic;
SamplePoint: out std_logic);
end CAN_Bit_Timing;

architecture RTL of CAN_Bit_Timing is

type estados is (Idle, SyncSeg, PropSeg, PhaseSeg1, PhaseSeg2);

signal estado  : estados;
signal liberaTq  : std_logic;
signal TimeQuantum : std_logic;
signal BRPCounter  : std_logic_vector(5 downto 0);
signal PRSEGCounter: std_logic_vector(2 downto 0);
signal PSEG1Counter: std_logic_vector(2 downto 0);
signal PSEG2Counter: std_logic_vector(2 downto 0);


begin

-- Processo que gera o TimeQuantum a partir do Clock e do
BRP (Baud Rate Prescaler)

process (Clock, Reset)
begin
if (Reset = '1') then
TimeQuantum <= '0';
```

```
BRPCounter  <= (others => '0');
elsif (Clock'event and Clock = '1') then
if liberaTq = '1' then
if (BRPCounter = BRP-1) then
TimeQuantum <= '1';
BRPCounter  <= (others => '0');
else
TimeQuantum <= '0';
BRPCounter  <= BRPCounter+1;
end if;
else
TimeQuantum <= '0';
BRPCounter  <= (others => '0');
end if;
end if;
end process;

-- Processo que controla a transição de estados entre os
segmentos de tempo de um bit CAN

process (Clock, Reset)
begin
if (Reset =  '1') then
PRSEGCounter <= (others => '0');
PSEG1Counter <= (others => '0');
PSEG2Counter <= (others => '0');
estado       <= Idle;
elsif (Clock'event and Clock = '1') then
case estado is
when Idle =>
ACKPoint    <= '0';
SamplePoint <= '0';
liberaTq    <= '0';
if RxBit = '0' then
liberaTq <= '1';
estado   <= SyncSeg;
end if;
when SyncSeg =>
ACKPoint    <= '1';
SamplePoint <= '0';
liberaTq    <= '1';
if TimeQuantum = '1' then
estado <= PropSeg;
end if;
when PropSeg =>
ACKPoint    <= '0';
SamplePoint <= '0';
```

```
liberaTq    <= '1';
if TimeQuantum = '1' then
if (PRSEGCounter = PRSEG-1) then
estado       <= PhaseSeg1;
PRSEGCounter <= (others => '0');
else
PRSEGCounter <= PRSEGCounter+1;
end if;
end if;
when PhaseSeg1 =>
ACKPoint    <= '0';
SamplePoint <= '0';
liberaTq    <= '1';
if TimeQuantum = '1' then
if (PSEG1Counter = PSEG1-1) then
SamplePoint  <= '1';
estado       <= PhaseSeg2;
PSEG1Counter <= (others => '0');
else
PSEG1Counter <= PSEG1Counter+1;
end if;
end if;
when PhaseSeg2 =>
ACKPoint    <= '0';
SamplePoint <= '0';
liberaTq    <= '1';
if TimeQuantum = '1' then
if (PSEG2Counter = PSEG2-1) and EndFr = '1' then
liberaTq     <= '0';
estado       <= Idle;
PSEG2Counter <= (others => '0');
elsif (PSEG2Counter = PSEG2-1) then
estado       <= SyncSeg;
PSEG2Counter <= (others => '0');
else
PSEG2Counter <= PSEG2Counter+1;
end if;
end if;
end case;
end if;
end process;

end RTL;
```

A.3.6 BIT TIMING 2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CAN_Bit_Timing2 is
port (Clock  : in  std_logic;
Reset  : in  std_logic;
TxReq  : in  std_logic;
RxBusy : in  std_logic;
TxComp : in  std_logic;
BRP    : in  std_logic_vector(5 downto 0);
PRSEG  : in  std_logic_vector(2 downto 0);
PSEG1  : in  std_logic_vector(2 downto 0);
PSEG2  : in  std_logic_vector(2 downto 0);
ResetTx: out std_logic;
TxPoint: out std_logic);
end CAN_Bit_Timing2;

architecture RTL of CAN_Bit_Timing2 is

type estados is (Idle, Transmission);

signal estado  : estados;
signal liberaTq  : std_logic;
signal TimeQuantum : std_logic;
signal BRPCounter  : std_logic_vector(5 downto 0);
signal Counter     : std_logic_vector(4 downto 0);

begin

-- Processo que gera o TimeQuantum a partir do Clock e do
 BRP (Baud Rate Prescaler)

process (Clock, Reset)
begin
if (Reset = '1') then
TimeQuantum <= '0';
BRPCounter  <= (others => '0');
elsif (Clock'event and Clock = '1') then
if liberaTq = '1' then
if (BRPCounter = BRP-1) then
TimeQuantum <= '1';
BRPCounter  <= (others => '0');
else
TimeQuantum <= '0';
BRPCounter  <= BRPCounter+1;
end if;
```

```
else
TimeQuantum <= '0';
BRPCounter  <= (others => '0');
end if;
end if;
end process;

-- Processo que gera o TxPoint a partir do TimeQuantum

process (Clock, Reset)
begin
if (Reset =  '1') then
estado       <= Idle;
liberaTq     <= '0';
Counter      <= (others => '0');
elsif (Clock'event and Clock = '1') then
case estado is
when Idle =>
TxPoint  <= '0';
liberaTq    <= '0';
ResetTx     <= '0';
estado      <= Idle;
Counter     <= (others => '0');
if TxReq = '1' and RxBusy = '0' then
TxPoint  <= '1';
liberaTq <= '1';
ResetTx  <= '1';
estado   <= Transmission;
end if;
when Transmission =>
TxPoint <= '0';
liberaTq    <= '1';
ResetTx     <= '0';
estado      <= Transmission;
if TxComp = '1' then
liberaTq <= '0';
estado   <= Idle;
Counter  <= (others => '0');
elsif TimeQuantum = '1' then
--if TimeQuantum = '1' then
if Counter = PRSEG+PSEG1+PSEG2 then
TxPoint  <= '1';
Counter  <= (others => '0');
else
Counter  <= Counter+1;
end if;
end if;
```

```
end case;
end if;
end process;

end RTL;
```

Apêndice B

Códigos C e MATLAB

Neste apêndice estão listados os códigos do amplificador sensível à fase digital implementado utilizando microcomputador com placa de aquisição de dados, escritos nas linguagens C e MATLAB .

B.1 Programa de Aquisição de Dados com a Placa DAS-20 em Linguagem C

```
/* Aquisição de Dados */

#include "lockin.h"

/* Imprime as informações sobre o programa e retorna os parâmetros do
   sinal de teste */

double parametros (void)

{
    double amplitude;

    printf ("\t\t%s\n  \t\t%s\n  \t\t%s\n  \t\t%s\n  \t\t%s\n\n \t\t%s\n\n",
            "  *******************************************   ",
            "        Universidade Federal de Pernambuco      ",
            "      Departamento de Eletronica e Sistemas     ",
            "   Laboratorio de Dispositivos e Nanoestruturas  ",
            "  *******************************************   ",
            "           Amplificador Lock-in Digital         ");

    printf ("Parametros pre-definidos: \n");
```

```
    printf (" - Frequencia: 1000 Hz \n");
    printf (" - Fase: 0 graus \n\n");
    printf ("Parametros a definir: \n");
    printf (" - Amplitude: - 5 V a + 5 V \n");
    printf ("Digite a amplitude do sinal (V): ");
    scanf  ("%lf", &amplitude);

    while (amplitude < -5 || amplitude > 5)
    {
        printf ("Amplitude invalida. Por favor, digite novamente\n\n");
        printf ("Digite a amplitude do sinal (V): ");
        scanf  ("%lf", &amplitude);
    }

    return amplitude;
}

/* Inicializa e configura a Placa DAS-20 */

void inicializa_placa (void)

{
    /* Reseta a Placa DAS-20 */

    printf ("Iniciando o Programa ... \n\n");

    printf("Resetando a Placa DAS-20 ... ");
    das20_master_reset();
    printf("OK\n");

    /* Gera a Taxa de Amostragem */

    printf("Gerando a Taxa de Amostragem ... ");
    generate_freq(2, 100000, 0.50);
/* Timer = 2, Frequencia = 100 kHz, Ciclo de Trabalho = 50% */
    printf("OK\n");

    /* Seleciona o Timer 2 para causar uma interrupção */

    printf("Configurando Interrupcoes ... ");
    set_int_source(INT_TMR2);
    printf("OK\n");

    /* Dispara o Timer 2 */

    printf("Selecionando Temporizadores ... ");
    arm_timers(2, 0);
```

```
    printf("OK\n\n");

    printf("Iniciando a Aquisicao de Dados ... \n");
}

/* Realiza a aquisicao de dados */

void aquisicao (double amplitude)

{
    int k;
    double T[80000], Vteste[80000], Vent[80000];
    FILE* arquivo;

    arquivo = fopen("aquisicao.txt","w");

    for (k=0; k < 10000; k++)
    {
        while (int_pending() == 0);
/* Fica aqui até ocorrer uma interrupção */
        int_clear(); /* Limpa "flag" de interrupção */

        /* Tempo da amostragem */
        T[k] = 0.00001 * k;

        /* Sinal de teste */
        Vteste[k] = (double) amplitude * cos (2 * 3.1415 * 10000 * T[k]);

        /* Faz a conversão D/A no Canal 0 com valor Vteste) */
        single_dac(0,Vteste[k]);

        /* Faz a Conversão A/D no Canal 0 e
Faixa de Operação 2 (-10 V a 10 V)*/
        Vent[k] = single_adc(0,2);



    }

    for (k = 0; k <10000; k ++)

    {

     fprintf(arquivo," %9.6lf, %9.6lf, %9.6lf \n", T[k], Vteste[k],Vent[k]);

    }
```

```
   fclose(arquivo);
}

/* Reseta a Placa DAS-20 e finaliza o Programa */

void finaliza_placa (void)

{
    printf("Finalizando o Programa ... \n\n");

    das20_master_reset();
    das20_cleanup();
}
```

B.2 Programas Usados na Implementação da Técnica *Lock-in* em MATLAB

B.2.1 lockincal.m

```
% Amplificador Lock-in Digital utilizando o MATLAB       %

% Programa de Calibraçao do Amplificador Lock-in Digital %

% AUTOR: José Edenilson Oliveira Reges                   %

% LDN  - Laboratório de Dispositivos e Nanoestruturas    %
% DES  - Departamento de Eletrônica e Sistemas           %
% UFPE - Universidade Federal de Pernambuco              %

load aquisicao.txt;

% Parâmetros utilizados %

f  = 1000;                  % Freqüência do sinal de referência     %
A  = max (aquisicao(:,2))   % Amplitude do sinal de referência      %
Rc = 993;                  % Resistor do conversor corrente/tensao %

% 1. Calibração do Amplificador Lock-in Digital %

% 1. Sinal de Referencia

% 1.2 Componente em fase %

Vfase = 1 * sin (2 * pi * f * aquisicao(:,1));
plot (aquisicao(:,1), Vfase);
xlabel ('Tempo (s)'), ylabel ('Vfase (V)');
```

```
% 1.3 Componente em quadratura %

Vquad = 1 * cos (2 * pi * f * aquisicao(:,1));
plot (aquisicao(:,1), Vquad);
xlabel ('Tempo (s)'), ylabel ('Vquad (V)');

% 2. Aquisiçao dos Sinais

% 2.1 Sinal de Teste

Vteste = aquisicao(:,2);
plot (aquisicao(:,1),aquisicao(:,2));
xlabel ('Tempo (s)'), ylabel ('Vteste (V)');

% 2.2 Sinal de Entrada

Vent = aquisicao(:,3);
plot (aquisicao(:,1),aquisicao(:,3));
xlabel ('Tempo (s)'), ylabel ('Vent (V)');

% 4. Multiplicador %

Vdet1 = Vent .* Vfase;
plot (aquisicao(:,1), Vdet1);
xlabel ('Tempo (s)'), ylabel ('Vdet1 (V)');

Vdet2 = Vent .* Vquad;
plot (aquisicao(:,1), Vdet2);
xlabel ('Tempo (s)'), ylabel ('Vdet2 (V)');

% 5. Filtro Passa-Baixa %

X = mean (Vdet1)
plot (aquisicao(:,1), X);
xlabel ('Tempo (s)'), ylabel ('Canal X');

Y = mean (Vdet2)
plot (aquisicao(:,1), Y);
xlabel ('Tempo (s)'), ylabel ('Canal Y');

% 1.7. Ajuste de Fase %

erro_graus = -((atan2(X,Y) * (180/pi)) - 90)

erro_rad = erro_graus * (pi/180)
```

B.2.2 lockinmed.m

```
% Amplificador Lock-in Digital utilizando o MATLAB    %

% Programa de mediçao com o amplificador lock-in      %

% AUTOR: José Edenilson Oliveira Reges                %

% LDN  - Laboratório de Dispositivos e Nanoestruturas %
% DES  - Departamento de Eletrônica e Sistemas        %
% UFPE - Universidade Federal de Pernambuco           %

load aquisicao.txt;

% Parâmetros utilizados %

f  = 1000;                   % Freqüência do sinal de referência     %
A  = max (aquisicao(:,2))    % Amplitude do sinal de referência      %
Rc = 993;                   % Resistor do conversor corrente/tensao %

% Simulaçao de Medições utilizando o Amplificador Lock-in Digital   %

% 1. Sinal de Referencia

% 1.2 Componente em fase %

Vfase = 1 * sin (2 * pi * f * aquisicao(:,1) + erro_rad);
plot (aquisicao(:,1), Vfase);
xlabel ('Tempo (s)'), ylabel ('Vfase (V)');

% 1.3 Componente em quadratura %

Vquad = 1 * cos (2 * pi * f * aquisicao(:,1) + erro_rad);
plot (aquisicao(:,1), Vquad);
xlabel ('Tempo (s)'), ylabel ('Vquad (V)');

% 2. Aquisiçao dos Sinais

% 2.1 Sinal de Teste

Vteste = aquisicao(:,2);
plot (aquisicao(:,1),aquisicao(:,2));
xlabel ('Tempo (s)'), ylabel ('Vteste (V)');

% 2.2 Sinal de Entrada

Vent = aquisicao(:,3);
```

```
plot (aquisicao(:,1),aquisicao(:,3));
xlabel ('Tempo (s)'), ylabel ('Vent (V)');

% 4. Multiplicador %

Vdet1 = Vent .* Vfase;
plot (aquisicao(:,1), Vdet1);
xlabel ('Tempo (s)'), ylabel ('Vdet1 (V)');

Vdet2 = Vent .* Vquad;
plot (aquisicao(:,1), Vdet2);
xlabel ('Tempo (s)'), ylabel ('Vdet2 (V)');

% 5. Filtro Passa-Baixa %

X = mean (Vdet1)
plot (aquisicao(:,1), X);
xlabel ('Tempo (s)'), ylabel ('Canal X');

Y = mean (Vdet2)
plot (aquisicao(:,1), Y);
xlabel ('Tempo (s)'), ylabel ('Canal Y');

% 6. Cálculo da Resistência %

R = - (1/2) * (A / X) * Rc

% 7. Cálculo da Capacitância %

C = - 2 * (Y / A) * (1 / (Rc * 2 * pi * f))
```

Apêndice C

Códigos ASM

Neste apêndice estão listados os códigos ASM utilizados na programação dos módulos CAN implementados com microcontroladores PIC 18F258.

C.1 Nó 0

```
CONFIG DEBUG = ON         ; HABILITA DEPURAÇÃO DE ERROS VIA ICD3 DEBUGGER

#INCLUDE <P18F258.INC>    ; INCLUI O ARQUIVO DE DEFINIÇÕES DO PIC18F258

ORG 0X0000  ; VETOR RESET.

MOVLB B'00001111' ; BANCO 15

;*********************************************************************
;                       CONFIGURAÇÃO DOS PINOS DE E/S                 *
;*********************************************************************

BSF TRISC,7  ; DEFINE RC7/RX/DT COMO ENTRADA.
BCF TRISC,6  ; DEFINE RC6/TX/CK COMO SAÍDA.
BSF TRISB,3  ; DEFINE RB3/CANRX COMO ENTRADA.
BCF TRISB,2  ; DEFINE RB2/CANTX/INT2 COMO SAÍDA.

;*********************************************************************
;                           CONFIGURAÇÃO DA USART                     *
;*********************************************************************

MOVLW .129  ; FOSC = 20 MHz, BAUD RATE = 9600 bps, BRGH = 1.
MOVWF SPBRG  ; DEFINE BAUD RATE = 9600 bps.
```

```
MOVLW B'00100100' ; TXEN = 1 (HABILITA TRANSMISSÃO), BRGH = 1 (ALTA VELOCIDADE).
MOVWF  TXSTA  ; CONFIGURA TRANSMISSÃO.

MOVLW B'10010000'  ; SPEN = 1 (HABILITA RECEPÇÃO), CREN = 1 (RECEPÇÃO CONTÍNUA).
MOVWF RCSTA  ; CONFIGURA RECEPÇÃO.

;*********************************************************************
;                        CONFIGURAÇÃO DO MÓDULO CAN                   *
;*********************************************************************

MOVLW B'10000000' ; REQOP2 = 1.
MOVWF CANCON  ; HABILITA MODO DE CONFIGURAÇÃO DO MÓDULO CAN.

AGUARDA_OPMODE2

BTFSS CANSTAT,OPMODE2 ; MÓDULO CAN PRONTO PARA SER CONFIGURADO ?
GOTO AGUARDA_OPMODE2 ; NÃO. AGUARDA OPMODE2 IR PARA NÍVEL ALTO.
; SIM. CONFIGURA MÓDULO CAN.

MOVLW B'00000100' ; SJW = 1TQ, BRP = 04h, TQ = 500ns, BAUD RATE =  125 kHz
MOVWF BRGCON1  ; CONFIGURA BAUD RATE CONTROL REGISTER 1.

MOVLW B'10110001' ; PHASE_SEG1 = 7 TQ, PROP_SEG = 2 TQ
MOVWF BRGCON2  ; CONFIGURA BAUD RATE CONTROL REGISTER 2.

MOVLW B'00000101' ; PHASE_SEG2 = 6 TQ
MOVWF BRGCON3  ; CONFIGURA BAUD RATE CONTROL REGISTER 3.

MOVLW B'11111111' ; TODAS AS MÁSCARAS HABILITADAS PARA O BUFFER 0
MOVWF RXM0SIDH

MOVLW B'11100000' ; TODAS AS MÁSCARAS HABILITADAS PARA O BUFFER 0
MOVWF RXM0SIDL

MOVLW B'11111111' ; TODAS AS MÁSCARAS HABILITADAS PARA O BUFFER 1
MOVWF RXM1SIDH

MOVLW B'11100000' ; TODAS AS MÁSCARAS HABILITADAS PARA O BUFFER 1
MOVWF RXM1SIDL

MOVLW B'11100110'  ; NÓ 0: ID = 11100110000
MOVWF RXF0SIDH

MOVLW B'00000000'
MOVWF RXF0SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
```

```
MOVWF RXF1SIDH

MOVLW B'00000000'
MOVWF RXF1SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF2SIDH

MOVLW B'00000000'
MOVWF RXF2SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF3SIDH

MOVLW B'00000000'
MOVWF RXF3SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF4SIDH

MOVLW B'00000000'
MOVWF RXF4SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF5SIDH

MOVLW B'00000000'
MOVWF RXF5SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXB0CON  ; RXM1:RXM0 = 00 (RECEBE TODAS MENSAGENS VÁLIDAS)

MOVLW B'00000000' ; REQOP = 000 (MODO DE OPERAÇÃO NORMAL).
MOVWF CANCON  ; PASSA O MÓDULO CAN PARA O MODO DE OPERAÇÃO NORMAL.

;*****************************************************************
;                          PROGRAMA PRINCIPAL                    *
;*****************************************************************

ENVIA_MENSAGEM_NO_1

MOVLW B'11100110'; NÓ 1: ID = 11100110001
MOVWF TXB0SIDH ; CARREGA SID10:SID3

MOVLW B'00100000'; CARREGA SID2:SID1
MOVWF TXB0SIDL ; TXB0SIDL,3 = EXIDE = 0 (ID PADRÃO)
```

```
MOVLW B'01000000'; QUADRO REMOTO (RTR = 1), 0 BYTES DE DADOS
MOVWF TXB0DLC

BSF TXB0CON,TXREQ ; ENVIA MENSAGEM QUANDO O BARRAMENTO CAN ESTIVER LIVRE

AGUARDA_ENVIO_NO_1

BTFSS PIR3,TXB0IF ; MENSAGEM ENVIADA ?
GOTO AGUARDA_ENVIO_NO_1 ; NÃO. AGUARDA ENVIO.
BCF TXB0CON,TXREQ ; SIM. LIMPA FLAG DE REQUISIÇÃO DE TRANSMISSÃO.

AGUARDA_RESPOSTA_NO_1

BTFSS PIR3,RXB0IF  ; FOI RECEBIDA MENSAGEM NO BUFFER 0 ?
GOTO AGUARDA_RESPOSTA_NO_1; NÃO. AGUARDA.
BCF PIR3,RXB0IF  ; SIM. LIMPA FLAG DE MENSAGEM RECEBIDA NO BUFFER 0.
MOVF RXB0D0,W  ; MOVE DADO DA MENSAGEM RECEBIDA PARA W.

ENVIA_DADO_PC

BTFSS TXSTA,TRMT  ; PRONTO PARA TRANSMITIR PARA O PC ?
GOTO ENVIA_DADO_PC  ; NÃO. ESPERA.
MOVWF TXREG  ; SIM. ENVIA DADO PARA O PC.
GOTO ENVIA_MENSAGEM_NO_1  ;

;*****************************************************************************
;                              FIM DO PROGRAMA                                *
;*****************************************************************************

END
```

C.2 Nó 1

```
CONFIG DEBUG = ON        ; HABILITA DEPURAÇÃO DE ERROS VIA ICD3 DEBUGGER

#INCLUDE <P18F258.INC>   ; INCLUI O ARQUIVO DE DEFINIÇÕES DO PIC18F258

ORG 0X0000  ; VETOR RESET.

MOVLB B'00001111' ; BANCO 15

;*****************************************************************************
;                       CONFIGURAÇÃO DOS PINOS DE E/S                        *
;*****************************************************************************

BSF TRISB,3  ; DEFINE RB3/CANRX COMO ENTRADA.
```

```
BCF TRISB,2  ; DEFINE RB2/CANTX/INT2 COMO SAÍDA.

;*********************************************************************
;                    INICIA CONFIGURAÇÃO DO MÓDULO CAN                 *
;*********************************************************************

MOVLW B'10000000' ; REQOP2 = 1.
MOVWF CANCON  ; HABILITA MODO DE CONFIGURAÇÃO DO MÓDULO CAN.

AGUARDA_OPMODE2

BTFSS CANSTAT,OPMODE2 ; MÓDULO CAN PRONTO PARA SER CONFIGURADO ?
GOTO AGUARDA_OPMODE2 ; NÃO. AGUARDA OPMODE2 IR PARA NÍVEL ALTO.
; SIM. CONFIGURA MÓDULO CAN.

MOVLW B'00000100' ; SJW = 1TQ, BRP = 04h, TQ = 500ns, BAUD RATE =  125 kHz
MOVWF BRGCON1  ; CONFIGURA BAUD RATE CONTROL REGISTER 1.

MOVLW B'10110001' ; PHASE_SEG1 = 7 TQ, PROP_SEG = 2 TQ
MOVWF BRGCON2  ; CONFIGURA BAUD RATE CONTROL REGISTER 2.

MOVLW B'00000101' ; PHASE_SEG2 = 6 TQ
MOVWF BRGCON3  ; CONFIGURA BAUD RATE CONTROL REGISTER 3.

MOVLW B'11111111' ; HABILITA TODAS AS MÁSCARAS DO BUFFER 0
MOVWF RXM0SIDH

MOVLW B'11100000' ; HABILITA TODAS AS MÁSCARAS DO BUFFER 0
MOVWF RXM0SIDL

MOVLW B'11111111' ; HABILITA TODAS AS MÁSCARAS DO BUFFER 1
MOVWF RXM1SIDH

MOVLW B'11100000' ; HABILITA TODAS AS MÁSCARAS DO BUFFER 1
MOVWF RXM1SIDL

MOVLW B'11100110' ; NÓ 1: ID = 11100110001
MOVWF RXF0SIDH

MOVLW B'00100000'
MOVWF RXF0SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF1SIDH

MOVLW B'00000000'
MOVWF RXF1SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).
```

```
MOVLW B'00000000'
MOVWF RXF2SIDH

MOVLW B'00000000'
MOVWF RXF2SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF3SIDH

MOVLW B'00000000'
MOVWF RXF3SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF4SIDH

MOVLW B'00000000'
MOVWF RXF4SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'00000000'
MOVWF RXF5SIDH

MOVLW B'00000000'
MOVWF RXF5SIDL  ; RXF0SIDL,3 = EXIDEN = 0 (ID PADRÃO).

MOVLW B'01100000' ; RXM1:RXM0 = 11: RECEBE TODAS AS MENSAGENS (QUADRO REMOTO)
MOVWF RXB0CON

MOVLW B'00000000' ; REQOP = 000 (MODO DE OPERAÇÃO NORMAL).
MOVWF CANCON  ; PASSA O MÓDULO CAN PARA O MODO DE OPERAÇÃO NORMAL.

;*********************************************************************
;                           PROGRAMA PRINCIPAL                       *
;*********************************************************************
INICIO

AGUARDA_MENSAGEM_NO_0

BTFSS PIR3,RXB0IF  ; FOI RECEBIDA MENSAGEM NO BUFFER 0 ?
GOTO AGUARDA_MENSAGEM_NO_0; NÃO. AGUARDA.
BCF PIR3,RXB0IF  ; SIM. LIMPA FLAG DE MENSAGEM RECEBIDA NO BUFFER 0.

ENVIA_MENSAGEM_NO_0

MOVLW B'11100110'; NÓ 0: ID = 11100110000
MOVWF TXB0SIDH ; CARREGA SID10:SID3
```

```
MOVLW B'00000000'; CARREGA SID2:SID1
MOVWF TXB0SIDL ; TXB0SIDL,3 = EXIDE = 0 (ID PADRÃO)

MOVLW B'00000001'; QUADRO DE DADOS (RTR = 0), 1 BYTE DE DADOS
MOVWF TXB0DLC

MOVLW B'01000001'; DADO = 01000001 = A (ASCII)
MOVWF TXB0D0

BSF TXB0CON,TXREQ ; ENVIA MENSAGEM QUANDO O BARRAMENTO CAN ESTIVER LIVRE

AGUARDA_ENVIO_NO_0

BTFSS PIR3,TXB0IF ; MENSAGEM ENVIADA ?
GOTO AGUARDA_ENVIO_NO_0 ; NÃO. AGUARDA ENVIO.
BCF TXB0CON,TXREQ ; SIM. LIMPA FLAG DE REQUISIÇÃO DE TRANSMISSÃO.
GOTO AGUARDA_MENSAGEM_NO_0


;*****************************************************************
;                              FIM DO PROGRAMA                    *
;*****************************************************************

END
```

Apêndice D

SOTR para Aquisição de Dados e Comunicação

Neste apêndice é apresentada a especificação de um Sistema Operacional de Tempo Real (SOTR) embarcado para gerenciamento de aquisição de dados e comunicação utilizando o microcontrolador LAMPIÃO e o módulo de acesso à rede MARIA.

A motivação deste trabalho é desenvolver um SOTR embarcado que possa ser implementado no microcontrolador LAMPIÃO. Este SOTR embarcado deve ser capaz de gerenciar a aquisição de dados realizada por um amplificador sensível à fase digital e a transferência de dados realizada pela interface de rede MARIA, via barramento CAN. O conjunto formado pelo microcontrolador LAMPIÃO, amplificador *lock-in* digital e interface de rede MARIA faz parte do sensor inteligente integrado em desenvolvimento no LDN.

Tipicamente, sistemas microcontrolados são implementados utilizando-se uma única rotina (principal) e algumas sub-rotinas (auxiliares). Basicamente, o programa principal lê as entradas do microcontrolador, executa o algoritmo de controle do processo e atualiza as saídas do microcontrolador. Esta estratégia funciona bem em sistemas "pequenos", nos quais poucas decisões são tomadas e poucos recursos são gerenciados. Além disso, esta estratégia é fácil de ser implementada, dado que a rotina principal, em geral, utiliza poucas instruções, não sendo necessário estruturar o sistema.

Entretanto, com o crescimento acelerado do número de sensores, houve um au-

mento na quantidade de informação trafegando nas redes industriais. Consequentemente, cresceu também a quantidade de decisões a serem tomadas pelos controladores (microcontroladores, controladores lógicos programáveis, etc.). Além disso, os sistemas de controle passaram a cada vez mais exigirem uma resposta em tempo real. Portanto, com o aumento na complexidade dos sistemas de controle, surge a necessidade de estruturar o sistema, de forma que diversas solicitações possam ser atendidas "em paralelo".

A utilização de um SOTR embarcado permite que problemas complexos sejam quebrados em tarefas mais simples, utilizando o conceito do "dividir para conquistar". Além disso, o sistema pode ser projetado para responder mais rapidamente a eventos mais importantes, por exemplo, a partir da definição de prioridades. Outra vantagem de se utilizar um SOTR embarcado é que uma tarefa pode ser executada enquanto outras tarefas estão esperando por algum evento, pela liberação de um recurso, etc., implementando o conceito de pseudoparalelismo. Finalmente, a utilização de um SOTR embarcado torna o projeto do sistema de controle modular, de maneira que, se o processo muda, não é necessário reprojetar todo o sistema, reescrevendo todo o programa principal. A estrutura básica (SOTR) permanece inalterada. Basta modificar as tarefas a serem executadas.

Este apêndice apresenta a especificação do SOTR embarcado proposto. Inicialmente, são definidos os requisitos de *hardware* e *software*. Em seguida, é realizada a modelagem do sistema utilizando a metodologia *Yourdon* [30], sendo discutidos os modelos ambiental e comportamental do SOTR. Finalmente, o projeto do SOTR é apresentado, discutindo-se a aplicação de diversos conceitos de Sistemas Operacionais de Tempo Real.

D.1 Objetivos

Este trabalho tem por objetivo desenvolver um SOTR embarcado capaz de: gerenciar a aquisição de dados realizada pelo amplificador *lock-in* digital; gerenciar a transmissão e a recepção de dados realizadas pela interface de rede MARIA; gerenciar a utilização de recursos, tais como o acesso à memória externa (RAM); gerenciar a execução do(s) processo(s) dependente(s) da aplicação, isto é, a(s) tarefa(s) relacionada(s) ao

algoritmo de controle do processo no qual o sensor inteligente está inserido.

D.2 Especificação do *Hardware*

O *hardware* envolvido no projeto é composto por:

- Amplificador *lock-in* digital (LOCK-IN): circuito utilizado para a medição e o condicionamento de sinais. Aplicado, por exemplo, na medição de impedâncias e em tomografia por impedância elétrica. Aciona uma interrupção periódica indicando ao microcontrolador LAMPIÃO que uma nova medição foi realizada;
- Módulo de acesso à rede (MARIA): responsável pela implementação do protocolo de rede (TCP/IP e CAN). Aciona um *flag* sempre que uma nova mensagem é recebida;
- Microcontrolador (LAMPIÃO): responsável pelo controle do processo;
- Memória externa (RAM): utilizada como memória de dados, por exemplo, para armazenamento da medição realizada pelo LOCK-IN ou da mensagem recebida por MARIA.

D.2.1 Microcontrolador LAMPIÃO

Inicialmente desenvolvido por Hércules Padilha [3] e atualmente em fase de expansão por Marco Antônio Diniz [31], o microcontrolador LAMPIÃO possui as especificações apresentadas na Tabela D.1.

Como pode ser observado na Tabela D.1, a versão atual do microcontrolador LAMPIÃO apresenta um maior número de recursos de *hardware*, como por exemplo, memória de programa, níveis de pilha e registradores (memória de dados). Entretanto, a principal diferença entre as duas versões do LAMPIÃO está no fato de que a versão atual está sendo desenvolvida para permitir a implementação de um Sistema Operacional em camadas. Neste sentido, torna-se possível, por exemplo, o acesso à pilha, a implementação de exclusão mútua, etc. Os diagramas em blocos das duas versões do microcontrolador LAMPIÃO são apresentados nas Figuras D.1 e D.2.

Tabela D.1: Especificações do microcontrolador LAMPIÃO

Versão Inicial (Padilha)	Versão Atual (Diniz)
Arquitetura Harvard	Arquitetura Harvard
2 portas de E/S de 8 bits	3 portas de E/S de 8 bits
1 temporizador guarda	1 temporizador guarda
1 relógio de tempo real	1 relógio de tempo real
2 interrupções externas	4 interrupções externas
256 posições de memória de programa	1024 posições de memória de programa
8 níveis de pilha	256 níveis de pilha
32 registradores	2048 registradores

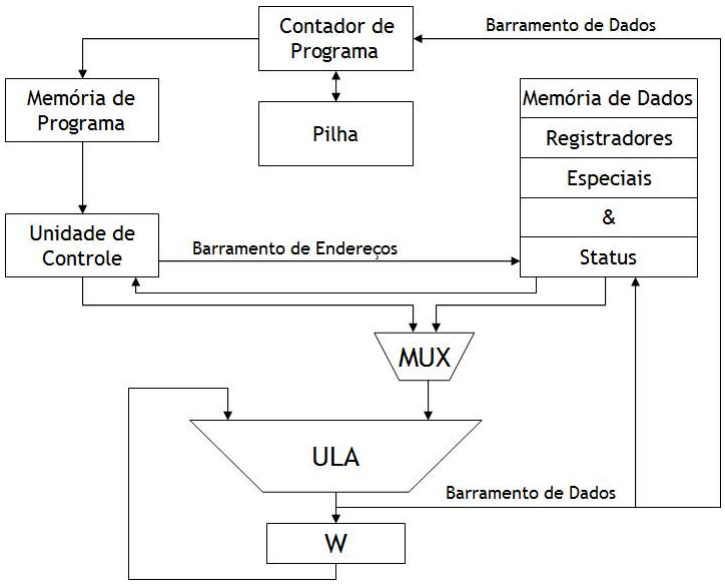


Figura D.1: Diagrama em blocos do microcontrolador LAMPIÃO (Versão Inicial) [3].

A proposta inicial deste trabalho seria projetar um SOTR embarcado a ser implementado na versão inicial do LAMPIÃO. Entretanto, do ponto de vista prático, a implementação de um Sistema Operacional na versão inicial do LAMPIÃO é inviável por limitações de hardware. Por exemplo, nenhuma instrução na versão inicial permite o acesso à pilha. Nesse sentido, será discutida a especificação de um SOTR embarcado a ser implementado na versão atual do LAMPIÃO, sem considerar, entretanto, as suas novas funcionalidades relativas a um Sistema Operacional em camadas.

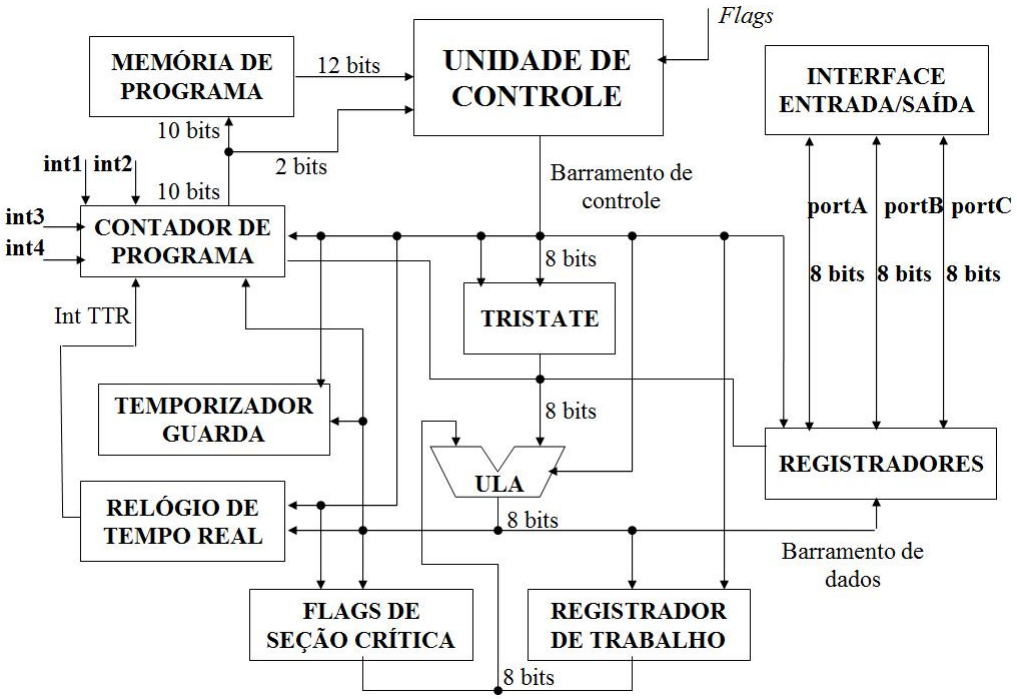


Figura D.2: Diagrama em blocos do microcontrolador LAMPIÃO (Versão Atual) [31].

D.3 Especificação do *Software* - Modelo Ambiental

Nesta seção é discutida a modelagem ambiental do sistema, de acordo com a Metodologia *Yourdon*.

D.3.1 Diagrama de Contexto

Na Figura D.3 é ilustrado o diagrama de contexto do sistema. No centro do diagrama está o SOTR a ser desenvolvido. O SOTR é responsável pelo gerenciamento dos recursos, das chamadas ao sistema, das interrupções e dos diversos processos de "usuário"envolvidos, representados no diagrama por retângulos.

D.3.2 Lista de Eventos

A lista de eventos é apresentada na Tabela D.2.

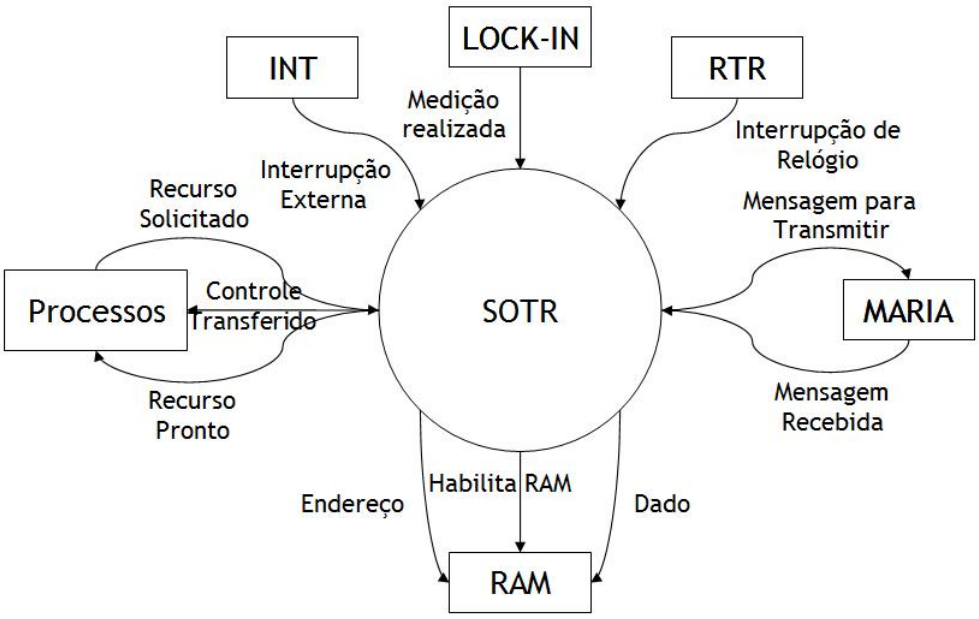


Figura D.3: Diagrama de contexto do sistema.

Tabela D.2: Lista de eventos do sistema e ações a serem realizadas.

Evento	Ação
Interrupção de relógio	Execute algoritmo de agendamento
Interrupção externa	Execute rotina de tratamento de interrupção
Medição realizada	Armazene na RAM
Mensagem recebida	Armazene na RAM
Recurso solicitado	Disponibilize recurso

D.4 Especificação do *Software* - Modelo Comportamental

Nesta seção é discutida a modelagem comportamental do sistema, de acordo com a Metodologia Yourdon.

D.4.1 Arquitetura do Sistema Operacional

A arquitetura do Sistema Operacional proposto é apresentada na Figura D.4 . O SOTR é composto por: rotinas de tratamento de chamadas ao sistema e interrupções; um algoritmo de troca de contexto; uma fila de processos e despachante.

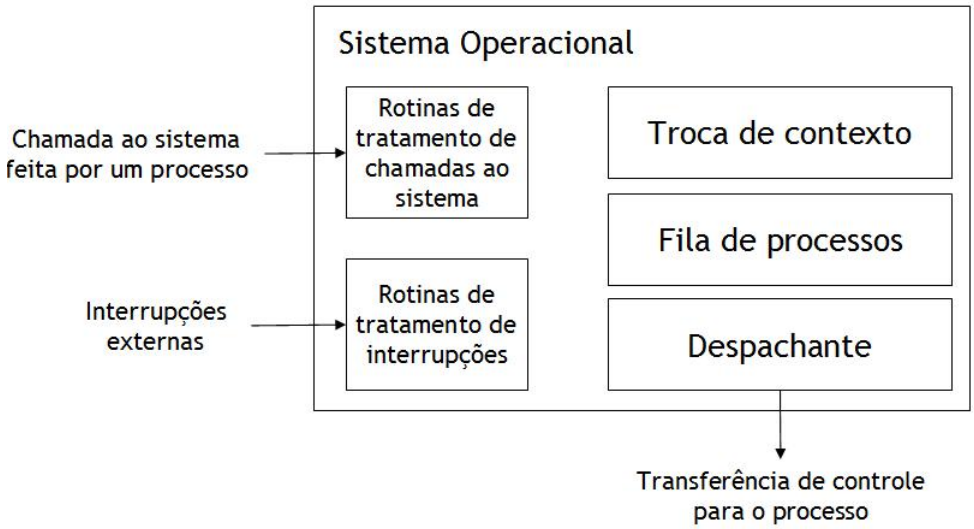


Figura D.4: Diagrama em blocos dos elementos do Sistema Operacional.

D.4.2 Tratamento de uma Interrupção de Relógio

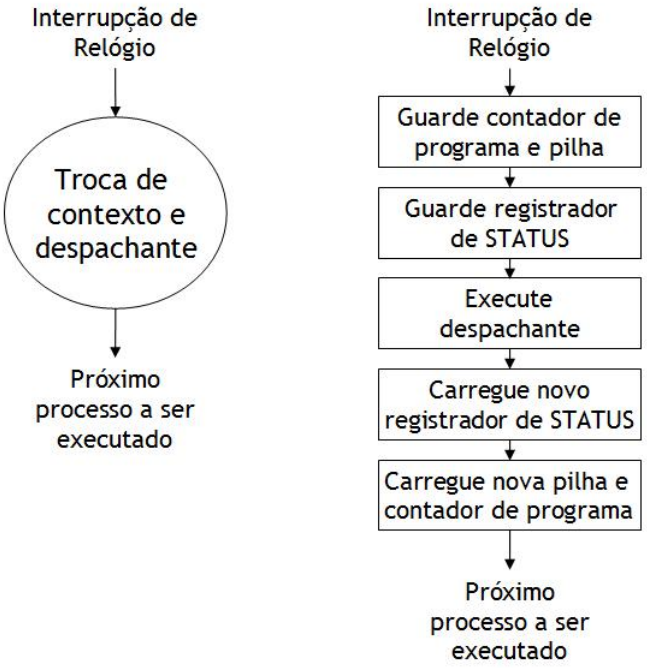


Figura D.5: Fluxograma da rotina de tratamento de uma interrupção de relógio.

Na Figura D.5 é ilustrado o fluxograma da rotina de tratamento de uma interrupção de relógio. Sempre que ocorre uma interrupção de relógio, o algoritmo de agendamento (despachante) é executado e a troca de contexto é realizada ou não, dependendo da fila de processos prontos. Nesse sentido, o SOTR guarda o contador de programa, a pilha e o registrador de STATUS. Em seguida, o SOTR executa o

despachante, que decide qual será o próximo processo a ser executado. Finalmente, o SOTR carrega o novo registrador de STATUS, a nova pilha e o novo contador de programa.

D.4.3 Tratamento de uma Interrupção Externa

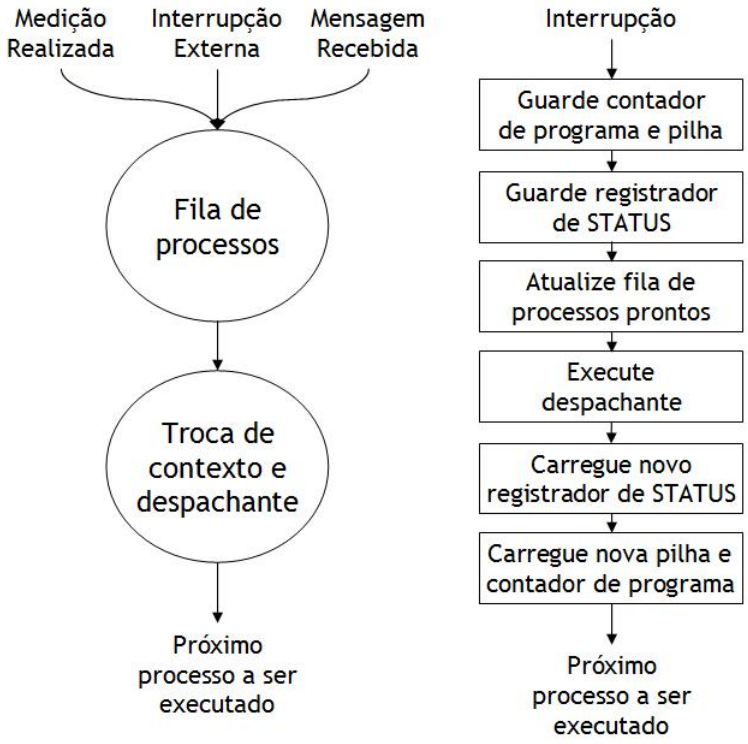


Figura D.6: Fluxograma da rotina de tratamento de uma interrupção externa.

Na Figura D.6 é ilustrado o fluxograma da rotina de tratamento de uma interrupção externa, de uma medição realizada e de uma mensagem recebida ocorrem de forma similar à interrupção de relógio. Entretanto, antes do algoritmo de agendamento ser realizado, a fila de processos prontos é atualizada, incluindo o processo relativo ao evento (interrupção externa, medição realizada ou mensagem recebida).

D.4.4 Tratamento de uma Chamada ao Sistema para Solicitação de um Recurso

Na Figura D.7 é apresentado o fluxograma da rotina de tratamento de uma chamada ao sistema para solicitação de recurso. Nenhum processo possui acesso direto a recursos, tais como memória externa. Esse acesso é feito por meio de uma chamada

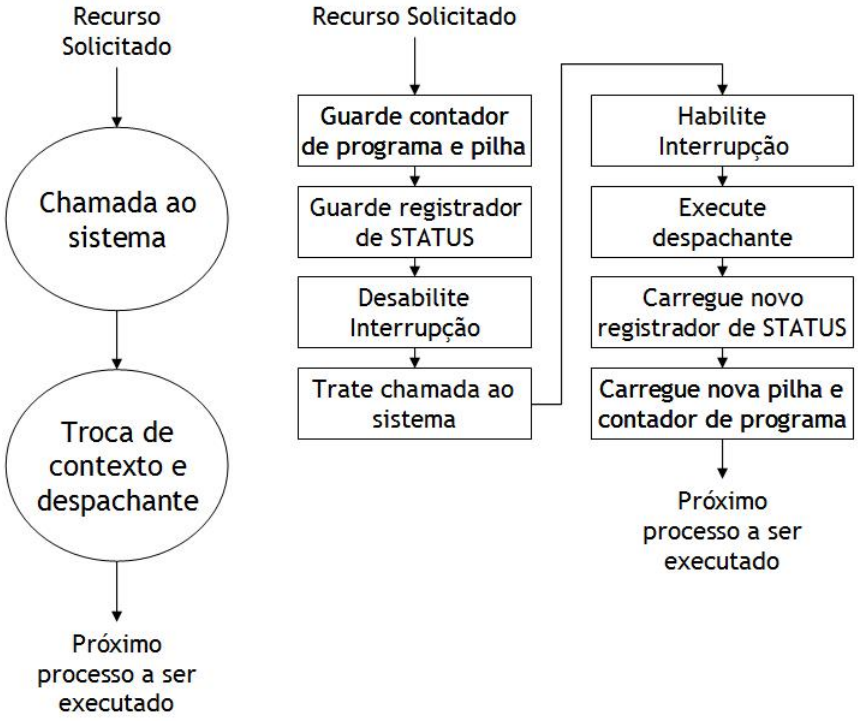


Figura D.7: Fluxograma da rotina de tratamento de uma chamada ao sistema para solicitação de recurso.

ao sistema. Portanto, quando um recurso é solicitado, a rotina de tratamento da chamada ao sistema é executada. Em seguida, é realizada a troca de contexto.

Para implementar a exclusão mútua, é utilizada a técnica de desabilitar interrupções. Tipicamente, o próprio processo é responsável por desabilitar e habilitar as interrupções. Entretanto, pode acontecer de um determinado processo não habilitar as interrupções novamente. Nesse caso, nem mesmo a interrupção de relógio será tratada. Conseqüentemente, o Sistema Operacional não terá novamente o controle.

Para evitar este problema, sempre que uma chamada ao sistema é realizada o próprio Sistema Operacional desabilita as interrupções. Após a chamada ao sistema ser tratada, o Sistema Operacional imediatamente habilita as interrupções, antes de executar o despachante.

D.5 Estrutura do Sistema Operacional

A estrutura do SOTR embarcado é apresentada na Figura D.8. Trata-se de um Sistema Operacional Monolítico, composto por procedimentos principais e de serviço.

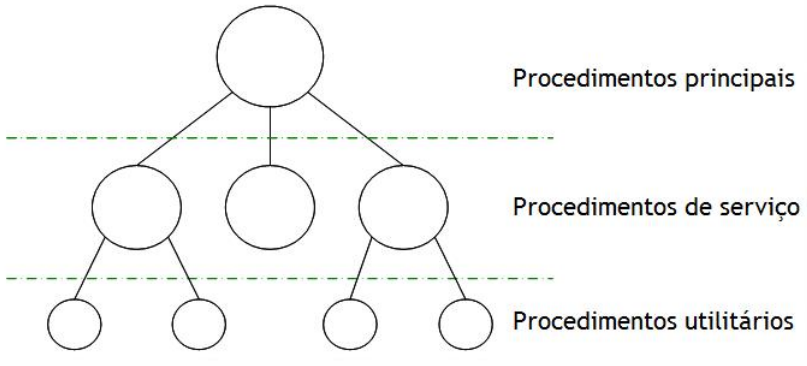


Figura D.8: Estrutura do Sistema Operacional Monolítico.

Os procedimentos utilitários são auxiliares aos de serviço, não sendo considerados neste projeto, dada a simplicidade do sistema proposto.

Os procedimentos principais são aqueles dependentes da aplicação. Podem implementar, por exemplo:

- O algoritmo de controle do processo;
- O algoritmo de tratamento da medição realizada pelo LOCK-IN;
- O algoritmo de tratamento das mensagens recebidas por MARIA.

Os procedimentos de serviço são as chamadas ao sistema e as rotinas de tratamento de interrupções. As chamadas de sistema, por sua vez, são classificadas em:

- Chamadas ao sistema para gerenciamento de E/S: aquisição do resultado do LOCK-IN, aquisição da mensagem recebida por MARIA, transmissão de mensagens por MARIA;
- Chamadas ao sistema para gerenciamento de recursos: acesso à memória RAM externa.

D.6 Processos

D.6.1 Diagrama de Estados dos Processos

Na Figura D.9 é ilustrado o diagrama de estados do Sistema Operacional. Neste projeto são considerados três estados dos processos: pronto, executando e bloqueado. As transições de estados são funções do sistema operacional e estão descritas a seguir.

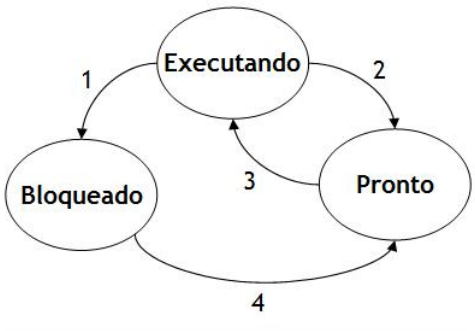


Figura D.9: Diagrama de estados dos processos.

- O processo é bloqueado esperando por um recurso que ainda não está disponível;
- O tempo de execução do processo terminou. O despachante seleciona outro processo a ser executado;
- Um novo *slot* de tempo é dedicado à execução do processo. O despachante seleciona esse processo para ser executado;
- O processo que estava bloqueado esperando recurso passa ao estado pronto quando o recurso estiver disponível.

D.6.2 Implementação de Processos

O Sistema Operacional mantém uma tabela de processos, com uma entrada por processo, com todos os dados necessários à execução dos processos. Esta tabela possui:

- Identificador do processo;
- Estado do processo;
- Contador de programa;
- Ponteiro da pilha;
- Alocação de memória.

D.6.3 Comunicação entre Processos

Como foi discutido anteriormente, a exclusão mútua é implementada desabilitando-se as interrupções. Esta técnica é utilizada neste projeto, uma vez que a implementação

inicial do LAMPIÃO é simples, com poucas instruções. Além disso, os processos executáveis estarão gravados na memória de programa antes do LAMPIÃO ser inserido na planta, de forma que novos processos não serão criados dinamicamente.

Tipicamente, nesta técnica, cada processo desabilita todas as instruções imediatamente após entrar em sua seção crítica, reativando-as imediatamente após sair dela. Entretanto, neste projeto, para garantir o funcionamento do sistema, as interrupções são desabilitadas pelo SOTR ao entrar na rotina de atendimento à chamada de sistema, sendo habilitadas novamente pelo SOTR ao sair da rotina de tratamento da chamada ao sistema (vide Figura D.7).

D.7 Algoritmo de Agendamento (Despachante)

Neste projeto é utilizado o agendamento tipo carrossel *(round-robin)* com prioridade. Este algoritmo de agendamento é preemptivo, atribuindo um intervalo de tempo *(slot* de tempo ou *quantum)* para a execução de cada processo. Um novo slot de tempo de execução é iniciado sempre que ocorre uma interrupção do relógio de tempo real.

A preempção é realizada em três situações:

- Quando o tempo de execução acaba;
- Quando o processo bloqueia ou termina;
- Quando ocorre uma interrupção.

O tempo de execução é determinado por:

- Frequência de relógio do LAMPIÃO;
- Constante de tempo do processo;
- Tempo de atualização do resultado do LOCK-IN;
- Tempo de transmissão e recepção de mensagens por MARIA;
- Tempo utilizado para troca de contexto.

São definidos cinco níveis (filas) de prioridades. Os níveis 4 a 1 são alocados às interrupções externas. O nível 0 é alocado aos processos de "usuário". Neste algoritmo de agendamento, a prioridade é decrementada cada vez que o processo é executado e o processo passa para uma fila de menor prioridade. O número máximo de processos em cada fila é apresentado na Tabela D.3. N é o número de processos de "usuário".

Tabela D.3: Número de processos por nível de prioridade.

	Nível 4	Nível 3	Nível 2	Nível 1	Nível 0
Número máximo de processos	1	2	3	4	N+4

O fluxograma de operação do despachante é apresentado na Figura D.10.

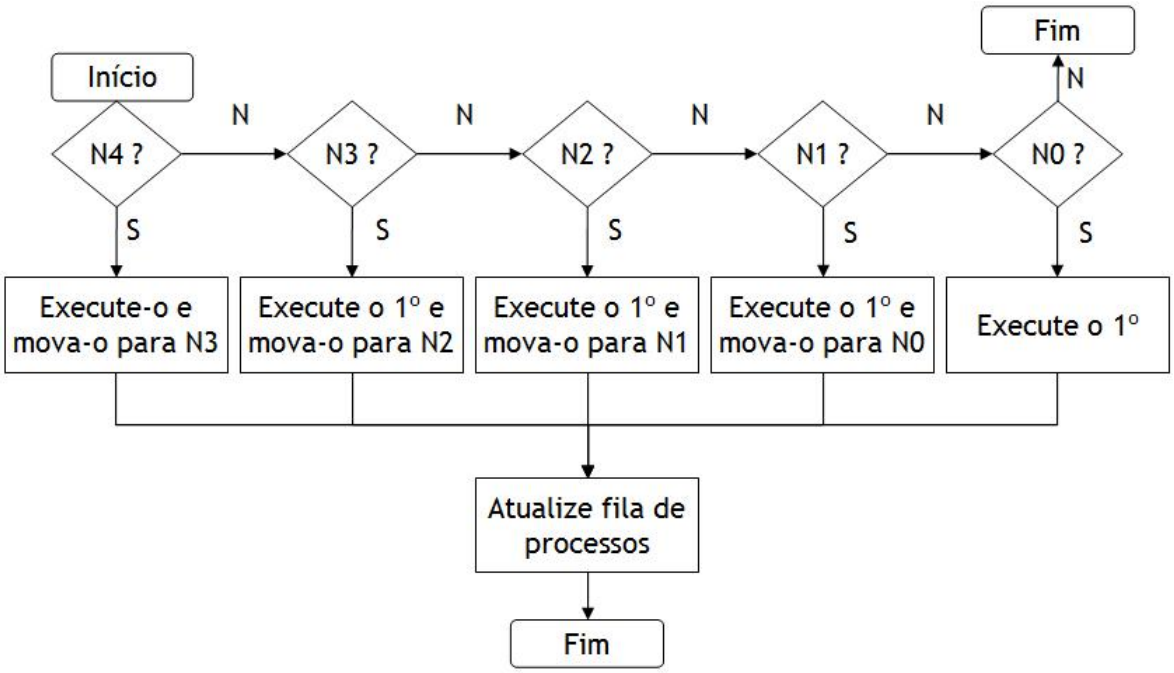


Figura D.10: Fluxograma de operação do despachante.

D.7.1 Estimativa do *Quantum*

Para estimar o valor do *quantum* de tempo, é necessário avaliar os fatores listados no estudo do despachante. Uma análise rápida é realizada a seguir.

1. Frequência de relógio do LAMPIÃO:

- $f_{Lampiao} > 1MHz$;
- Tempo de execução de 01 instrução: $T_{Lampiao} < 1\mu s$;
- Consideremos $f_{Lampiao} = 4MHz$, isto é, $T_{Lampiao} = 0,25\mu s$.

2. Constante de tempo do processo (planta):

- $T = 1ms$.

3. Tempo de atualização do resultado do LOCK-IN:

- Frequência do sinal de referência: $f_{Lock-in} = 10kHz$;
- Tempo de atualização da saída: $T_{Lock-in} = 100\mu s$.

4. Tempo de recepção e transmissão de mensagens:

- Número de bits por mensagem: $N_{bits} > 60$;
- Taxa de transmissão: $f_{Maria-CAN} < 1Mbps$;
- Tempo de transmissão: $T_{Maria-CAN} > 60\mu s$.

5. Tempo utilizado na troca de contexto:

- Guardar contador de programa e pilha: 2 instruções;
- Guardar registrador de STATUS: 2 instruções;
- Executar despachante: 30 instruções (pior caso);
- Carregar novo registrador de STATUS: 2 instruções;
- Carregar novo registrado de programa e pilha: 2 instruções;
- Tempo total: $T_{contexto} = 10\mu s$ (considerando $T_{Lampiao} = 0,25\mu s$).

Embora nenhum processo específico (planta) tenha sido definido neste trabalho, é razoável estimar uma constante de tempo da ordem de alguns milésimos de segundo, uma vez que grande parte dos processos industriais apresenta constante de tempo desta magnitude.

Embora o tempo de transmissão e recepção de mensagens seja aleatório (dependendo do tamanho da mensagem, da taxa de transmissão e da necessidade de enviar uma mensagem na rede), foi considerado no cálculo anterior que a mensagem mais curta está sendo recebida na taxa de transmissão mais rápida, durante todo o tempo. Este seria o pior caso no qual o sensor inteligente estaria inserido, exigindo uma resposta mais rápida do SOTR. Entretanto, esse caso é impossível de acontecer na prática. Portanto, é uma boa estimativa considerar que o evento mais rápido a ocorrer é a interrupção do LOCK-IN, indicando que uma nova medição foi realizada.

Após examinar as restrições temporais acima, acredita-se que seja razoável utilizar um *quantum* de $20\mu s$. Nesse caso, cada processo poderá executar até 80 instruções. Além disso, até 3 processos poderão ser executados durante o intervalo de tempo no qual espera-se a interrupção do LOCK-IN, o que deve ser suficiente para executar o algoritmo de controle e o tratamento de mensagens "paralelamente".

Apêndice E

Publicações

Desta dissertação, foram publicados os seguintes artigos:

1. J. E. O. Reges e E. J. P. Santos. VHDL Digital Lock-in Amplifier for Smart Sensors. *XIV Iberchip*, 2008.

2. J. E. O. Reges e E. J. P. Santos. A VHDL CAN Module for Smart Sensors. *Southern Programmable Logic Conference*, 2008.

Bibliografia

[1] The Free Dictionary. http://encyclopedia2.thefreedictionary.com/smart+sensor, acessado em 14/06/2010.

[2] Wikipedia. A Enciclopédia Livre. http://en.wikipedia.org/wiki/Smart_transducer, acessado em 14/06/2010.

[3] H. Padilha. *Descrição VHDL de Microcontrolador para Sensores Inteligentes.* Trabalho de Graduação. Universidade Federal de Pernambuco, 2003.

[4] D. Dubey. *Smart Sensor. M. Tech. Credit Seminar Report. Electronic Systems Group. EE Dept. IIT Bombay*, 2002.

[5] National Institute of Standards and Technology. http://ieee1451.nist.gov/, acessado em 14/06/2010.

[6] F. K. Tani. *Proposta de Desenvolvimento de Transdutores Inteligentes baseados na Norma IEEE 1451 aplicados a Redes Lonworks.* Dissertação de Mestrado. Escola Politécnica da Universidade de São Paulo, 2006.

[7] R. D. Regazzi, P. S. Pereira e M. F. da Silva Jr. *Soluções Práticas de Instrumentação e Automação.* Rio de Janeiro, 2005.

[8] P. Vieira. *Redes de Computação Industrial.* Notas de Aula do Curso de Formação em Automação Industrial. Petrobras. Rio de Janeiro, 2009.

[9] J. E. Thomas, et al. *Fundamentos de Engenharia de Petróleo.* Rio de Janeiro, 2001.

[10] E. J. P. Santos, P. L. Guzzo, A. H. Shinohara, et al. *Sensores Inteligentes de Vazão, Pressão e Temperatura para Monitoramento de Fluxos Multifásicos (Petróleo, Água e Gás).*

[11] S. L. Ceccio and D. L. George. A review of electrical impedance techniques for the measurement of multiphase flow. *J. of Fluids Eng.*, 118, 391-399, 1996.

[12] G. J. Saulnier, R. S. Blue, J. C. Newell, D. Isaacson, and P. M. Edic. Electrical impedance tomography. *IEEE Signal Processing Magazine*, 11, 31-43, 2001.

[13] H. Lemonnier. Multiphase Instrumentation: The Keystone of Multidimensional Multiphase Flow Modeling. *Exper. Thermal and Fluid Sci.*, 15, 154-162, 1997.

[14] Z. Szczepanik and Z. Rucki. Frequency Analysis of Electrical Impedance Tomography System. *IEEE Trans on Inst. and Meas.*, 49, 844-851, 2000.

[15] D. Holder. Electrical Impedance Tomography. *Inst of Physics Pub Inc*, 2005.

[16] D. L. George, K. A. Shollenberger, J. R. Torczynski, T. J. O'Hern e S. L. Ceccio. Three-phase Material Distribution Measurements in a Vertical Flow using gamma-densitomwtry tomography and electrical-impedance tomography. *Int. J. Multiphase Flow*, 27, 1903-1930, 2001.

[17] Bloodshed Software. http://www.bloodshed.net, acessado em jan./2004.

[18] Terminal v.19b. Disponível em http://bray.velenje.cx/avr/terminal, set./2006.

[19] Xilinx. Basic FPGA. Architecture. Xilinx Inc, 2005.

[20] Xilinx. Spartan-3E FPGA FAmily: Complete Data Sheet. Xilinx Inc, 2006.

[21] Xilinx. Spartan-II 2.5V FPGA Family: Complete Data Sheet. Xilinx Inc, 2003.

[22] R. d'Amore. *VHDL - Descrição e Síntese de Circuitos Digitais.* LTC, 2005.

[23] E. J. P. Santos. *Eletrônica Analógica Integrada e Aplicações.* UFPE, 2003.

[24] T. L. Keiser. *The DAS-20 Software Library and its use for Control System Implementation.* 1995.

[25] D. E. Johnson, J. L. Hilburn, J. R. Johnson. *Fundamentos de Análise de Circuitos Elétricos*. PHB, 1990.

[26] A. V. Oppenheim, A. S. Willsky, S. H. Nawab. *Signals & Systems*. Prentice Hall, 1996.

[27] Analog Devices. AD5447 Data Sheet. Analog Devices Inc, 2005.

[28] CAN in Automation. http://www.can-cia.org, acessado em 14/06/2010.

[29] R. Bosch. *CAN Specification*. Version 2.0, Parts A and B, Sept. 1991.

[30] E. Yourdon. *Analisis Estructurado Moderno*. Prentice-Hall, 1993.

[31] M. A. Diniz. *LAMPIÃO - Microcontrolador para Sensores Inteligentes*. UFPE, 2008.

[32] E-Sensors. http://www.eesensors.com/, acessado em 14/06/2010.

[33] Smart Sensors Systems. http://www.smartsensorsystems.com/, acessado em 14/06/2010.

[34] Sensors Synerg., http://www.sensorsynergy.com/, acessado em 14/06/2010.

[35] Industrial Embedded Systems. http://www.industrial-embedded.com/, acessado em 14/06/2010.

[36] A. Hac. *Wireless Sensor Network Designs*. John Wiley & Sons Ltd, 2003.

[37] L. B. Torri. *A Norma IEEE 1451 aplicada a Redes Heterogêneas de Sensores sem Fio*. Trabalho de Conclusão de Curso. Universidade Federal de Santa Catarina, 2008.

[38] A. S. Tanembaum. *Redes de Computadores*. Quarta Edição, 2003.

[39] Jener T. L. e Silva. *Instrumentação virtual para microscopia de varredura*. Dissertação de Mestrado. Universidade Federal de Pernambuco, 2002.

[40] A. Restelli, R. Abbiati e A. Geraci. Digital field programmable gate array-based lock-in amplifier for high performance photon counting applications. *Rev. Sci. Instrum.*, 2005.

[41] M. O. Sonnaillon e F. J. Bonetto. A low-cost, highperformance, digital signal processor-based lock-in amplifiercapable of measuring multiple frequency sweeps simultaneously. *Rev. Sci. Instrum.*, 2005.

[42] P-A. Probst e A. Jaquier. Multiple-channel digital lock-in amplifier with PPM resolution. *Rev. Sci. Instrum.*, 1994.

[43] P. K Dixon e L. Wu. Broadband digital lock-in amplifier techniques. *Rev. Sci. Instrum.*, 1989.

[44] J. S. Scofield. A Frequency-Domain Description of a Lockin Amplifier. *American Journal of Physics*,1994.

[45] J. E. O. Reges e E. J. P. Santos. Amplificador Lock-in Digital utilizando Placa de Aquisição de Dados e MATLAB. *XXXIII Congresso Brasileiro de Ensino de Engenharia*. Campina Grande, 2005.

[46] S. Corrigan. *Introduction to Controller Area Network (CAN)*. Texas Instruments Inc, Application Report, 2002.

[47] L. Stagnaro. *HurriCANe - Free VHDL CAN Controller Core*. European Space Agency, 2000.

[48] R. Stoneking. *A Simple CAN Node Using the MCP2510 and PIC12C67*. Microchip Technology Inc, Application Note, 2002.

[49] A. Amory and J. P. Júnior. *Sistema Integrado e Multiplataforma para controle remoto de residências*. Pontifícia Universidade Católica do Rio Grande do Sul, 2000.

[50] R. J. Tocci e N. S. Widmer. *Sistemas Digitais - Princípio e Aplicações*. Prentice-Hall, 2004.

[51] R. Airiau, J. M. Bergé e V. Olive. *Circuit Synthesis with VHDL*. Kluwer Academic Publishers, 1994.

[52] J. Bhasker. *A VHDL Synthesis Primer*. Star Galaxy Publishing, 1996.

[53] A. V. Oppenheim, A. S. Willsky e S. H. Nawab. *Signals and Systems*. Prentice Hall, 1996.

[54] C. H. Chen. *Signal Processing Handbook*. New York, Marcel Dekker, 1988.

[55] E. C. Ifeachor and B. W. Jervis. *Digital Signal Processing - A Practical Approach*. Addison-Wesley, 1993.

[56] J. H. Mcclellan, J. C. Burrus, et al. *Computer-Based Exercises for Signal Processing using MATLAB 5*. New Jersey: Prentice Hall, 1998.

[57] E. R. Davies. *Electronics Noise and Signal Recovery*. Academic Press, 1993.

[58] A. S. Tanembaum. *Operating Systems - Design and Implementation*. Second Edition, 1997.