PLATAFORMA RECONFIGURÁVEL PARA TILT TEST NAVAL

Alexandre Bazyl Zacarias de França

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Antonio Carneiro de Mesquita Filho

Rio de Janeiro
Setembro de 2013

PLATAFORMA RECONFIGURÁVEL PARA TILT TEST NAVAL

Alexandre Bazyl Zacarias de França

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Antonio Carneiro de Mesquita Filho, Dr.d'Etat

Prof. Jorge Lopes de Souza Leão, Dr.Ing

Prof. Paulo de Tarso Themistocles Esperança, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2013

França, Alexandre Bazyl Zacarias de

Plataforma Reconfigurável para Tilt Test Naval/Alexandre Bazyl Zacarias de França. – Rio de Janeiro: UFRJ/COPPE, 2013.

XV, 130 p.: il.; 29, 7cm.

Orientador: Antonio Carneiro de Mesquita Filho

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2013.

Referências Bibliográficas: p. 103 – 106.

1. Tilt. 2. Tilt-Test. 3. Inclinômetro. 4. FPGA. 5. VHDL. 6. SoPC. I. Mesquita Filho, Antonio Carneiro de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Dedico este trabalho a minha família, amigos, professores e todas as outras pessoas que, de uma forma ou de outra, me ajudaram a crescer.

Agradecimentos

Aos meus pais Antonio e Garlete (*in memoriam*), minha gratidão por tudo que fizeram por mim ao longo de minha vida. Desejo ter sido merecedor do esforço dedicado por vocês em todos os aspectos, especialmente quanto à minha formação. Obviamente, não poderia deixar de mencionar as minhas irmãs, Adriana e Daniele.

À minha esposa Hiromi Lara por compartilhar comigo momentos de dedicação e algumas ausências para a conclusão deste trabalho. Agradeço ao seu amor, carinho, amizade e compreensão.

Ao meu orientador Antonio Mesquita, sempre de bom humor e que não mede esforços em ajudar no que for preciso.

Aos amigos e colegas do Centro de Apoio a Sistemas Operativos, que contribuíram com informações valiosas e com os quais tive a oportunidade de aprender muito.

Por fim, gostaria de agradecer aos amigos da COPPE, alunos de mestrado e doutorado, pela ajuda e grato convívio no dia-a-dia.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PLATAFORMA RECONFIGURÁVEL PARA TILT TEST NAVAL

Alexandre Bazyl Zacarias de França

Setembro/2013

Orientador: Antonio Carneiro de Mesquita Filho

Programa: Engenharia Elétrica

Os avanços obtidos nos últimos anos na tecnologia de *Field-Programmable Gate Array* (FPGA), resultam em dispositivos capazes de suportar a implementação de um sistema computacional completo, superando as restrições de área e potência consumida. Nesta vertente, os FPGAs tornam-se uma plataforma promissora para este tipo de sistema, com baixo custo e tempo de desenvolvimento também relativamente baixo. Assim sendo, esta tese apresenta que os altos requisitos de alinhamento e paralelismo das plataformas dos sensores e armas de um navio de guerra, aliado aos custos e tempo despendidos para a docagem, medições e providência das medidas corretivas, motiva para o desenvolvimento de uma solução automatizada, portátil e de baixo consumo. Partindo de uma implementação piloto, o trabalho procura demonstrar a viabilidade da migração de uma ou diversas funcionalidades de um sistema computacional para dispositivos reconfiguráveis a fim de obter um sistema robusto, flexível e de bom desempenho e que possa substituir, no todo ou em parte, um sistema computacional convencional facilitando o trabalho a bordo dos navios.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

RECONFIGURABLE PLATFORM FOR NAVAL TILT TEST

Alexandre Bazyl Zacarias de França

September/2013

Advisor: Antonio Carneiro de Mesquita Filho

Department: Electrical Engineering

The advances gained in the last years in *Field-Programmable Gate Arrays* (FPGA), result in devices capable of sustaining an implementation of a complete computer system, upscaling the limits of area and consumed power. In this aspect, the FGPAs turn into a promissing platform for this kind of system, with its low cost and relatively low development time frame. This way, this thesis presents that the high requirements of alignment and parallelism of sensors platforms and warships weaponry, allied with the costs and time for docking, measurement and providence of corrective actions, motivates the development of an automated, portable and low power consumption solution. Starting off with a primary implementation, this work seeks to demonstrate the viability of migrating from one or several functionalities of a computer software to reconfigurable devices, in order to achieve a robust, flexible and high performance system, substituting partially or the entire conventional computer system, making the work on board the ships easier.

Sumário

Lista de Figuras x

Lista de Tabelas xii

Lista de Abreviaturas xiii

1 Introdução 1

2 Tilt Test 5

2.1 Representação Gráfica da Inclinação entre Planos 7

2.2 Tilt Test com Clinômetros de Bolha 8

2.3 Tilt Test Eletrônico 11

2.3.1 Filtro Digital 15

2.3.2 Desenvolvimento Matemático 16

2.3.3 Modelagem 20

2.3.4 Conclusão do Tilt Test Eletrônico 22

2.4 Considerações e Motivação para um Novo Protótipo 23

3 Plataforma Reconfigurável **26**

3.1 Dispositivos Reconfiguráveis 27

3.2 VHDL 31

3.3 SoPC 32

3.4 Plataforma de Desenvolvimento 34

4 Implementação **39**

4.1 Processador *soft-core* 39

4.1.1 Processador MB-Lite 43

4.2 Barramento de Comunicação 46

4.2.1 Barramento Wishbone 47

4.3 PLL 53

4.4 Mapeador de Memória 56

4.5 Memória RAM 60

4.6 Memória ROM 62
4.7 Controlador de Interrupção 64
4.8 UART 66
4.8.1 Módulo de Recepção 68
4.8.2 Gerador de *Baud Rate* 70
4.8.3 Módulo de Transmissão 70
4.8.4 Módulo Completo de UART 71
4.9 Controlador de Vídeo 75
4.9.1 Circuito Gerador de Sincronismo 76
4.9.2 Circuito Gerador de Pixel 79
4.9.3 Circuito Gerador de Caracteres 80
4.9.4 Controlador com *Frame Buffer* 83
4.9.5 Controlador Básico 86
4.9.6 Touch Panel 88
4.10 Controlador do LCD 16x2 91
4.11 Timers 93
4.12 Periféricos Básicos de E/S 93
4.12.1 LEDs 94
4.12.2 Hex 7 Segmentos 94
4.12.3 Botões 95
4.13 Módulo de *Debug* 95

5 Conclusões **99**

Referências Bibliográficas **103**

A Módulo Serial RS-232 **107**

B Controlador do Módulo SDRAM **108**

C Controlador do Módulo SSRAM **117**

D Controlador do *Touch Panel* **121**

E Controlador do LCD 16x2 **125**

Lista de Figuras

1.1 Exemplo de um Sistema de Combate 2
1.2 Interligação do Sistema de Combate 3
1.3 Alinhamento do Sistema 4

2.1 Exemplos de *Master Level* 6
2.2 Inclinação relativa entre dois planos 7
2.3 Exemplo Ponto Alto 8
2.4 Diagrama Vetorial 8
2.5 Escoramento Recomendado para a Condição de *Tilt Test* 9
2.6 Clinômetro de Bolha 10
2.7 Posicionamento dos Clinômetros 11
2.8 Inclinômetro modelo T7 13
2.9 Dados Brutos do Sensor T7 14
2.10 Histograma (densidade) das Medições 14
2.11 Aplicação de Filtros Recursivos 15
2.12 Aplicação de Filtro Polinomial 16
2.13 Comparação entre Sensor e Giro 17
2.14 Representação da Amplitude 20
2.15 Janelas do *Tilt Test* Eletrônico 21
2.16 Comparação do Método *Tradicional* x *Automatizado* 22
2.17 Gabinete do *Tilt Test* Eletrônico 23
2.18 Sensor de Dois Eixos 24

3.1 Arquitetura de um FPGA 28
3.2 Fluxo de Programação de um FPGA 30
3.3 Exemplo de um SoC fabricado pela Lucent 32
3.4 Placa de Desenvolvimento Terasic DE2-70 35
3.5 Diagrama de Blocos da placa DE2-70 36
3.6 LCD Touch Panel Module 38

4.1 Diagrama de Blocos do CASoPC 40
4.2 Formato adotado no MicroBlaze e MB-Lite 43

4.3 Estrutura do MB-Lite 44
4.4 Estrutura do Decodificador de Endereços 45
4.5 Esquema de Interconexão de Barramento 46
4.6 Interconexão Ponto-a-Ponto 48
4.7 Interconexão com Barramento Compartilhado 49
4.8 Conexão Padrão entre *Master-Slave* 49
4.9 Ciclo de Leitura e Escrita 52
4.10 Diagrama de um PLL Básico 54
4.11 Sinais dos PLLs do Cyclone II 55
4.12 E/S Mapeado em Memória 57
4.13 Diagrama Funcional da Memória SDRAM 61
4.14 Diagrama dos Módulos da ROM 63
4.15 Controlador de Interrupção 65
4.16 Diagrama do Módulo RS-232 67
4.17 Transmissão de um Byte 68
4.18 Receptor UART 69
4.19 Diagrama em Blocos do Módulo UART 71
4.20 Diagrama Simplificado de um Controlador de Vídeo 76
4.21 Diagrama de Temporização da Varredura Horizontal 77
4.22 Diagrama de Temporização da Varredura Vertical 78
4.23 Circuito de Geração de Caracteres 82
4.24 Diagrama do Controlador de Vídeo usando SSRAM 84
4.25 Regeneração de um Sinal Lento 85
4.26 Diagrama do Controlador de Vídeo usando DPRAM 86
4.27 Interface Serial do Módulo LCD e *Touch Panel* 89
4.28 Temporização da Interface Serial 89
4.29 Temporização do LCD para Escrita 91
4.30 Diagrama das Chaves *Push-Button* 95
4.31 Conexão do Módulo RS232_syscon ao Barramento Mestre 97

5.1 *Tilt Test* Naval 101

A.1 Diagrama Esquemático do Módulo RS-232 107

Lista de Tabelas

2.1 Comparação dos Sensores 25

3.1 Comparação entre SoPC, SoC e GPP 33

4.1 Características dos Processadores Avaliados 41
4.2 Descrição dos Sinais do PLL 55
4.3 Registradores do Controlador de Interrupção 66
4.4 Composição dos Registradores 66
4.5 Registradores da UART 74
4.6 Temporização dos Modos VGA e SVGA 78
4.7 Registradores de Vídeo 87
4.8 Registradores do *Touch Panel* 90
4.9 Temporização do LCD 91
4.10 Registradores do Módulo LCD 16x2 92
4.11 Registradores dos Contadores 93
4.12 Composição do Registrador de Controle 93
4.13 Registradores dos LEDs 94
4.14 Registradores dos Displays de 7 Segmentos 94
4.15 Registradores das Chaves e Botões 96
4.16 Comandos do Módulo de *Debug* 97
4.17 Respostas Geradas pelo Módulo de Debug 98

Lista de Abreviaturas

ADC	Analog-to-Digital Converter, p. 38
AMBA	Advanced Microcontroller Bus Architecture, p. 47
ASIC	Application-Specific Integrated Circuit, p. 26
BGA	Ball Grid Array, p. 34
CASOP	Centro de Apoio a Sistemas Operativos, p. 12
CPLD	Complex Programmable Logic Device, p. 27
CRT	Cathode Ray Tube, p. 76
CVBS	Composite Video Broadcast Signal, p. 38
DAC	Digital-to-Analog Converter, p. 37
DMA	Direct Memory Access, p. 37
DPRAM	Dual-Port RAM, p. 86
FIFO	First In, First Out, p. 71
FPGA	Field-Programmable Gate Array, p. 27
FPS	Frames per Second, p. 79
FSM	Finite-State Machine, p. 84
GPP	General-Purpose Processor, p. 26
HDL	Hardware Description Language, p. 29
IAR	Interrupt Acknowledge Register, p. 65
ID	Instruction Decode, p. 43, 66
IER	Interrupt Enable Register, p. 65

IF	Instruction Fetch, p. 43
IP	Intellectual Property, p. 34
IRQ	Interrupt Request, p. 64
ISR	Interrupt Service Register, p. 65
IrDA	Infrared Data Association, p. 38
LCD	Liquid Crystal Display, p. 38
LCD	Liquid-Crystal Display, p. 76
LE	Logic Elements, p. 35
LSB	Least Significant Bit, p. 82
LTM	LCD Touch Panel Module, p. 38
M4K	Dual-Port 4Kbits Memory Block, p. 35
MER	Master Enable Register, p. 65
MMU	Memory Management Unit, p. 42
MPU	Memory Protection Unit, p. 42
MSB	Most Significant Bit, p. 82
NMI	Non-Maskable Interrupt, p. 64
PAL	Programmable Array Logic, p. 27
PC	Program Counter, p. 44
PFD	Phase-Frequency Detector, p. 55
PIC	Programmable Interrupt Controller, p. 65
PIO	Programmed I/O, p. 37
PLL	Phase-Locked Loops, p. 28
PPR	Plano de Passeio de Roletes, p. 6
RAM	Random Access Memory, p. 60
RISC	Reduced Instruction Set Computing, p. 40

ROM	Read-Only Memory, p. 62
RTC	Real-Time Clock, p. 93
SDRAM	Synchronous Dynamic RAM, p. 35
SMA	SubMiniature version A, p. 36
SPI	Serial Peripheral Interface, p. 36
SSRAM	Synchronous Static RAM, p. 35
SoC	System on Chip, p. 32
SoPC	System on Programmable Chip, p. 32
UART	Universal Asynchronous Receiver and Transmitter, p. 66
ULA	Unidade Lógica Aritmética, p. 57
VCO	Voltage-Controlled Oscillator, p. 54
VHDL	VHSIC Hardware Description Language, p. 31
VHSIC	Very High-Speed Integrated Circuit, p. 31

Capítulo 1

Introdução

Os navios, dos mais variados tipos e classificação, militares ou não, com diferentes recursos e inovações, são compostos por sensores, como radares, sonares e ecobatímetros, entre outros.

Os navios de guerra, especificamente, são compostos por armamentos, que diferem em tipos e categorias, conforme a aplicação. Cada armamento, por sua vez, pode possuir uma aplicação específica ou uma finalidade para a qual é mais eficiente.

Um armamento pode ser aplicado no combate a alvos aéreos, submarinos ou de superfície. Embora, alguns armamentos sejam empregados para combater em mais de um ambiente (aéreo, submarino ou superfície), eles costumam ser mais eficientes em um determinado ambiente.

Com o desenvolvimento tecnológico, radares e armamentos cada vez mais complexos, passou-se a utilizar computadores (inicialmente analógicos) para realizar o processamento dos dados oriundos dos sensores e transmiti-los para os sistemas de navegação ou táticos.

Na figura 1.1 é possível visualizar um exemplo com as interligações entre os grandes sistemas de um Sistema de Combate utilizado em alguns navios da Marinha do Brasil.

Os sistemas de interesse neste trabalho são:

a) Sistema Tático

   É o sistema responsável pela compilação do cenário tático, avaliação da situação tática e resposta tática, bem como pelo controle e coordenação dos equipamentos de Guerra Eletrônica.

   Este sistema compreende os Consoles Táticos responsáveis pelos cálculos e apresentação das compilações e avaliações táticas, os radares de busca aérea e de superfície, o sonar de casco e alguns outros sensores.

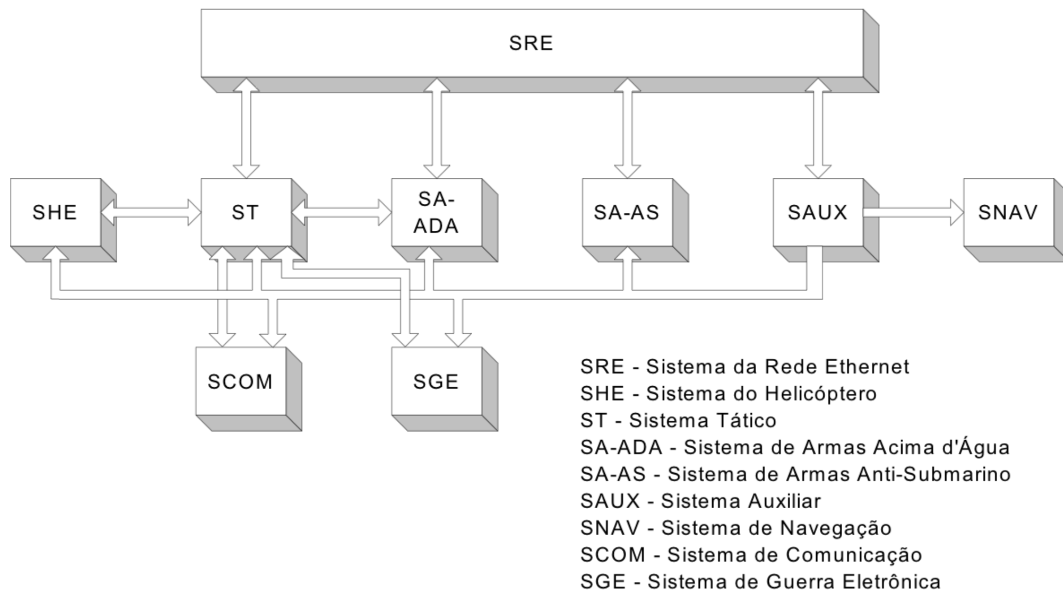


Figura 1.1: Exemplo de um Sistema de Combate

b) Sistema de Armas Acima d'Água

É o sistema responsável pelo engajamento com armas acima d'água.

Neste sistema são apresentados, principalmente, os Consoles de Controle de Armas Acima d'Água, os radares de Direção de Tiro, os canhões e os sistemas lançadores de mísseis MSA (Mísseis Superfície-Ar) e MSS (Mísseis Superfície-Superfície).

Na figura 1.2 é apresentada uma visão geral destes dois Sistemas integrando o Sistema de Combate tomado como exemplo.

Analisando as conexões apresentadas na figura, percebe-se mais facilmente que os sensores pertencem ao Sistema Tático que é o responsável por receber os dados dos sensores e processá-los. O sistema avalia esses dados e aqueles considerados ameaças, podem ser repassados (automaticamente ou por meio de intervenção humana) ao Sistema de Armas.

O Sistema de Armas, especificamente os Consoles de Controle de Armas, é responsável pela realização dos cálculos balísticos e a respectiva solução de tiro. Ou seja, os computadores deste Sistema são responsáveis em receber os dados de um determinado alvo, processar esses dados (realizar cálculos para estimar o ponto futuro do alvo, por exemplo), acionar os Radares de Direção de Tiro para acompanhar o alvo e direcionar o armamento para a posição correta. Em alguns modos de operação, esses computadores também são responsáveis por selecionar o melhor armamento para neutralizar a ameaça detectada.

A seguir, um exemplo mais objetivo deste trâmite de informações:

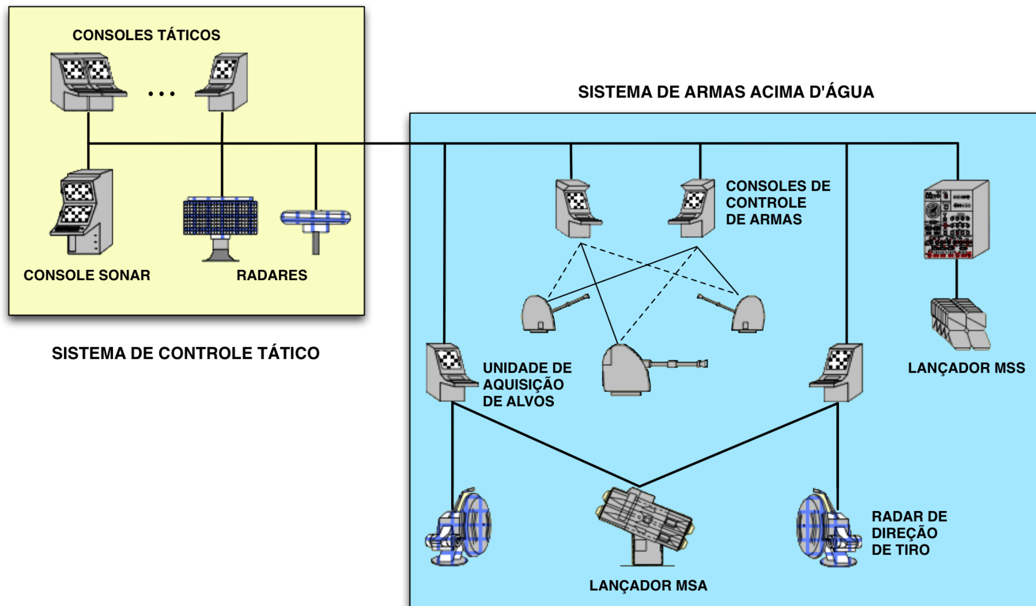


Figura 1.2: Interligação do Sistema de Combate

1. Um contato é repassado de um sensor (radar aéreo, por exemplo) para os Consoles Táticos;

2. O contato é avaliado como ameaça (por um operador humano ou automaticamente) e retransmitido ao Sistema de Armas;

3. O contato agora é designado alvo e, por meio do Console de Armas, o ponto futuro do alvo é calculado e um Radar de Direção de Tiro é designado para acompanhar este alvo;

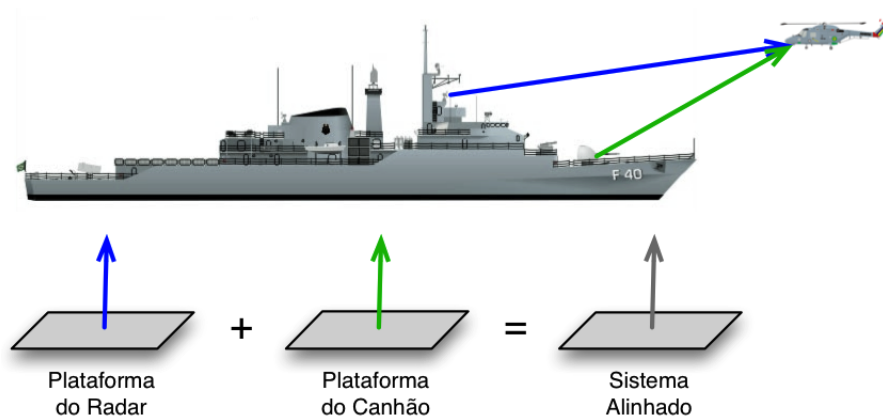
4. E por fim, um armamento é designado para combater a ameaça.

Visualizando este sistema como um sistema básico de controle onde genericamente existem sensores, computadores/calculadores e atuadores, talvez os problemas mecânicos presentes nestes sistemas de controle fiquem mais evidentes. Ou seja, problemas de referência e alinhamento como pode ser verificado na figura 1.3.

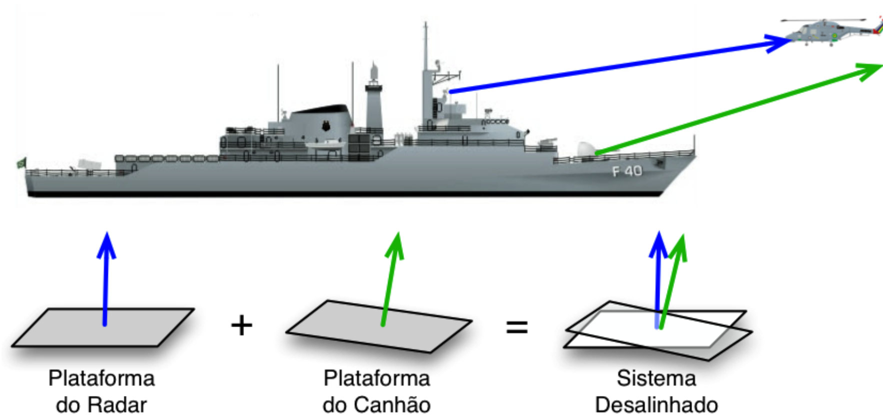
A figura 1.3b apresenta um desalinhamento da plataforma do canhão, desta maneira, o radar detecta o alvo na posição correta, o sistema de armas calcula corretamente o ponto futuro do alvo e comanda corretamente o canhão que, por estar desalinhado com o sistema, apontará para uma posição incorreta.

Ocorrendo o desalinhamento da plataforma do radar ao invés da plataforma do canhão conforme apresentado na figura, acarretará um resultado bastante semelhante.

Os sistemas de armas dos navios processam informações de elevação, marcação e distância de alvos para estabelecer soluções de tiro com o mesmo grau de precisão



(a) Sistema Alinhado



(b) Sistema Desalinhado

Figura 1.3: Alinhamento do Sistema

obtido pelos sensores. Apesar de parecer trivial, o processo de direcionamento preciso de uma arma para um alvo é um desafio para o sistema de armas que testa a capacidade de todas as suas funcionalidades.

Mesmo que todos os sensores e armas estivessem em um mesmo plano estabilizado, o processamento da solução de tiro seria um processo complexo devido às variáveis existentes como velocidades dos alvos, projéteis, intensidade dos ventos e temperatura. Como os sensores e armas estão posicionados em diversos conveses de um navio, é necessário conhecer a diferença angular entre estes planos para possibilitar a compensação matemática durante o processamento da solução de tiro.

Os processos de medição entre os planos e a motivação para o desenvolvimento de uma nova solução, objeto deste trabalho, são apresentados no Capítulo 2.

Capítulo 2

Tilt Test

Um ciclo completo de alinhamento de um navio de guerra normalmente totaliza 38 dias, sendo 9 dias em testes e rotina de alinhamento docado, 18 dias em testes e rotina de alinhamento no porto e 11 dias em testes e rotina de alinhamento no mar.

O propósito do alinhamento é determinar as inclinações entre os planos das bases dos elementos do sistema (radares, canhões, lançadores de mísseis, etc) e o plano de referência. Esse alinhamento também inclui a verificação do sistema giroscópico do navio (teste de verticalidade da referência). Os valores de inclinação, sendo superiores aos valores de tolerância, serão introduzidos como correções para obter o paralelismo desejado. Esses valores de tolerância podem ser da ordem 6' para radares e canhões e alguns graus para os lançadores de mísseis[1].

O alinhamento realizado na primeira fase do ciclo de alinhamento (fase de docagem) é conhecido como *Tilt Test*, expressão inglesa que pode ser traduzida como *Teste de Alinhamento* e que teve o seu uso conservado por ser comum no meio naval mundial e difundido aqui no país durante o recebimento das primeiras Fragatas Classe Niterói compradas da Inglaterra na década de 70 [1].

A necessidade de verificações periódicas das condições de paralelismo entre os planos das bases dos elementos do sistema decorre da própria vida operativa do navio onde pequenas deformações, desgastes e avarias cooperam para alterar as condições desejáveis de paralelismo e desta forma passam a comprometer a eficácia do sistema.

Os principais conceitos envolvidos no alinhamento das bases dos elementos são os seguintes:

- Inclinação de um Elemento

  O conceito de inclinação puro e simples não existe se não existir um referencial. Assim, para ser possível afirmar que um plano está inclinado é necessário

[1] O lançamento de um míssil MSS, por exemplo o míssil de fabricação francesa *EXOCET*, por possuir um radar interno para aquisição do alvo após uma certa distância percorrida, não exige um alinhamento rigoroso pois o próprio radar interno do míssil corrigirá a sua trajetória até o alvo.

determinar um plano de referência. O ângulo entre esses dois planos determina a inclinação.

- Plano de Referência

  O plano de referência em relação ao qual são determinados os ângulos de inclinação das bases dos elementos do Sistema de Armas chama-se *Master Level* (figura 2.1).

  O *Master Level* define um plano paralelo à linha d'água de projeto do navio sendo materializado por uma placa de aço perfeitamente usinada, localizada no interior do compartimento das agulhas giroscópicas.

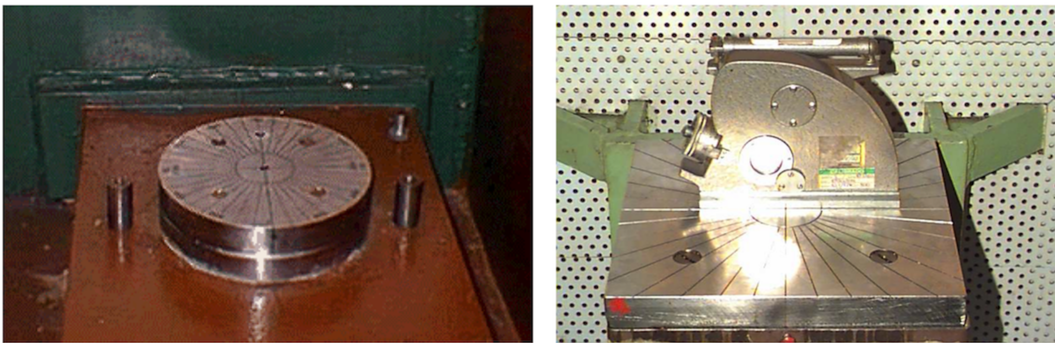


Figura 2.1: Exemplos de *Master Level*

- Inclinação relativa entre dois planos

  A inclinação entre o Plano de Passeio de Roletes (PPR) de um elemento qualquer e o plano de referência é expressa pelo valor da inclinação e da conteira no ponto onde a máxima inclinação ocorre. Este ponto é conhecido como *Ponto Alto*.

  Assim, desde que seja estabelecida uma linha de referência para a origem das medidas de marcação, a inclinação de um plano fica bem determinada pelo conhecimento da conteira do ponto alto e o valor de sua inclinação.

A figura 2.2 apresenta dois planos que se interceptam segundo a linha AB, um representando o plano de referência e o outro o plano de passeio de roletes (PPR) de um elemento qualquer.

É possível observar também que o ângulo assume seu valor máximo (α) na direção **OF** e o valor de $0°$ na direção **OB**. Na direção **OE** assume o valor de -α e na direção **OA** assume novamente o valor de $0°$, ou seja, pode-se dizer que sua variação é senoidal [2].

Desse modo, a inclinação entre os dois planos é representada pelo ângulo α e a conteira onde esta inclinação ocorre, pelo ângulo β.

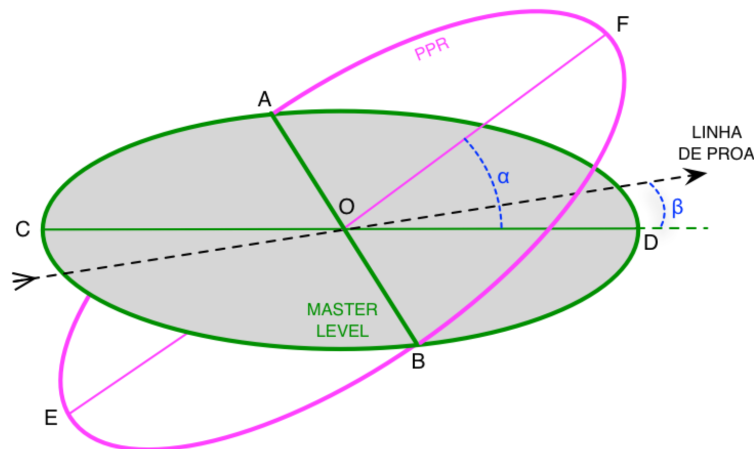


Figura 2.2: Inclinação relativa entre dois planos

2.1 Representação Gráfica da Inclinação entre Planos

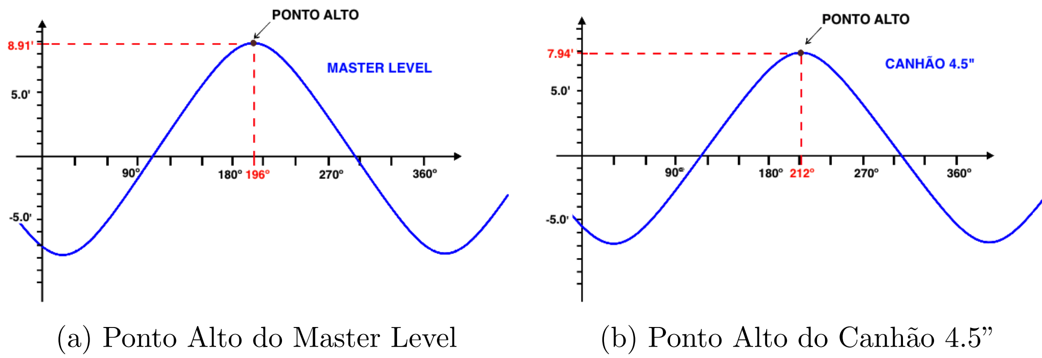
Realizando as medidas de 10 em 10 graus para obter os valores α e β até completar 360° e plotando estes valores em eixos cartesianos, onde as ordenadas representam as inclinações e as abscissas os valores de conteira, pode-se reproduzir curvas senoidais [3] de onde será possível retirar a conteira do ponto alto e a inclinação de cada plano em relação ao plano horizontal.

Obtendo uma curva perfeitamente senoidal indica um plano de passeio de roletes perfeitamente plano. Entretanto, se os pontos plotados divergirem de uma curva senoidal perfeita, há a indicação de uma leitura incorreta ou um plano de passeio de roletes com problema de planicidade.

A figura 2.3 apresenta um exemplo de medidas realizadas em um *Master Level* e do plano de passeio de roletes de um canhão 4.5” em relação ao horizonte verdadeiro.

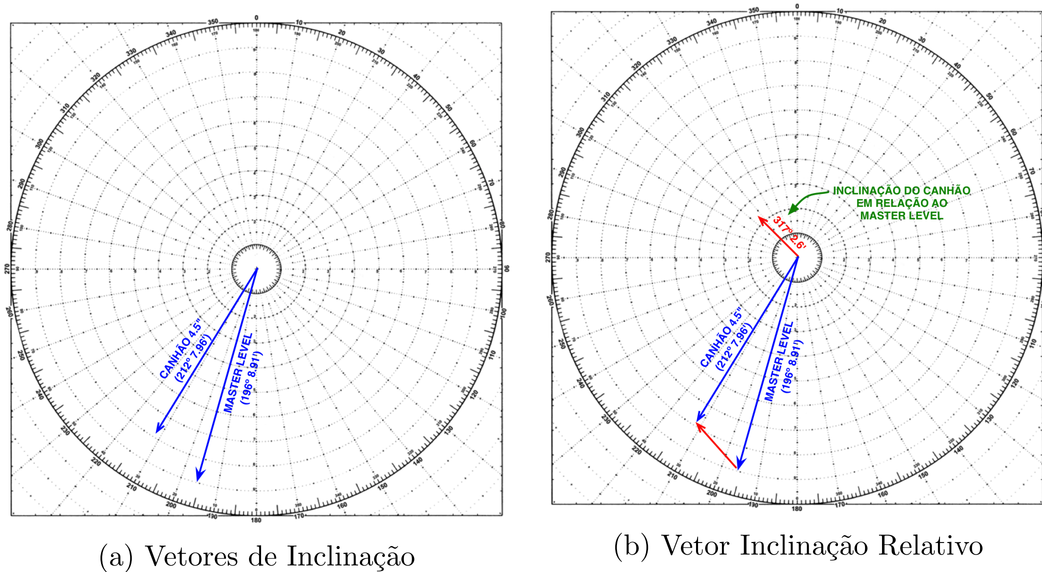
Estas inclinações podem ser convenientemente plotadas em um diagrama vetorial como apresentado na figura 2.4. A escala a ser escolhida para a plotagem na rosa de manobra deve ser a maior possível para maior precisão.

A inclinação relativa entre os planos é obtida pela combinação dos dois vetores, observando que o “vetor inclinação” deve ter a sua origem na extremidade do vetor de referência e o seu outro ponto na extremidade do vetor do elemento. O vetor inclinação relativo apresentado na figura 2.4b representa a inclinação do plano de passeio de roletes do Canhão 4.5” em relação ao *Master Level* que, neste exemplo, é de 2,6’ ao 317° relativos.



(a) Ponto Alto do Master Level (b) Ponto Alto do Canhão 4.5”

Figura 2.3: Exemplo Ponto Alto



(a) Vetores de Inclinação (b) Vetor Inclinação Relativo

Figura 2.4: Diagrama Vetorial

2.2 Tilt Test com Clinômetros de Bolha

Os elevados requisitos de precisão exigidos pelo Sistema de Armas e, principalmente, pela sensibilidade dos instrumentos de medição empregados neste processo, exigem condições especiais para a condução do *Tilt Test*. Portanto, convencionou-se chamar de “condições de *Tilt Test*” as condições especiais às quais o navio deve se submeter de modo a permitir a condução dos alinhamentos dentro dos requisitos de precisão exigidos.

Dentre as “condições de *Tilt Test*”, destacam-se algumas, como:

- Navio docado e flutuando com cerca de 60 cm de água entre o domo do sonar e o fundo do dique para reduzir a influência do vento;

- Navio escorado no dique com valor máximo de banda de 30’. Como o valor recomendado é de 5’, pequenas correções na inclinação poderão ser feitas com a transferência de óleo ou água entre tanques ou com o aperto das escoras. O

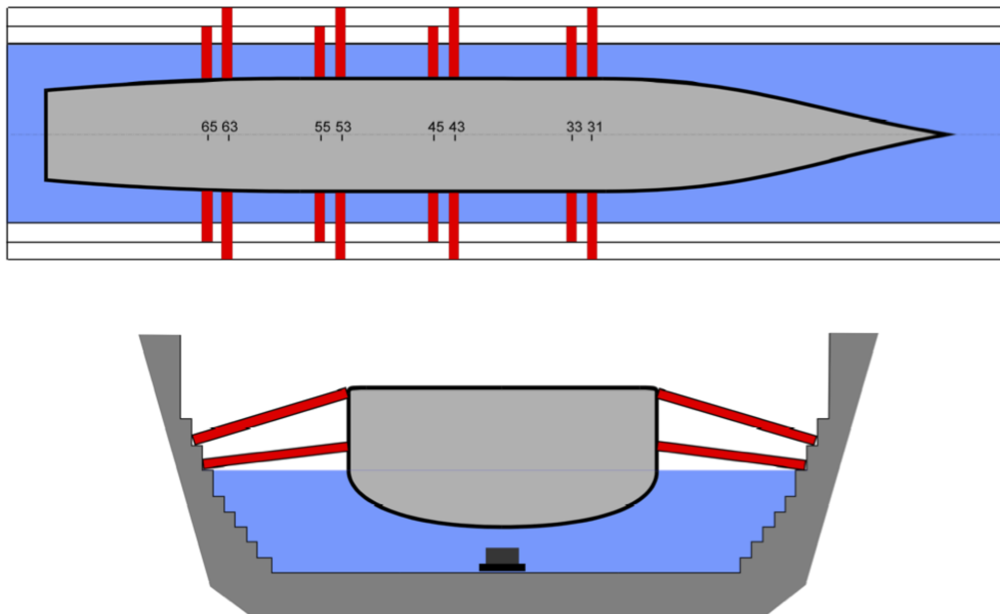


Figura 2.5: Escoramento Recomendado para a Condição de *Tilt Test*

escoramento tem por finalidade manter a verticalidade do navio e consiste ao todo de 16 escoras, cuja disposição pode ser visualizada na figura 2.5;

- Movimentação de pesos e o trânsito a bordo devem ser proibidos;
- Máquinas cujo funcionamento causem trepidações ou oscilações na estrutura do navio devem ser desligadas, mantendo alimentados apenas os equipamentos indispensáveis (motores e bombas da rede de incêndio, por exemplo); e
- As leituras deve ser executadas apenas nos períodos noturnos de modo a evitar as distorções térmicas da estrutura do navio. Estas distorções sempre ocorrem e são especialmente significativas quando o navio, no porto, fica submetido às altas temperaturas diurnas.

É importante frisar que qualquer vibração ou oscilação do navio pode ser sentida nos clinômetros, o que pode causar erros de leitura e a consequente invalidação dos resultados.

Essas atividades de teste de paralelismo consistem em realizar todas as medidas das inclinações das bases das instalações dos sistemas de armas e sensores em relação ao plano de referência (*Master Level*) e são realizadas, manualmente, utilizando um Clinômetro de Bolha. A partir dessas medidas são calculados os valores de correção e, a posteriori, poderão ser aplicados diretamente nas próprias instalações ou introduzidos nos softwares de controle dos Sistemas Táticos e de Armas de cada navio.

O Clinômetro é um instrumento utilizado para medir a inclinação de uma superfície plana em relação ao horizonte. Consiste basicamente de um nível de bolha

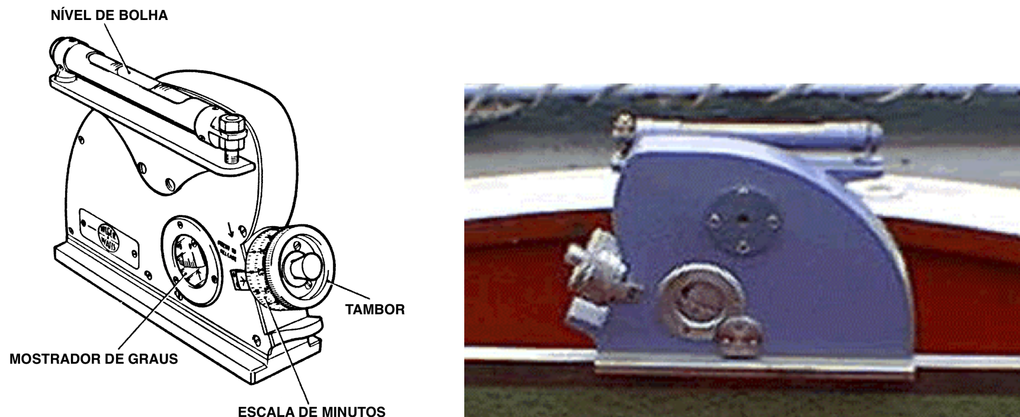


Figura 2.6: Clinômetro de Bolha

associado a um tambor de ajuste e uma escala graduada (figura 2.6).

No caso do *Tilt Test*, as superfícies a terem sua inclinação medida são as bases ou os planos de passeio de roletes (PPR) das instalações de armamento.

A tomada de leitura no clinômetro é feita diretamente de suas escalas. Inicialmente, posiciona-se o clinômetro na superfície onde se deseja medir a inclinação. A seguir, girando convenientemente o tambor busca-se colocar a bolha ao centro estabilizando-a nesta posição.

De um modo geral, nas bases das instalações de armamento há um local próprio para se posicionar o clinômetro, o que é importante, pois evita a introdução de erros devido a rugosidades ou deformações da superfície onde serão colocados os instrumentos. A figura 2.7 apresenta exemplos do posicionamento de clinômetros nas bases de alguns elementos.

Os clinômetros utilizados (Higer & Watts) possuem uma escala dividida em graus variando de 0° a 90° e um tambor em 120 divisões de 0.5'. Os valores inteiros correspondem aos minutos e os pontos entre divisões indicam 0,5' ou 30".

A execução propriamente dita do teste consiste em verificar, simultaneamente, a inclinação das bases das instalações de armamento, através dos diversos clinômetros instalados nos locais apropriados nos elementos envolvidos.

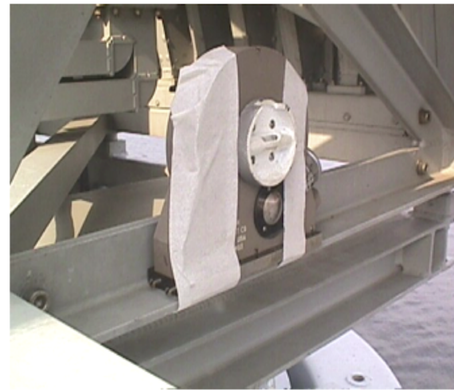
A estação controladora fica no Compartimento das Giros (Agulhas Giroscópicas) junto à estação do *Master Level* e, todas as outras a ela são interligadas por comunicações interiores do próprio navio. Quando a bolha do clinômetro do *Master Level* estiver na posição correta, o seu operador informa ao controlador do teste que, por sua vez, aciona as demais estações com um sinal de "Top". Assim, o operador de cada estação posicionará a bolha de seus clinômetros ao centro utilizando o tambor e informará a medida do ângulo lido.

Após o registro dos valores lidos, em décimos de minutos, a conteira de todos os elementos deve ser alterada de 10° no sentido horário. Todos os canhões e sensores devem ser conteirados manualmente. O processo se repete até completar os 360°. E,

(a) Canhão de 4.5”

(b) Radar de Direção de Tiro

(c) Radar de Busca



(d) Radar de Busca

Figura 2.7: Posicionamento dos Clinômetros

concluindo a leitura das inclinações de todos os elementos de 10 em 10 graus pode-se proceder ao cálculo dos vetores de inclinação de cada elemento em relação ao seu plano de referência. Esta inclinação relativa pode ser obtida através de composição vetorial como apresentado na seção 2.1.

Com base nos requisitos rígidos e nos procedimentos apresentados nesta seção para a medição das inclinações de todas as plataformas de um navio, não fica difícil imaginar a motivação para o desenvolvimento de uma solução moderna e mais automatizada para a obtenção e cálculo dos elementos necessários para a realização do *Tilt Test*. Foi, então, desenvolvida uma solução piloto apresentada na seção a seguir.

2.3 Tilt Test Eletrônico

O projeto *Tilt Test Eletrônico* foi concebido com a finalidade de atender alguns objetivos:

- Aumentar a fidelidade do *Tilt Test* diminuindo o tempo entre a medição e o exercício de tiro;
- Elevar a precisão e exatidão das medidas de *tilt* mantendo o conceito legado de alinhamento;

- Eliminar a fase de docagem dos navios para esta atividade; e
- Elevar o nível de automação no processo de aquisição das medidas de *tilt*.

O procedimento de realização do *Tilt Test* "convencional" agrega a experiência adquirida ao longo dos últimos 20 anos com o histórico evolutivo das Avaliações Operacionais das Fragatas Classe Niterói. Durante esse período, as diversas fases do procedimento têm sido observadas continuamente. Desta forma, foram identificados os pontos "fortes" do processo e aqueles que poderiam ser melhorados para elevar a qualidade dos resultados e a redução dos custos. Dessa análise resultaram as seguintes observações:

- Pontos "fortes"
  - Método usado no processamento matemático; e
  - Apresentação dos resultado.
- Pontos a serem melhorados
  - Qualidade estatística dos dados observados;
  - Minimizar a intervenção humana na observação das medidas;
  - Intervenção humana nas anotações intermediárias; e
  - Sincronismo das medições.

O Centro de Apoio a Sistemas Operativos (CASOP), organização militar da Marinha do Brasil, elaborou um projeto piloto com os objetivos de automatizar o processo e verificar a viabilidade de um sistema completamente automático de alinhamento. Tal projeto englobou tanto a coleta automatizada de dados dos sensores (monoaxiais), processamento e filtragem de ruídos, cálculo do desalinhamento das bases das armas e sensores e da verticalidade dos sistemas giroscópicos nos navios da Marinha do Brasil.

Foram considerados os seguintes aspectos quanto à condução do projeto piloto:

- Estudo de viabilidade técnica e econômica a partir de um protótipo piloto;
- Levantamento e análise dos requisitos para um futuro projeto final; e
- Modelagem, implementação e testes de validação do protótipo.

Considerando que se tratava de um projeto piloto, o desenvolvimento do projeto de alinhamento teve as seguintes premissas atendidas:

- Utilização de uma solução mono-canal, ou seja, utilizando sensores inclinômetros de um eixo;
- Utilização de material para o desenvolvimento conforme o conceito COTS (*Commercial-off-the-shelf*), amplamente adotado nas últimas décadas pela Marinha do Brasil;
- Seleção de todo o material envolvido visando custo compatível a um projeto piloto, mas que permitisse flexibilidade na implementação; e
- Custo do sistema final após produção inferior aos sistemas importados, que variam na faixa de R$ 600.000,00 [4].

Logo, optou-se para aquisição de um sensor inclinômetro (figura 2.8) de apenas um eixo, com comunicação serial RS-232 [5], por atender as premissas e, como no caso trata-se de um projeto piloto, também por estarem intimamente relacionados ao baixo custo e a precisão abaixo do definido nos requisitos de alinhamento.

Figura 2.8: Inclinômetro modelo T7

A primeira dificuldade a ser superada foi o tratamento dos dados lidos dos sensores de inclinação. Ao contrário do Clinômetro de Bolha onde é realizada apenas uma leitura para cada marcação (ao sinal de "Top"), o sensor é lido milhares de vezes para uma mesma marcação e como pode ser visualizado na figura 2.9, os valores oscilam consideravelmente ao longo do tempo.

Diversas análises foram realizadas sobre o sensor para verificar a precisão e exatidão, considerando o comportamento, estabilidade e velocidade das medições nos mais variados contextos.

Os sinais observados demonstraram comportamento semelhante em ensaio dinâmico, apesar da clara necessidade de tratamento digital e aplicação de filtro. Esta constatação positiva aliada à grande quantidade de amostras obtidas em um pequeno intervalo de tempo, motivaram as análises estatísticas de densidade das medições e variância (ou desvio) para a verificação da precisão e exatidão.

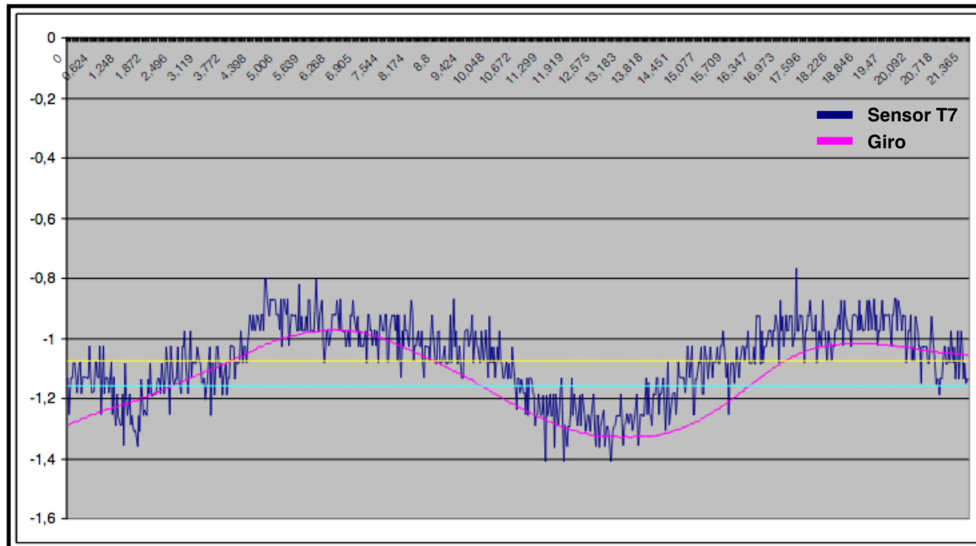


Figura 2.9: Dados Brutos do Sensor T7

Durante análises laboratoriais, em condição estática, foram realizadas comparações entre as medidas dos sensores com os sinais de uma agulha giroscópica. Estes sinais foram aproveitados para verificar a precisão utilizando uma análise estatística.

Foram observadas 17983 medições angulares das quais 17233 atenderam os requisitos de precisão, ou seja, 95.8% das leituras são consideradas precisas para as necessidades de *Tilt Test*. As leituras excedentes apresentaram desvio padrão de 15,70'. O resultado desta análise é apresentado na figura 2.10.

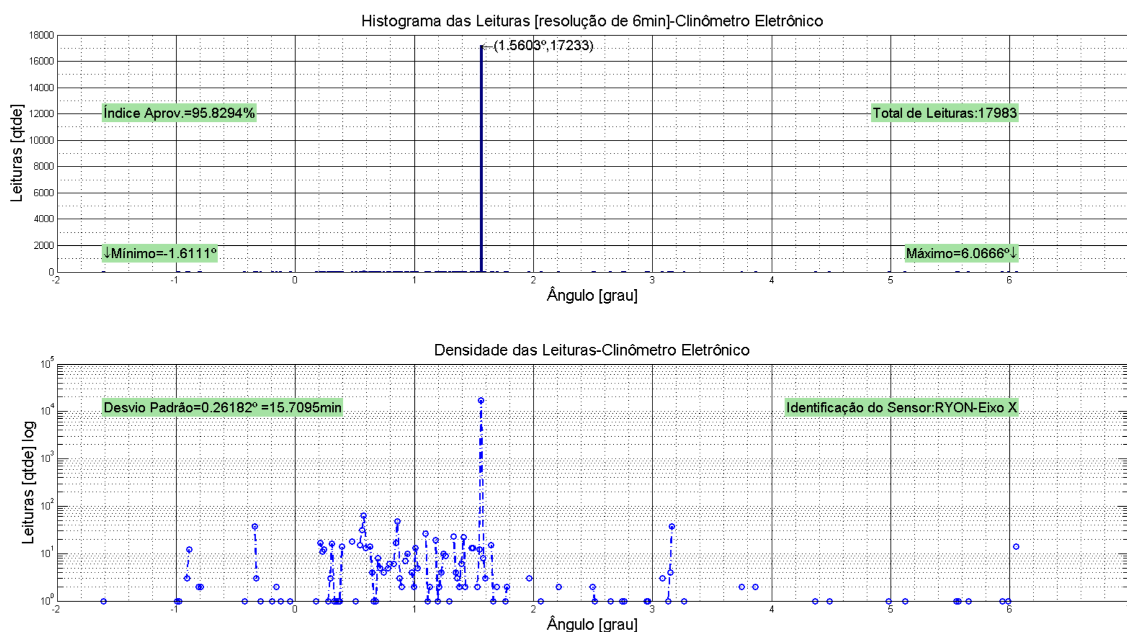


Figura 2.10: Histograma (densidade) das Medições

2.3.1 Filtro Digital

O desenvolvimento do filtro digital iniciou após a análise das leituras dos sensores. Desta forma, o escopo da implementação do filtro concentrou-se em manter os fatores de qualidade (precisão e exatidão) e melhorar os fatores relacionados à variância (dispersão, desvio, medida máxima e medida mínima).

Os estudos foram iniciados pelos filtros recursivos. Foram realizados testes com os filtros de *Kalman*, *Alfa* e *Beta*. Em todos os casos foram observados comportamentos semelhantes ao exemplificado na figura 2.11 (em vermelho).

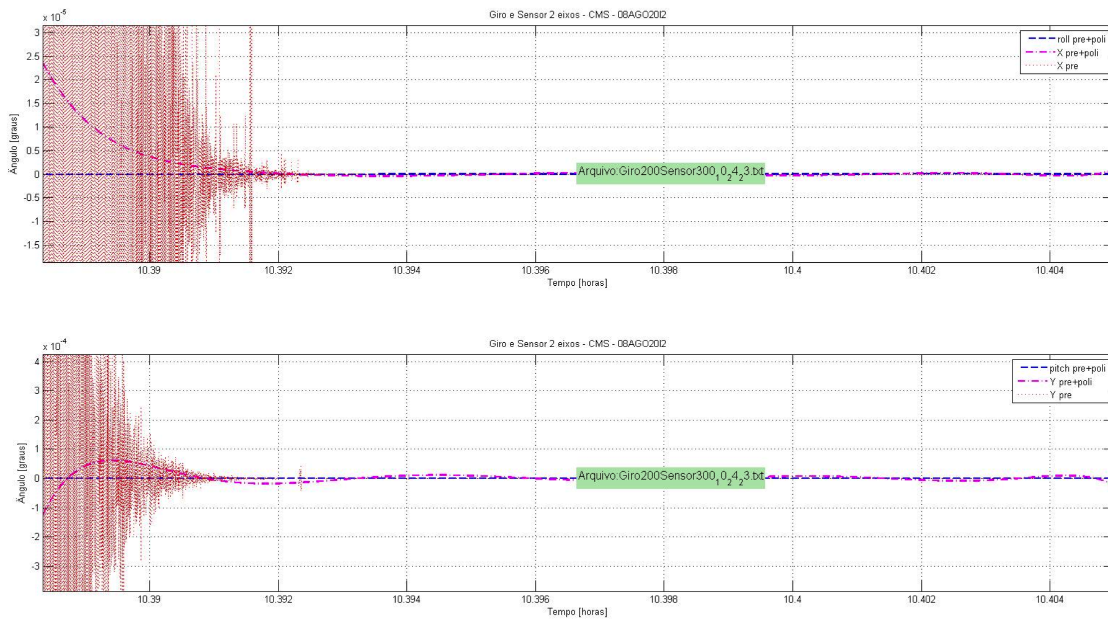


Figura 2.11: Aplicação de Filtros Recursivos

Apesar da característica destes filtros em reduzir a variância (matriz covariância) do sinal pela aplicação de estimadores, foram constatados alguns efeitos indesejados:

- Considerável consumo de amostras (medições) para o funcionamento do filtro;
- Alta frequência residual, mesmo após a estabilização do filtro; e
- Variância (matriz covariância) remanescente.

A solução foi alcançada com a constatação de que os sinais dos sensores não necessitavam de filtragem "dinâmica" e poderia ser realizada após a leitura da última medição. Desta forma, foi possível a aplicação de *Aproximação Linear* na curva obtida.

$$Y = \sum_{i=0}^{8} a_i X^i = a_8 X^8 + a_7 X^7 + a_6 X^6 + a_5 X^5 + a_4 X^4 + a_3 X^3 + a_2 X^2 + a_1 X + a_0$$

O grau oito foi atribuído ao filtro após estudos empíricos das condições de variação angular do navio, sendo este grau que apresentou os melhores resultados. Os coeficientes são calculados a partir das medições.

Desta forma, conseguiu-se uma filtragem ótima dos sinais (figura 2.12), ou seja:

- Sem perdas de amostras para o funcionamento do filtro;
- Sem necessidade de estabilização;
- A frequência de balanço do navio é a única que compõe o sinal;
- Utilização simultânea das medições aproveitáveis e dispersas; e
- Efeito de variância nula em torno do sinal filtrado.

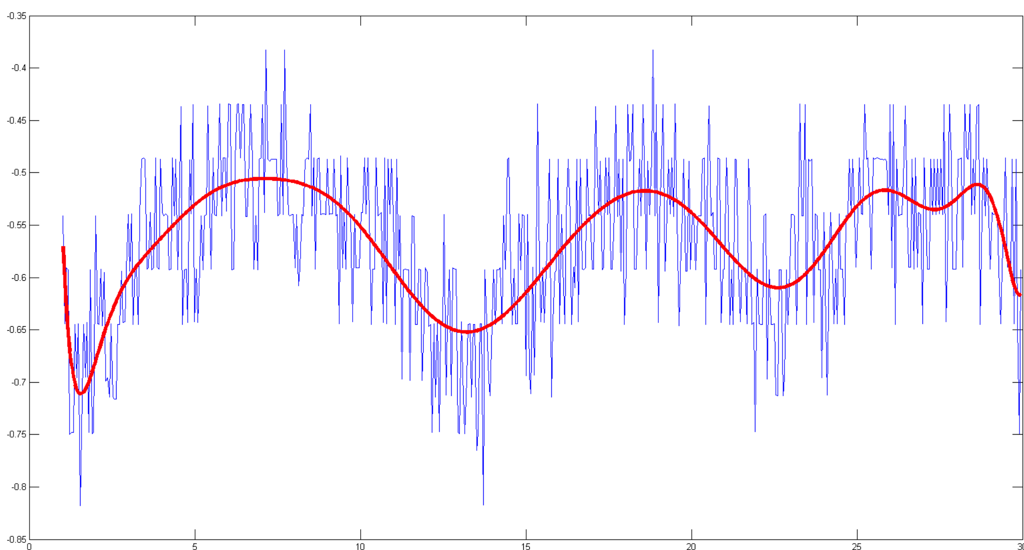


Figura 2.12: Aplicação de Filtro Polinomial

Para verificar a exatidão dos dados, foram realizadas comparações entre as leituras dos sensores com os sinais obtidos de uma agulha giroscópica utilizada como referência. A figura 2.13 apresenta um exemplo de comparação.

Os sensores foram posicionados de forma a alinhar os seus eixos internos (chamado de X em um sensor e de Y no outro) com o *pitch* e *roll* (balanço e caturro) gerado pela agulha giroscópica. Desta forma, é possível concluir que a qualidade das leituras dos sensores eletrônicos, após a aplicação de filtro, se equipara com a qualidade fornecida pela *giro*.

2.3.2 Desenvolvimento Matemático

Nas seções anteriores, foi apresentado que através das leituras realizadas nos vários elementos, de 10 em 10 graus, é obtido um conjunto de dados que, quando plotados

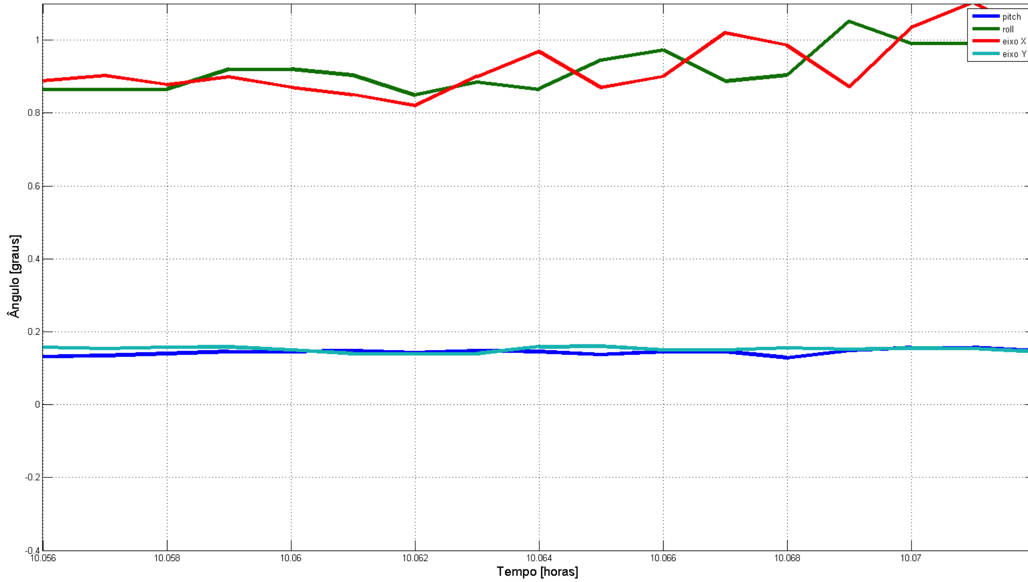


Figura 2.13: Comparação entre Sensor e Giro

no plano cartesiano, produzem um *Diagrama de Dispersão*. Este diagrama é a representação gráfica dos valores das leituras feitas no espaço cartesiano XY e os pontos são representativos de uma curva de aparência senoidal.

Do diagrama de dispersão, obtém-se um relação funcional $y = f(x)$ desconhecida e com uma forma sugerida pelo gráfico dos pontos que se assemelha à uma senoide, então, o problema consiste em determinar um curva $y = g(x)$ que melhor se ajusta ao conjunto de pontos observado. Nestas condições, a função $g(x)$ corresponde a uma aproximação da relação funcional desconhecida $y = f(x)$.

O modelo mais simples para relacionar duas variáveis x e y é uma equação da reta, $Y = a + bX$, onde a e b são os parâmetros do modelo. Para cada ponto X_n, corresponde um ponto $\hat{Y}_n$ de valor $(a + bX_n)$. Os dados no sistema de coordenadas (X,Y) podem ser representados pelos pontos (X_1, Y_1), (X_2, Y_2), ..., (X_N, Y_N) e para um dado valor X_1, haverá uma diferença entre Y_1 e o valor correspondente determinado na curva C. Esta diferença é representada por D_1, denominada como *desvio*, *erro* ou *resíduo*, que corresponde à distância vertical do ponto ate é reta. Da mesma maneira, são obtidos os desvios D_2, ..., D_N. Ou seja:

$$D_1 = Y_1 - \hat{Y}_1 \quad \therefore \quad D_1 = Y_1 - (a + bX_1)$$

$$D_2 = Y_2 - \hat{Y}_2 \quad \therefore \quad D_2 = Y_2 - (a + bX_2)$$

Então, a forma geral pode ser definida como:

$$D_n = Y_n - \hat{Y}_n \quad \therefore \quad D_n = Y_n - (a + bX_n)$$

Considerando a soma dos quadrados dos desvios, obtém-se o desvio total dos pontos observados à reta estimada.

$$D(a,b) = \sum_{i=1}^{n}(Y_1 - \hat{Y}_1)^2$$

$$D = \sum(Y_1 - (a + bX_1))^2$$

A solução consiste em encontrar os valores de a e b que minimizem a função quadrática $D(a,b)$ utilizando o *Método dos Mínimos Quadrados*.

A reta que melhor se ajusta aos dados do diagrama de dispersão pode ser representada como $y = a + bx$.

A medida de qualidade do ajustamento da curva C aos dados apresentados, chamada *aderência*, é proporcionada pela quantidade ${D_1}^2 + {D_2}^2 + ... + {D_N}^2$ que, quanto menor, melhor a qualidade do ajustamento. Portanto, de todas as curvas que se ajustam a um conjunto de pontos, a que apresenta o menor valor do somatório acima, é denominada a melhor curva de ajustamento.

A reta de mínimo quadrado que se ajusta a um conjunto de pontos pode ser definida pela equação:

$$\hat{Y} = a_0 + a_1X$$

A relação entre uma variável dependente y com $n + 1$ variáveis independentes pode ser expressa por um modelo linear, na forma:

$$Y = a_0 + a_1X_1 + a_2X_2 + ... + a_nX_n$$

Para um ajuste linear múltiplo com três variáveis, a equação é apresentada como:

$$\hat{Y} = a_0 + a_1X_1 + a_2X_2$$

Conforme apresentado anteriormente, a representação dos pontos no diagrama de dispersão apresenta uma forma senoidal. Portanto, utilizando a forma geral da senoide:

$$Y = A.sen(\alpha + \theta) + d$$

Onde:

A = amplitude

α = deslocamento horizontal angular

θ = deslocamento vertical da curva da senoide

d = deslocamento médio da curva no eixo Y

Aplicando uma regra geral da trigonometria, $sen(a+b) = sena.cosb+senb.cosa$, obtém-se:

$$Y = A.(sen\alpha.cos\theta + sen\theta.cos\alpha) + d$$

$$Y = A.cos\theta.sen\alpha + A.sen\theta.cos\alpha + d$$

Retornando à equação de ajuste múltiplo $\hat{Y} = a_0 + a_1X_1 + a_2X_2$ e realizando a correspondência com a equação trigonométrica acima, é possível associar:

$a_0 = d$
$a_1 = A.cos\theta$
$a_2 = A.sen\theta$
$X_1 = sen\alpha$
$X_2 = cos\alpha$

As constantes a_0, a_1 e a_2 podem ser determinadas com a resolução de um sistema linear de equações.

$$\sum Y = a_0N + a_1\sum X_1 + a_2\sum X_2$$

$$\sum X_1Y = a_0\sum X_1 + a_1\sum {X_1}^2 + a_2\sum X_1X_2$$
(multiplicado por $\sum X_1$)

$$\sum X_2Y = a_0\sum X_2 + a_1\sum X_1X_2 + a_2\sum {X_2}^2$$
(multiplicado por $\sum X_2$)

Onde N corresponde ao número de amostras obtidas para cada elemento, no caso de uma leitura de 10 em 10 graus, num arco de 360°, $N = 36$ amostras.

Os coeficientes a_0, a_1 e a_2 podem ser determinados pela *Regra de Cramer* para a solução de um sistema de equações lineares com n equações e n incógnitas.

O primeiro determinante é calculado para obter o *divisor*.

$$div = \begin{vmatrix} N & \sum X_1 & \sum X_2 \\ \sum X_1 & \sum {X_1}^2 & \sum X_1X_2 \\ \sum X_2 & \sum X_1X_2 & \sum {X_2}^2 \end{vmatrix}$$

Os coeficientes a_0, a_1 e a_2 podem ser calculados pela substituição das respectivas colunas e divisão por *div*.

$$a_0 = \begin{vmatrix} N & \sum Y & \sum X_2 \\ \sum X_1 & \sum X_1Y & \sum X_1X_2 \\ \sum X_2 & \sum X_2Y & \sum {X_2}^2 \end{vmatrix} / div$$

$$a_1 = \begin{vmatrix} N & \sum X_1 & \sum Y \\ \sum X_1 & \sum {X_1}^2 & \sum X_1Y \\ \sum X_2 & \sum X_1X_2 & \sum X_2Y \end{vmatrix} / div$$

$$a_2 = \begin{vmatrix} \sum Y & \sum X_1 & \sum X_2 \\ \sum X_1 Y & \sum {X_1}^2 & \sum X_1 X_2 \\ \sum X_2 Y & \sum X_1 X_2 & \sum {X_2}^2 \end{vmatrix} / div$$

A amplitude da curva pode ser expressa em minutos e sua direção em graus. Estes valores pode ser representados no plano cartesiano (X, Y) e, como apresentado anteriormente, $a_1 = A.cos\theta$ e $a_2 = A.sen\theta$, que correspondem às projeções do vetor de amplitude nos eixos X e Y (figura 2.14).

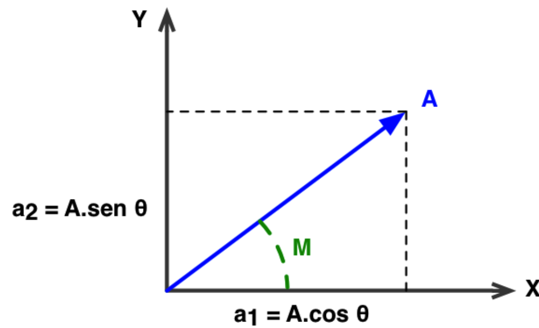


Figura 2.14: Representação da Amplitude

Recorrendo ao *Teorema de Pitágoras*:

$$A^2 = {a_1}^2 + {a_2}^2$$

$$\boxed{A = \sqrt{{a_1}^2 + {a_2}^2}}$$

O valor da marcação onde ocorre a amplitude da curva é obtido pela função *arco tangente*.

$$\boxed{M = arctg(\frac{a_2}{a_1})}$$

2.3.3 Modelagem

A modelagem foi dividida em duas partes: matemática e de software. Na modelagem matemática, foram estudadas funções e filtros para tratamento dos dados adquiridos pelos sensores conforme apresentado nas seções anteriores. Na modelagem do software, foram construídos modelos para implementação do aplicativo de interação com os sensores e o banco de dados contendo algumas informações básicas dos navios, sensores e armas.

A implementação dos modelos de software foi realizada na linguagem de programação Java e a concepção dos modelos matemáticos foi realizada no software Matlab (figura 2.15) que possui um componente chamado *Matlab Builder JA* para

compilar e gerar Classes Java. Então, é possível instanciar classes do Matlab e executar os métodos necessários, integrando, desta forma, o software escrito em Java com o Matlab.

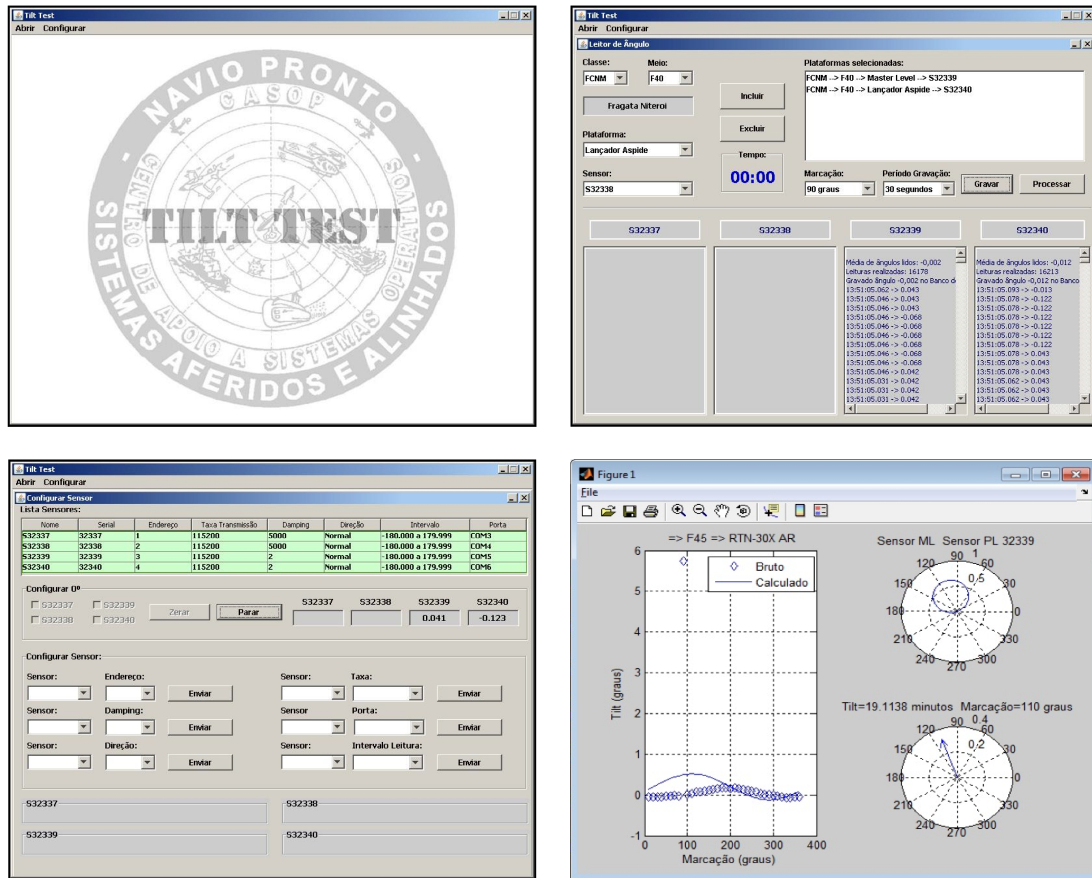


Figura 2.15: Janelas do *Tilt Test* Eletrônico

O aplicativo desenvolvido em Java é responsável também pela filtragem inicial dos dados capturados pelos sensores, denominada filtragem de sincronização. Este processo é conduzido da seguinte forma:

1. O software instancia uma *thread* para cada sensor e realiza a leitura durante um tempo pré-determinado. Cada *thread* possui uma estrutura de dados (lista encadeada) para armazenar os ângulos com seus respectivos tempos de leitura.

2. Ao término da fase de leitura, as medições são comparadas pelo registro de tempo e, aquelas com diferença acima do limite estabelecido, normalmente 1 ms, são descartadas.

3. A lista de ângulos contendo os tempos com a precisão desejada é transferida para processamento no Matlab.

4. Um método de filtragem é aplicado para permitir o cálculo da diferença angular

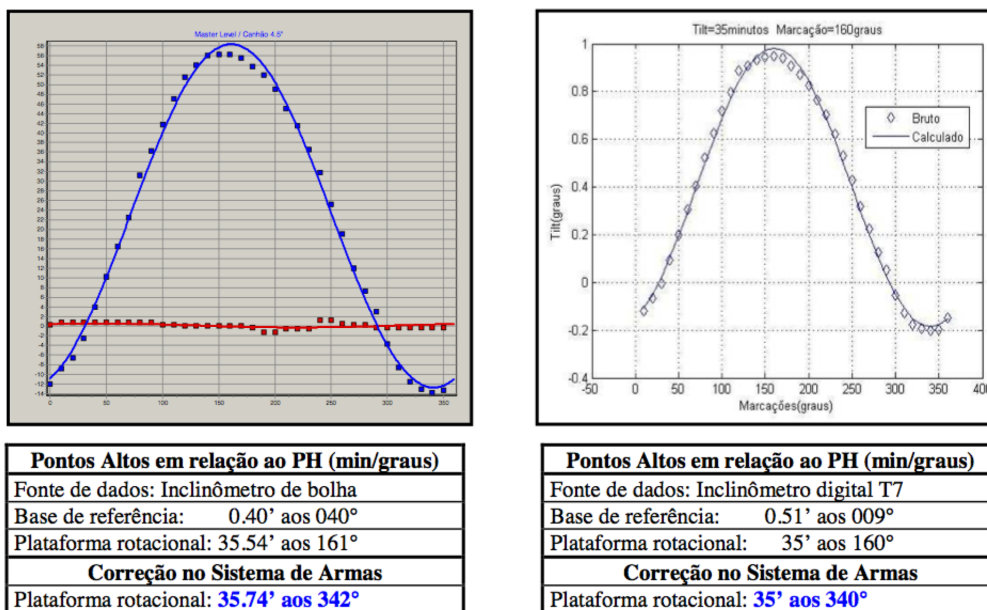
entre o *Master Level* e cada um dos sensores. Este ângulo é retornado ao software em Java.

5. O valor retornado representa o ângulo entre a plataforma do sensor e o *Master Level* em uma determinada marcação. Este dado é armazenado e o processo é repetido até que sejam obtidas as medidas das 36 marcações (executadas de 10 em 10 graus).

6. Com as 36 marcações, é possível gerar a curva resultante, então, o conjunto de dados é transferido novamente ao Matlab para plotar a curva senoidal, exibir o gráfico e identificar o valor e a direção do ponto alto.

2.3.4 Conclusão do Tilt Test Eletrônico

Os testes de validação foram realizados por meio da comparação dos resultados da execução dos métodos atualmente em uso e o automatizado (figura 2.16), a bordo de uma Fragata, por ocasião da fase docado (9 dias) do seu ciclo de alinhamento.



Pontos Altos em relação ao PH (min/graus)
Fonte de dados: Inclinômetro de bolha
Base de referência: 0.40' aos 040°
Plataforma rotacional: 35.54' aos 161°
Correção no Sistema de Armas
Plataforma rotacional: **35.74' aos 342°**

Pontos Altos em relação ao PH (min/graus)
Fonte de dados: Inclinômetro digital T7
Base de referência: 0.51' aos 009°
Plataforma rotacional: 35' aos 160°
Correção no Sistema de Armas
Plataforma rotacional: **35' aos 340°**

Figura 2.16: Comparação do Método *Tradicional* x *Automatizado*

Com a realização de diversos testes comparativos [4], verificou-se que com o tratamento dos dados coletados no método automatizado aproximou-se dos resultados do processo convencional: diferença inferior a 19 décimos de minuto de arco, com diferença de orientação inferior a 6°. Considerando o fato do projeto piloto utilizado não possuir a precisão adequada em consequência do inclinômetro utilizado, os resultados foram altamente motivadores, corroborando a viabilidade de desenvolvimento de um sistema robusto, com mais canais, utilizando sensores precisos de dois eixos e confiável, que automatize a metodologia atual em uso. Ressalta-se também, que os

investimentos feitos no projeto serão recuperados em pouco tempo, pois ele poderá reduzir em até 50% ou extinguir a necessidade de docagem dos navios na fase 1 do ciclo de alinhamento.

Na figura 2.17 pode-se visualizar o gabinete onde foi montado o *Tilt Test* Eletrônico. O gabinete, com tamanho de 10U[2], é composto por:

- Módulo contendo monitor LCD, teclado e *touch panel* (tamanho 1U);
- Gaveta de tamanho 3U para o armazenamento dos sensores;
- Computador tipo PC industrial (tamanho 4U); e
- Módulo de alimentação e *No-break* (tamanho 2U).

Figura 2.17: Gabinete do *Tilt Test* Eletrônico

2.4 Considerações e Motivação para um Novo Protótipo

Como apresentado neste Capítulo, a realização do alinhamento, especificamente a fase 1 do ciclo de alinhamento (*Tilt Test*) pois é o período a que este trabalho se dedica, é de vital importância para o bom emprego e eficácia dos meios navais.

[2] *Rack Unit* (U ou RU) é a unidade de medida que descreve a altura dos equipamentos projetados para instalação em *racks* com largura de 19" (482,6 mm) ou 23" (584,2 mm). Um *Rack Unit* equivale a 1,75" (44,45 mm) de altura.

A necessidade de docagem dos navios para a realização do alinhamento das plataformas, considerando as diversas exigências para alcançar as "condições de *Tilt Test*", sugere um considerável gasto de recursos financeiros. Com a implementação do projeto piloto do *Tilt Test* Eletrônico, a necessidade de docagem pode ser eliminada num primeiro momento (ainda necessário para a verificação da verticalidade das agulhas giroscópicas, porém com periodicidade maior). Isto significa que, não havendo a necessidade de verificação das "Giros", todo o alinhamento das plataformas dos elementos pode ser realizado com o navio atracado no porto. Como as medições são relativas, capturadas por sensores eletrônicos e os requisitos de tempo para a leitura simultânea dos sensores foram cumpridos, o balanço do navio atracado não é fator limitante.

Como o *Tilt Test* Eletrônico permite a obtenção do *tilt* com o navio flutuando, os resultados adquiridos são mais fieis porque são obtidos em condições reais de temperatura, dilatação, tensões e torção do navio. Ou seja, as mesmas condições encontradas pelo navio na ocasião dos exercícios de tiro. O *Tilt Test* "convencional" é realizado com o navio em condições estáticas e bastante controladas, portanto são resultados para aquela situação específica e não representam a realidade encontrada pelo navio em operação no mar.

Como foi apresentado também, trata-se de um projeto piloto, então, ocorre a necessidade de uma segunda versão do *Tilt Test* Eletrônico com sensores mais precisos e contendo dois eixos uma vez que o protótipo, por utilizar inclinômetros de um eixo, não eliminou a necessidade de realização das medidas de 10 em 10 graus, ou seja, 36 medidas para cada plataforma. Outro fator a ser considerado é o tamanho e o peso do protótipo desenvolvido que dificulta o transporte e a passagem pelas vigias e escotilhas dos navios.

Figura 2.18: Sensor de Dois Eixos

Item	**Clinômetro de Bolha (usual)**	**Clinômetro T7 [6] (piloto)**	**Clinômetro AKA2000 [7] (precisão)**
Número de Eixos	1	1	2
Precisão	0,25	0,1°	0,02°
Resolução	0,25	0,01°	0,005°
Tempo de Resposta	n/a	0,002 $\sim$ 5 seg	0,002 $\sim$ 5 seg
Aferição 0°	Bancada	Software	Software
Condição do Navio	Docado	Docado ou Atracado	Docado, Atracado ou no Mar

Tabela 2.1: Comparação dos Sensores

O CASOP adquiriu recentemente um novo conjunto de sensores inclinômetros contendo dois eixos e com maior precisão (figura 2.18). Um outro trabalho de caracterização destes sensores ainda está sendo realizado, mas na tabela 2.1 pode-se visualizar as principais características dos sensores disponíveis atualmente.

Capítulo 3

Plataforma Reconfigurável

Neste capítulo será apresentado o desenvolvimento de um sistema computacional implementado em plataforma reconfigurável para ser utilizado como protótipo para a segunda versão do *Tilt Test* Eletrônico. Tratando-se de uma plataforma de desenvolvimento, inicialmente, são apresentados alguns conceitos básicos sobre circuitos reconfiguráveis e fluxo de desenvolvimento do hardware e software. Mas antes, é importante esclarecer que a denominação de "plataforma reconfigurável" refere-se à utilização de dispositivos reconfiguráveis e às facilidades de reuso, em todo ou em parte, dos dispositivos e sistemas implementados e não ao uso de técnicas para a reconfiguração dinâmica de um circuito.

Um sistema computacional pode ser implementado utilizando três diferentes tipos de arquiteturas de hardware: microprocessador de propósito geral (GPP), circuitos integrados de aplicação específica (ASIC) ou dispositivos programáveis em hardware (FPGA).

Os sistemas implementados utilizando microprocessador de propósito geral, do inglês, *General-Purpose Processor* (GPP) apresentam um menor custo e facilidade de implementação. Porém, por não serem projetados especificamente para a tarefa na qual estão sendo utilizados, perdem em desempenho, ou seja, existe um custo pela generalidade [8].

O uso de ASICs (*Application-Specific Integrated Circuit*), em contrapartida, por serem otimizados para desenvolver tarefas específicas, apresentam um desempenho superior. Entretanto, possuem um alto custo de produção, sendo seu uso justificado para produção em grandes volumes de forma a baratear as unidades individuais do componente.

Assim, pode-se colocar os microprocessadores de propósito geral e os ASICs em extremos opostos, considerando flexibilidade e desempenho. Os GPPs apresentam grande flexibilidade de aplicações e baixo custo, mas perdem em eficiência. Os ASICs apresentam alto nível de eficiência mas são restritos a uma aplicação específica e possuem alto custo de implementação e fabricação.

Entre a flexibilidade da arquitetura dos GPPs e o desempenho dos ASICs, pode-se destacar a presença dos dispositivos de lógica programáveis FPGAs (*Field-Programmable Gate Array*) em uma posição intermediária a estas características. A capacidade do usuário programar as funcionalidades do FPGA em campo aliado ao baixo custo envolvido, fazem dos dispositivos de lógica programáveis uma plataforma atraente para diversos tipos de aplicações. O FPGA possui vantagem significativa de performance sobre os microprocessadores devido ao seu alto grau de paralelismo [9]. Comparado ao ASIC, os FPGAs ocupam uma área maior, consomem mais e possuem uma performance inferior mas sua capacidade de reconfiguração o torna uma fácil escolha para diversas aplicações (especialmente as de baixo volume de produção).

As principais vantagens de um FPGA comparado ao projeto de um ASIC podem ser listadas como a seguir:

- O *time-to-market* é mínimo quando comparado ao projeto de um ASIC. Quando os testes com o projeto implementado estiverem concluídos, o circuito poderá ser gravado, iniciando a fase de produção;
- FPGAs são ideais para protótipos. Testes e verificações de hardware podem ser executados no *chip* e os erros podem ser corrigidos facilmente sem custos adicionais;
- Não existindo custo para a geração de máscara torna os FPGAs excelentes para a produção em baixa escala; e
- FPGAs são flexíveis e reprogramáveis, facilitando o uso em mais de um projeto sem aumentar os custos.

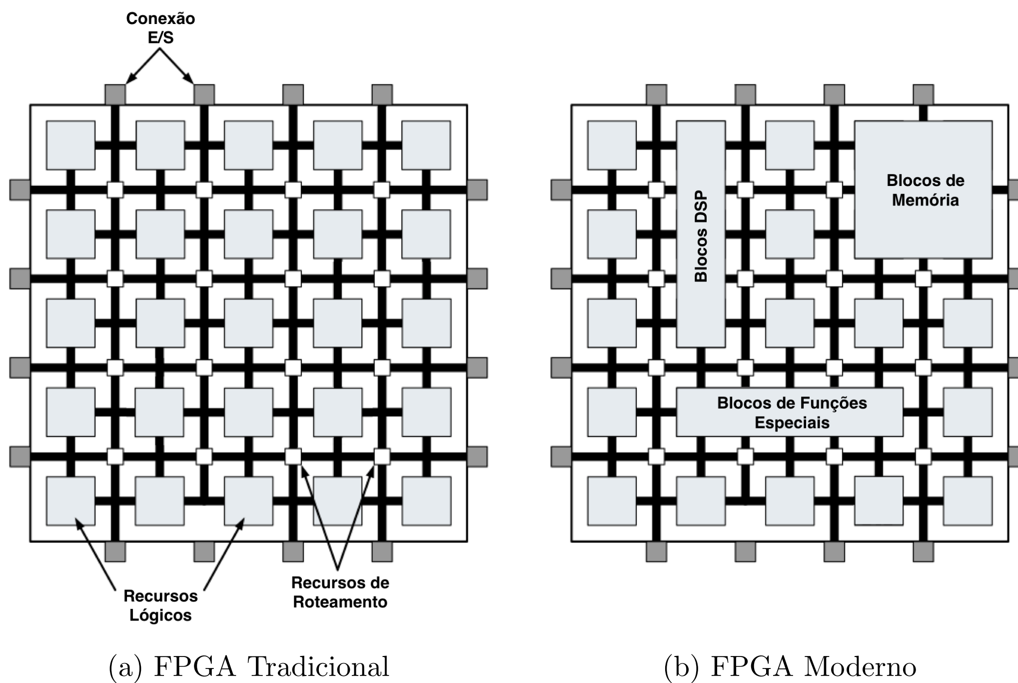
Os ASICs, contudo, podem oferecer menor custo por unidade de produção quando a quantidade é grande o suficiente para compensar os valores de desenvolvimento e geração da máscara. Eles podem também ser especialmente necessários quando há a necessidade de funcionamento por longos períodos utilizando baterias.

3.1 Dispositivos Reconfiguráveis

Dispositivos reconfiguráveis fazem parte de um tipo especial de circuitos integrados os quais possibilitam que os usuários alterem suas funcionalidades lógicas após a etapa de fabricação do *chip*. Os FPGAs são, atualmente, os principais circuitos dessa categoria embora existam outros componentes programáveis como CPLD (*Complex Programmable Logic Device*) e PAL (*Programmable Array Logic*). Como

apresentado anteriormente, os FPGAs podem ser usados com qualquer lógica digital implementada por um ASIC, o que faz estes dispositivos obterem uma aceitação crescente nas últimas décadas.

Tradicionalmente, FPGAs consistem de conexões de entrada/saída, recursos lógicos e recursos de roteamento (figura 3.1a). Contudo, FPGAs mais modernos podem incluir também blocos de memória, blocos para processamento de sinais, *phase-locked loops* (PLL), processadores embarcados, conversores analógico-digital, etc, conforme figura 3.1b.



(a) FPGA Tradicional (b) FPGA Moderno

Figura 3.1: Arquitetura de um FPGA

Os elementos básicos dos FPGAs são:

- Recursos Lógicos

  Estes recursos, assim como sua nomenclatura, variam de fabricante para fabricante mas, basicamente, implementam portas lógicas, tabelas verdade e algum recurso de armazenamento. São os principais componentes do FPGA responsáveis pelo processamento. A quantidade destes elementos representa o tamanho ou quantidade de lógica que o FPGA consegue implementar e os fabricantes utilizam-se deste número como referência para os componentes dentro de uma mesma família.

- Conexões de Entrada/Saída

  Responsáveis pela interface entre os recursos lógicos e o mundo externo, podendo ser configurados como entrada, saída ou bidirecional.

- Recursos de Roteamento

  Responsáveis pelo roteamento e interconexões entre diversos recursos lógicos e entre recursos lógicos e recursos de entrada/saída. Existem também roteamentos dedicados, normalmente usados para sinais de *clock* e *reset* para diminuir atrasos e distorções no sinal. Dispositivos mais modernos possuem a flexibilidade de fornecer diferentes domínios de *clock* dentro do próprio FPGA facilitando projetos assíncronos.

A forma mais comum para utilizar estes recursos é fazendo uso de uma Linguagem de Descrição de Hardware ou, do inglês, *Hardware Description Language* (HDL). É possível realizar a programação utilizando "captura de esquemático" mas como normalmente o sistema a ser implementado em FPGA é bastante complexo, esta acaba não sendo uma boa alternativa.

Ao contrário do que muitos pensam, as HDLs mais populares não foram criadas para serem utilizadas da maneira como são atualmente, ou seja, elas foram criadas com o propósito de permitir documentar o comportamento do hardware. Foi percebido, então, que seria possível simular circuitos completos em um processador genérico. Este processo de tradução de um código HDL para uma forma reconhecível por um GPP para representar o hardware descrito chama-se **simulação** [10].

A simulação mostrou-se extremamente útil para o desenvolvimento e verificação das funcionalidades antes da fabricação do hardware. Posteriormente, as HDLs passaram a ser utilizadas para **sintetizar** o código, gerando automaticamente a configuração lógica para programar um dispositivo específico. As HDLs mais difundidas são: VHDL e Verilog. Na seção 3.2 é apresentada uma breve descrição do VHDL por tratar-se da linguagem principal utilizada para o desenvolvimento do protótipo.

Implementar um projeto em FPGA exige um pequeno fluxo de ações e informações até alcançar o arquivo de programação do dispositivo. Este fluxo é apresentado genericamente na figura 3.2 e suas etapas podem ser descritas como:

1. Inicialmente, o projeto deve ser codificado em uma HDL que permitirá que ele seja simulado e sintetizado para um FPGA. Embora muitas HDLs sejam baseadas em linguagens de software, elas não representam um software mas descrevem o circuito eletrônico necessário para implementar o projeto.

2. A fase de síntese lógica significa que o código-fonte é verificado para erros de sintaxe e hierarquia dos componentes e é gerado um *netlist* contendo as conexões entre os componentes no nível de transferência entre registradores ou, do inglês, *Register Transfer Level* (RTL).

3. A fase chamada síntese física é dividida em duas partes: *Mapping* que determina como o *netlist* será mapeado para o FPGA especificado em relação aos recursos lógicos; e a outra parte, *Place and Route* associa estes componentes mapeados com blocos físicos específicos do FPGA e determina o roteamento necessário para conectar estes blocos, memórias e entradas/saídas.

4. A geração do arquivo de configuração significa a criação de um *bitstream* que é a denominação utilizada para o arquivo que será gravado no FPGA.

Após a "compilação" do código-fonte em cada fase apresentada acima é possível executar uma simulação correspondente (Comportamental, Funcional e Temporal). Na simulação comportamental são verificadas as saídas de acordo com as entradas fornecidas de forma que seja possível validar o comportamento do circuito. Nesta etapa não são considerados os atrasos das portas lógicas. Nas demais simulações são considerados os atrasos provenientes da implementação em hardware do circuito.

É possível também realizar uma verificação no próprio dispositivo FPGA (*in-circuit*), incluindo o hardware necessário para monitoramento adicional e extração de informações.

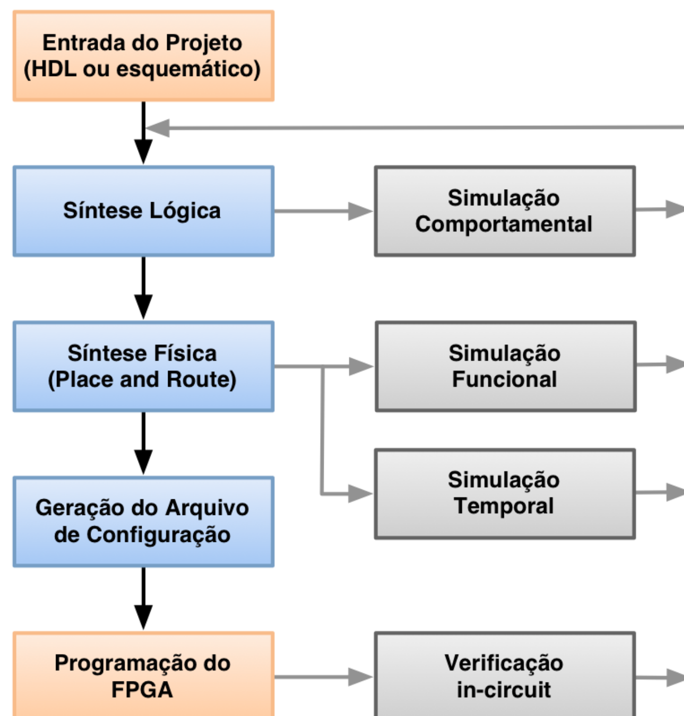


Figura 3.2: Fluxo de Programação de um FPGA

3.2 VHDL

O VHDL (*VHSIC*[1] *Hardware Description Language*) descreve o comportamento de um circuito digital ou sistema. Sua primeira versão é conhecida como VHDL 87 e depois foi atualizada para a famosa VHDL 93. VHDL foi a primeira HDL padronizada pelo *Institute of Electrical and Electronics Engineers* (IEEE), padrão IEEE 1076 [11]. Um padrão adicional, IEEE 1164, foi adicionado para introduzir um sistema lógico multivalorado. O padrão foi atualizado subsequentemente em 2000 e 2002.

VHDL é destinada para a *síntese* e *simulação* de circuitos. Embora, VHDL seja completamente simulável, nem todos seus componentes são sintetizáveis.

Em simulação, os códigos-fontes VHDL são analisados e uma descrição do comportamento é expresso na forma de um *netlist*. Como anteriormente apresentado, *netlist* é uma representação das unidades lógicas utilizadas e como elas são conectadas. As unidades lógicas, normalmente, são portas lógicas e algumas outras primitivas utilizadas pelo dispositivo a ser programado. O comportamento do circuito pode ser verificado fornecendo uma sequência de valores de entrada. O componente utilizado para gerar esses vetores de entrada e colocar o dispositivo em teste chama-se *test bench*.

Em VHDL existe dois tipos principais de estilos ou formas de escrever a descrição do hardware. Ambos estilos são códigos VHDL válidos, contudo, eles podem modelar o hardware diferentemente. Estas formas são:

- **Forma Estrutural** indica que um circuito é construído em partes menores. A descrição especifica que tipos de partes são usadas e como estas partes são conectadas. Nesta forma é empregado o conceito de *componente*. Um componente pode ser uma unidade lógica conhecida (como portas AND, OR ou NOT) ou unidades maiores (por exemplo, multiplicadores) criadas e instanciadas pelo usuário.

- **Forma Comportamental** é descrita em uma linguagem imperativa (procedural) para descrever como as saídas são relacionadas com as entradas na forma de um processo sequencial. Um processo possui uma "lista de sensibilidade" composta de um conjunto de sinais. Quando um sinal na lista de sensibilidade muda, o processo é então ativado. Dentro do processo, a semântica é semelhante a uma linguagem de programação tradicional, podem ser usadas variáveis e a execução é sequencial [12]. Entretanto faz-se necessário um cuidado especial para evitar que sejam criados códigos que não são sintetizáveis.

[1]Very High-Speed Integrated Circuit

Ainda existe um terceiro estilo, talvez o mais utilizado, que é a mistura das duas formas apresentadas.

Uma motivação fundamental para o uso de VHDL (ou ainda, Verilog) é que representa uma linguagem padronizada, independente de tecnologia e fabricantes, portável e reutilizável. As duas principais aplicações do VHDL são para a programação de FPGA/CPLD e também para ASICs. Uma vez o código em VHDL esteja escrito, ele pode ser implementado em um dispositivo programável (fabricado pela Altera, Xilinx, Atmel, etc) ou pode ser submetido para uma fábrica para a produção de um ASIC. Atualmente, muitos dispositivos comerciais complexos (microcontroladores, por exemplo) são projetados desta forma.

3.3 SoPC

O termo *System on Programmable Chip* (SoPC) refere-se à combinação de um processador com um hardware customizado implementado usando um FPGA e blocos de memória. Esta metodologia originou-se no SoC (*System on Chip*), tradicionalmente, implementado em ASIC. A figura 3.3 ilustra um SoC controlador de rede desenvolvido pela Lucent contendo 137M de transistores na tecnologia de 0,16μm. Como já mencionado neste capítulo e principalmente ao custo envolvido, os SoPC ganham cada vez mais espaço como plataforma para a implementação de um sistema computacional.

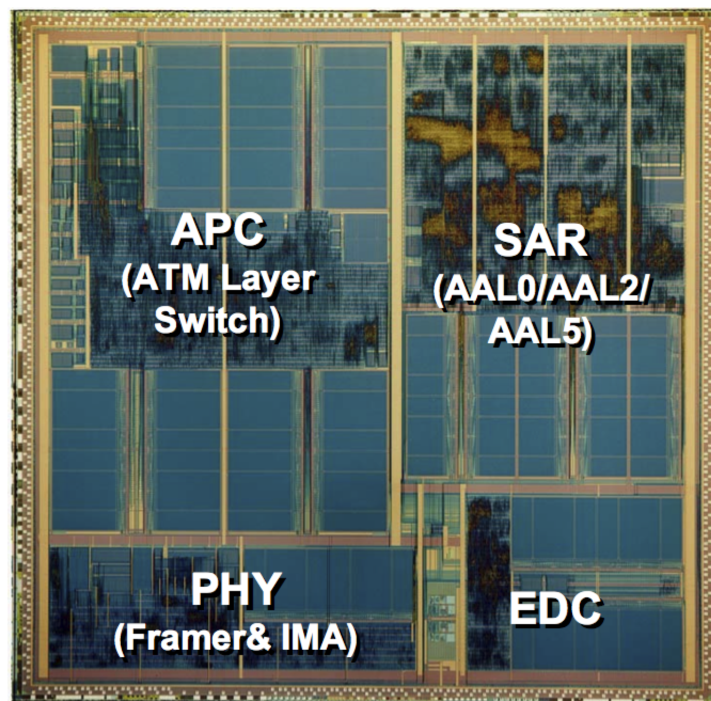


Figura 3.3: Exemplo de um SoC fabricado pela Lucent

De modo a complementar a comparação já apresentada entre FPGA e ASIC, a

tabela 3.1 [13] relaciona algumas características de sistemas computacionais implementados em SoPC (FPGA), SoC (ASIC) e GPP. Os SoPC possuem vantagens e desvantagens em relação às outras alternativas, como vantagem é possível citar a reconfigurabilidade, flexibilidade e um ciclo de desenvolvimento curto.

Característica	**SoPC**	**SoC**	**GPP**
Flexibilidade de SW	●	●	●
Flexibilidade de HW	●	○	○
Reconfigurabilidade	●	○	○
Tempo/Custo Desenvolvimento	○	●	○
Custo de periféricos	○	○	●
Performance	◐	●	●
Custo de produção	◐	○	○
Eficiência/Consumo	○	●	●

Legenda: ● - Alto; ◐ - Moderado; ○ - Baixo

Tabela 3.1: Comparação entre SoPC, SoC e GPP

Quando especificado que um SoPC é composto por um processador e um hardware customizado, entende-se que os processadores podem ser classificados como "*hard-core*", implementado em um chip dedicado de silício, ou "*soft-core*", implementado no próprio FPGA. *Hard processors* oferecem maior performance e menor consumo, enquanto os chamados *soft processors* são mais flexíveis e podem oferecer mais recursos.

Os *hard processors* podem não ser um elemento totalmente distinto do FPGA, ou seja, eles podem representar um processador embarcado em adição aos elementos lógicos programáveis do FPGA. Por exemplo, existem famílias de FPGA da Altera que incluem processadores ARM. A Xilinx, por sua vez, dispõe de FPGA contendo processadores PowerPC. Estas mesmas empresas possuem processadores *soft-core*, Nios II e MicroBlaze/PicoBlaze, respectivamente.

Os processadores *soft-core* utilizam os elementos lógicos disponíveis no FPGA para implementar a lógica do processador. Este tipo de processadores normalmente são cheios de recursos e flexíveis, permitindo ao desenvolvedor especificar a largura dos barramentos da memória, funcionalidades da ULA (Unidade Lógica Aritmética), número e tipos de periféricos e espaço de endereçamento de memória. Mas como flexibilidade também possui um "custo", os *soft-cores* possuem taxas de *clock* menores e utilizam mais corrente que um processador *hard-core* equivalente.

O desenvolvimento de um SoPC envolve, não apenas o projeto do hardware, mas também do software que será executado pelo processador. O software normalmente é escrito em C ou C++ mas pode incluir partes escritas em Assembly.

Em termos de velocidade e utilização de elementos lógicos, um projeto de um SoPC oferece três alternativas para a implementação de um determinado algoritmo:

1. Hardware dedicado com o algoritmo implementado com elementos lógicos.
   - Proporciona maior velocidade porém com menos eficiente uso dos elementos lógicos;
   - Difícil de implementar algoritmos complexos.
2. Algoritmo implementado em software.
   - Facilidade de implementação de algoritmos complexos;
   - Mais eficiente no uso de elementos lógicos, por exemplo, em *loops*.
3. Adição de instrução customizada ao processador.
   - Relação de compromisso entre velocidade e uso de elementos lógicos. Por exemplo, uma aplicação poderá ter ganho de velocidade mas o uso de elementos lógicos também aumentará.

Alguns sistemas, dependendo da aplicação, podem necessitar de multi-tarefa, escalonamento, *threads* e alguns outros recursos avançados de gerenciamento de processos e memória. Para estes casos, podem ser utilizados sistemas operacionais embarcados para gerenciar o hardware e executar os softwares necessários à aplicação. Porém a utilização de módulos de terceiros para a implementação do SoPC, os chamados IP (*Intellectual Property*), bem como o licenciamento para o uso de um determinado sistema operacional, pode aumentar consideravelmente o custo de implementação do projeto e a sua real necessidade deve ser avaliada com bastante critério.

Para facilitar o aprendizado e a implementação de SoPC, alguns fabricantes de FPGA oferecem em seus catálogos, placas para o desenvolvimento de SoPC. Estas placas incluem um FPGA com boa capacidade, diversos tipos de memórias e uma outra variedade de dispositivos de E/S (Entrada/Saída) que são capazes de suportar a implementação de um processador *soft-core* e hardware associado.

3.4 Plataforma de Desenvolvimento

O dispositivo FPGA a ser utilizado para o desenvolvimento deste trabalho é da série Cyclone II da Altera e está disponível no *kit* de desenvolvimento modelo DE2-70 fabricado pela Terasic (figura 3.4). O FPGA utilizado nesta placa de desenvolvimento é um Cyclone II 2C70 [14] com encapsulamento BGA de 896 pinos. Existem alguns dispositivos periféricos conectados aos pinos do FPGA, permitindo ao usuário controlar todos os aspectos de operação da placa.

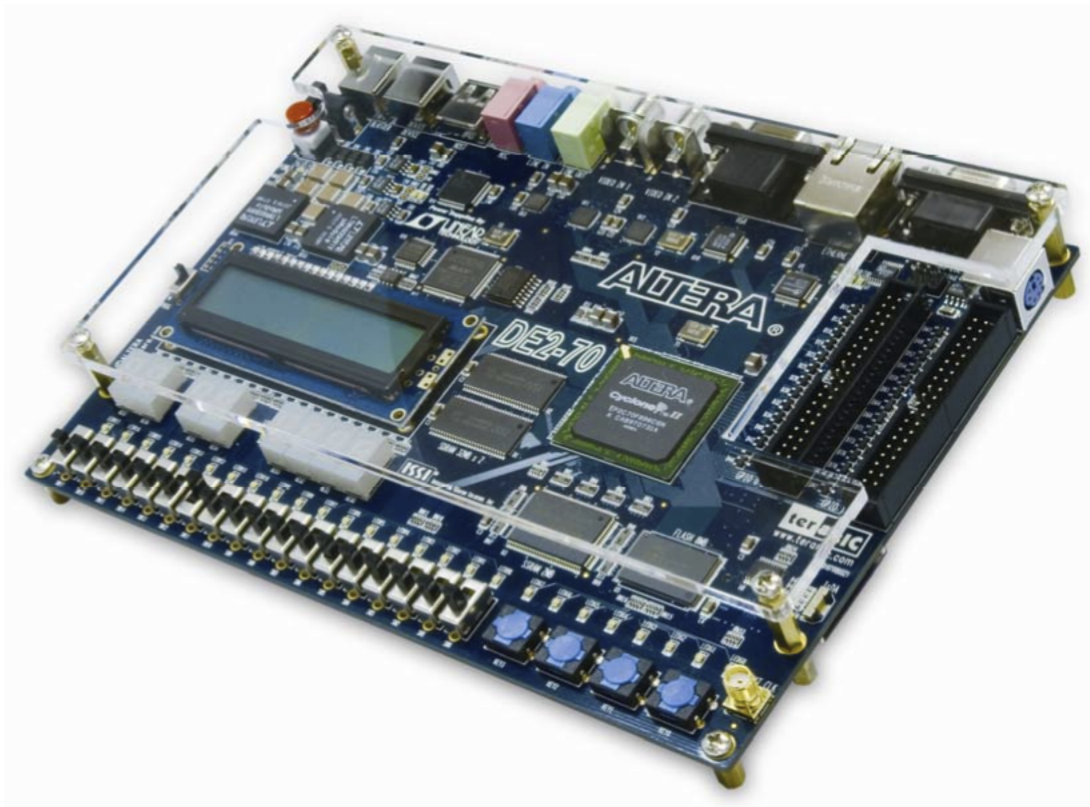


Figura 3.4: Placa de Desenvolvimento Terasic DE2-70

A placa DE2-70 inclui um bom número de chaves, botões, LEDs e *displays* de 7 segmentos. Existem também dispositivos de memória como SSRAM, SDRAM e Flash. A figura 3.5 apresenta o diagrama de blocos do *kit* de desenvolvimento contendo o FPGA e periféricos. A seguir uma breve apresentação dos principais componentes disponíveis:

Cyclone II 2C70 FPGA [14]

- 68.416 LE (elementos lógicos);
- 250 blocos de RAM do tipo M4K;
- 1.152.000 RAM (bits);
- 4 x PLL;
- 622 pinos de E/S.

Circuito USB Blaster e Configuração Serial

- Dispositivo de configuração serial Altera EPCS16;
- USB Blaster disponível *on-board* para programação;
- Modo de programação JTAG.

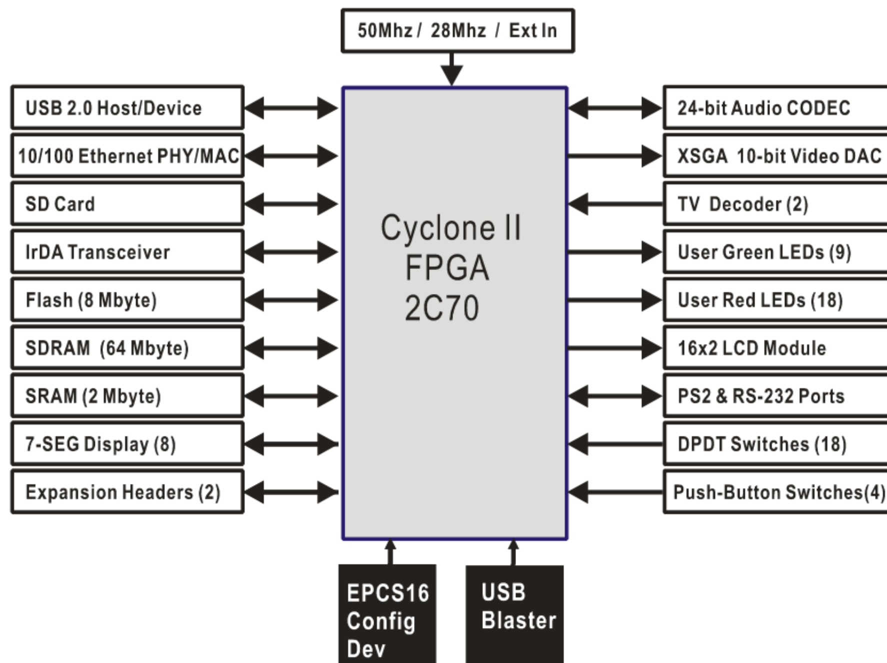


Figura 3.5: Diagrama de Blocos da placa DE2-70

SSRAM [15]

- 2 Mbyte SSRAM (*Synchronous Static RAM*);
- Organizada como 512K x 36 bits.

SDRAM (*Synchronous Dynamic RAM*) [16]

- 2 x 32 Mbyte SDRAM;
- Organizada como 4M x 16 bits x 4 bancos.

Memória Flash [17]

- 8 Mbyte NOR Flash;
- Acesso por *byte* ou *word* (16 bits).

SD Card

- Acesso ao SD card por SPI e modo SD 1-bit.

Clock

- 50 MHz;
- 28,63 MHz;
- Entrada para clock externo SMA.

Codec de Audio

- Codec Wolfson WM8731, 24-bits sigma-delta;
- Entradas Line e Mic, Saída Line;
- Frequência de amostragem: 8 a 96 KHz.

Saída VGA

- DAC ADV7123 140 MHz, 10-bits por canal;
- Conector D-sub com 15 pinos;
- Resolução até 1600 x 1200 em 100 Hz.

Controlador Ethernet 10/100

- MAC e PHY integrados;
- Suporta 100Base-T e 10Base-T;
- Operação *full-duplex* 10 Mb/s e 100 Mb/s;
- Conformidade com o padrão IEEE 802.3u;
- Suporta geração/verificação de *checksum* IP/TCP/UDP.

Controlador USB

- Conformidade com padrão USB 2.0;
- Transferência em alta e baixa velocidade;
- Suporta USB em modo *host* e *device*;
- Suporte a *Programmed I/O* (PIO) e acesso direto à memoria (DMA).

Portas Seriais

- Porta RS-232 com conector DB-9;
- Porta PS/2 com conector PS/2 para conexão de teclado ou mouse.

Conectores para Expansão

- 72 pinos de E/S do FPGA e 8 sinais alimentação estão disponíveis em dois conectores de 40 pinos;
- São utilizados conectores de 40 pinos no mesmo padrão dos discos rígidos IDE para facilitar a conexão.

Decoder de Vídeo

- Dois ADV7180 Decoder vídeo SDTV;

- Suporte ao padrão de cores NTSC/PAL/SECAM;
- ADC 10 bits;
- Entrada CVBS (vídeo composto);

Transceiver IrDA

- Transceiver infravermelho 115,2 Kb/s;
- LED de 32mA;

Utilizando um dispositivo FPGA Cyclone II com boa capacidade e performance, opções de memórias e avançados dispositivos de E/S, a placa DE2-70 é uma ótima plataforma para a implementação e desenvolvimento de sistemas digitais.

Adicionalmente à placa DE2-70, a plataforma de desenvolvimento inclui também um módulo de LCD com *touch screen* de 4.3" com suporte a RGB de 24-bits [18]. Este módulo LCD, chamado de LTM (***L****CD* ***T****ouch Panel* ***M****odule*), também fabricado pela Terasic, possui uma interface que pode ser ligada facilmente a um dos conectores de expansão da placa DE2-70. A figura 3.6 apresenta o módulo LTM e o *flat cable* de 40 pinos que o acompanha.

Figura 3.6: LCD Touch Panel Module

Capítulo 4

Implementação

Neste capítulo é apresentado o desenvolvimento do sistema computacional utilizando a plataforma reconfigurável especificada no capítulo anterior e que será utilizado como protótipo para o *Tilt Test* e, futuramente, para outros projetos já iniciados e que dependem de uma plataforma computacional. Desta forma, definindo uma plataforma básica operacional e com desempenho satisfatório, módulos poderão ser adicionados ou removidos para atender aos requisitos dos demais projetos, ou seja, reconfigurar a plataforma computacional sem, com isso, aumentar os custos envolvidos.

O sistema computacional que será apresentado é um SoPC e sua coincidência com a sigla do Centro de Apoio a Sistemas Operativos (CASOP) originou o nome do sistema implementado: **CASoPC**.

O desenvolvimento apresentado nas próximas seções tem como propósito documentar os principais módulos do sistema com os seus registradores e funções que poderão ser acessadas via software e, sempre quando possível, oferecer alguns aspectos técnicos da implementação.

Um diagrama completo do sistema **CASoPC** pode ser verificado na figura 4.1.

Os módulos apresentados na cor cinza serão implementados no futuro e aqueles conectados por linhas tracejadas foram implementados e testados, porém não foram incluídos na versão atual do sistema, principalmente, por pendências de solução na camada de software.

4.1 Processador *soft-core*

Os processadores *soft-core* apresentam um conjunto de benefícios que os tornam bastante atraentes para uso em sistemas computacionais. A escolha do processador a ser incluído no **CASoPC** considerou principalmente três características básicas: desempenho, área ocupada e facilidade de programação. No entanto, outras características importantes foram examinadas:

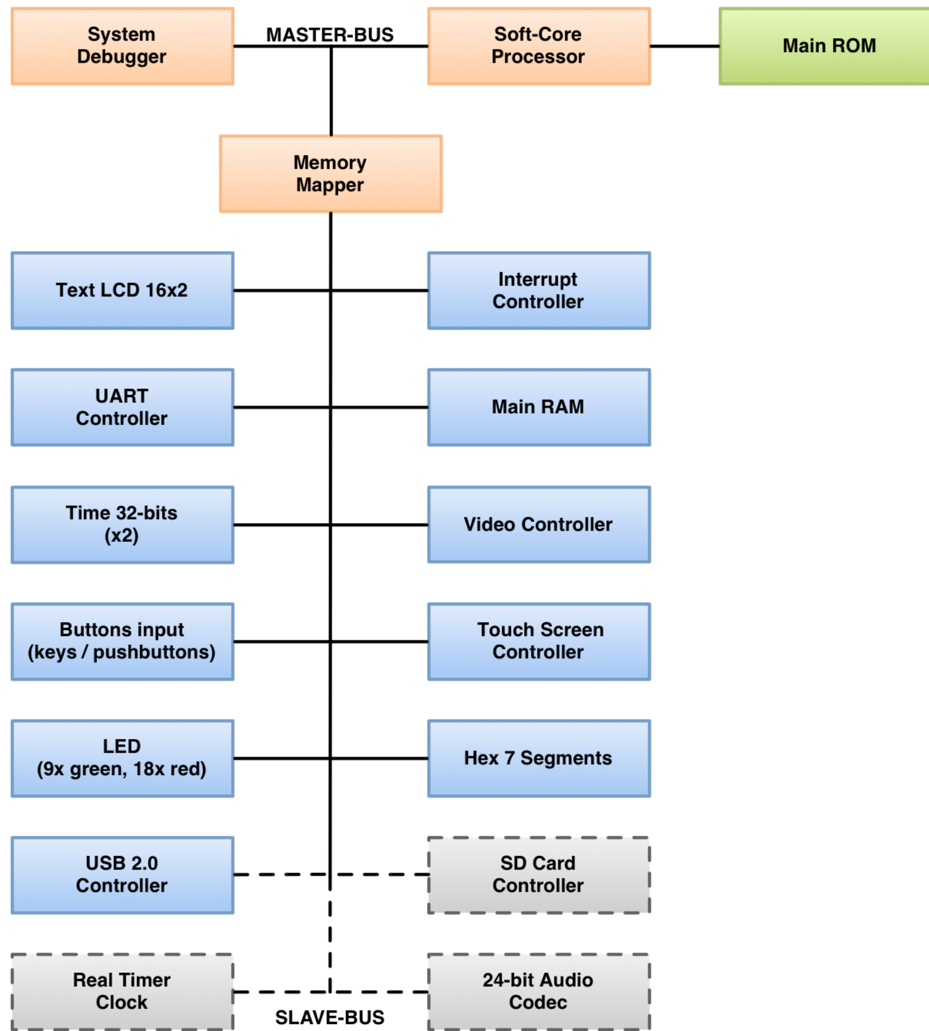


Figura 4.1: Diagrama de Blocos do CASoPC

- Flexibilidade do processador, ou seja, a possibilidade e facilidade para a remoção ou inclusão de unidades funcionais (unidades de ponto-flutuante, unidades de divisão e multiplicação em hardware, memórias cache, etc).

- Compatibilidade com um barramento de comunicação simples e eficiente de forma a facilitar a integração do processador e demais componentes.

- Documentação, legibilidade do código e organização hierárquica do projeto.

- Portabilidade, ou seja, se o *soft-core* é restrito a uma plataforma ou a um determinado fabricante.

Algumas deficiências dos processadores *soft-core* implementados em dispositivos reconfiguráveis, muitas vezes, estão relacionadas à limitação dos recursos lógicos e à menor velocidade de processamento. A implementação de uma arquitetura RISC (*Reduced Instruction Set Computing*) tenta minimizar estes problemas pois consegue

velocidades de processamento mais altas com menor utilização de recursos. Por esta razão, um grande número de *soft-cores* existentes implementa uma arquitetura RISC [19] e são baseados em arquiteturas existentes como MIPS, PowerPC, ARM e SPARC, por exemplo.

A facilidade de programação está diretamente relacionada com a existência de um *tool-chain* (ferramentas de desenvolvimento) que possibilite a programação dos componentes de software em C ou C++. Programar em Assembly, apesar de permitir excelente otimização por ser uma linguagem mais próxima da arquitetura, é extremamente penosa para o programador [19]. A existência de um *debugger* contribui também para uma maior facilidade de programação, uma vez que, facilita e agiliza a detecção e correção de erros.

Documentação, legibilidade do código e organização hierárquica do projeto são importantes, principalmente, quando há a necessidade de realizar modificações ou adaptação no código HDL do processador, facilitando a compreensão do seu funcionamento e das modificações necessárias para a sua utilização. Por fim, a portabilidade implica em não limitar a implementação do sistema a uma única plataforma.

Muitos dos processadores avaliados foram testados na prática, ou seja, a documentação e códigos foram, de alguma forma, analisados e sintetizados para permitir uma melhor verificação da flexibilidade e portabilidade. Porém, para obter uma comparação razoavelmente padronizada foram utilizados alguns estudos publicados ([19], [20], [21], [22] e [23]). Na tabela 4.1 são apresentados alguns atributos considerados importantes para a escolha de um *core* para o sistema.

CPU	F_{MAX} **(MHz)**	**Arquitetura**	***Pipeline***	**Interface**	***open source***
MicroBlaze	200	MicroBlaze 32-bit RISC	3 ou 5	FSL	Não
Nios II/f	200	Nios II 32-bit RISC	6	Avalon	Não
UT Nios	83	Nios 32-bit RISC	4	Avalon	Sim
aeMB	279	MicroBlaze 32-bit RISC	3	Wishbone e FSL	Sim
OpenFire	198	MicroBlaze 32-bit RISC	3	FSL	Sim
MB-Lite	229	MicroBlaze 32-bit RISC	5	Wishbone	Sim
SecretBlaze	90	MicroBlaze 32-bit RISC	5	Wishbone	Sim

Tabela 4.1: Características dos Processadores Avaliados

O processador UT Nios refere-se a uma versão *open source* do Nios I (16-bits), porém, usando uma implementação de 32-bits. Embora na tabela 4.1 conste apenas

a versão Nios II/f da Altera, também foram verificadas as versões Nios II/s e Nios II/e. A seguir é apresentada uma breve descrição das três versões:

Nios II/f
: Otimizado para desempenho, utiliza 6 estágios de pipeline e executa 1 instrução por ciclo de clock. Esta versão inclui ainda *caches* separadas para instruções e dados, MMU (*Memory Management Unit*), MPU (*Memory Protection Unit*) e operações de multiplicação, divisão e deslocamentos em hardware.

Nios II/s
: Desenvolvido para ser uma versão intermediária na relação desempenho vs otimização (área ocupada). Possui 5 estágios de pipeline, executa 1 instrução por ciclo, possui *cache* para instruções e operações de multiplicação, divisão e deslocamentos em hardware.

Nios II/e
: Otimizado para ocupar a menor área possível. Esta versão não utiliza pipeline e, por este motivo, executa 1 instrução a cada 6 ciclos e não implementa nenhum bloco aritmético adicional.

O número de elementos lógicos utilizado pelo Nios II é comparável ao ocupado pelo MicroBlaze, exceto a versão Nios II/e que utiliza significativamente menos elementos lógicos.

Na análise realizada, optou-se por um processador *soft-core* de código aberto (*open source*), não só pela vantagem de não estar limitado a uma determinada plataforma mas, se necessário, acesso ao código-fonte do processador para realizar alterações ou adaptações que possam ser necessárias. Adicionalmente, a diferença de desempenho entre os processadores *open source* e as versões comerciais é relativamente pequena.

Depois de uma análise cuidadosa, a escolha do *soft-core* limitou-se aos processadores *SecretBlaze* [22] e *MB-Lite* [20]. Ambos os processadores apresentam bons resultados em termos de desempenho, possuindo tempo de execução inferior aos outros *cores* avaliados e com utilização de recursos relativamente baixa.

Os dois processadores possuem desempenho parecidos e, embora o SecretBlaze possua desempenho um pouco superior, por fim, optou-se pela escolha do processador MB-Lite, principalmente, pela implementação ligeiramente mais simples e menor. O esquema de barramento empregado também é mais simples e de fácil adaptação.

4.1.1 Processador MB-Lite

O processador MB-Lite é uma implementação em VHDL e de código aberto da arquitetura MicroBlaze da Xilinx [24]. A arquitetura deste processador RISC se assemelha aos conhecidos MIPS e DLX. As ferramentas de desenvolvimento (*tool chain*) estão largamente disponíveis, existindo alguns pacotes de desenvolvimento de software e, ainda, uma versão do sistema operacional uClinux para esta arquitetura.

O MB-Lite implementa 32 registradores internos, sendo que alguns são reservados para funções específicas, como armazenar o endereço de retorno de uma interrupção, por exemplo.

O MicroBlaze usa o formato *Big-Endian* para ordenar os bytes e bits individualmente. Na implementação do MB-Lite essa ordenação foi realizada utilizando outro formato (*Little-Endian*), onde o bit 31 é considerado o mais significativo (MSB) enquanto o bit 0 é o menos significativo (LSB). Como os acessos à memória são efetuados em bytes e as leituras e escritas são alinhadas, o formato de ordenação não altera o funcionamento do sistema. Na figura 4.2 pode-se observar o formato dos dados e das instruções Tipo A e Tipo B do MicroBlaze e do MB-Lite.

MicroBlaze	0 5	6 10	11 15	16 20	21 31
MB-Lite	31 26	25 21	20 16	15 11	10 0
Tipo A	Opcode	reg D	reg A	reg B	Function
Tipo B	Opcode	reg D	reg A	Immediate	

(a) Formato das Instruções

MicroBlaze	0 7	8 15	16 23	24 31
MB-Lite	31 24	23 16	15 8	7 0
Byte Address	n	n + 1	n + 2	n + 3
Byte Order	MSB			LSB

(b) Formato dos Dados

Figura 4.2: Formato adotado no MicroBlaze e MB-Lite

As instruções implementadas no MB-Lite possuem a mesma latência definida na especificação da arquitetura do MicroBlaze. A maioria das instruções é executada em um ciclo de clock. As instruções de salto são executadas em 2 ou 3 ciclos dependendo do tipo de salto.

No MB-Lite as interrupções podem ser utilizadas da mesma forma como na implementação original do MicroBlaze. Isto também significa que é implementada apenas uma linha de interrupção e, consequentemente, existe apenas uma função associada para o seu atendimento.

O MB-Lite implementa os cinco clássicos estágios de *pipeline*: IF (*Instruction Fetch*), ID (*Instruction Decode*), EX (*Execute*), MEM (*Memory*) e WB (*Write-Back*). O módulo IF alimenta o *pipeline* com as instruções requisitadas e armazena o

PC (*Program Counter*) atual. O PC corresponde ao endereço da instrução dividido por 4, uma vez que, as instruções ocupam sempre 32-bits, não é necessário um endereçamento com granularidade menor. O PC é incrementado automaticamente a cada ciclo de clock[1].

No estágio ID (também chamado de OF, *Operand Fetch*), as instruções são traduzidas em sinais de controle. Estes sinais de controle atravessam o *pipeline* juntamente com as instruções. As interrupções também são avaliadas neste estágio, onde é verificada se a execução normal pode ser interrompida sem causar problemas.

No estágio EX todas as operações aritméticas são executadas. Devido à dependência dos dados, neste estágio também é verificada a origem dos operandos para evitar problemas de consistência. Estes dados podem ser oriundos do próprio estágio de execução, da memória, do estágio de *write-back* ou de um registrador.

A utilização de 5 estágios de *pipeline* é um método efetivo para explorar algum tipo de paralelismo. Na figura 4.3 é possível observar a estrutura do *pipeline* do MB-Lite com as entradas e saídas de cada estágio. Os sinais representados são: dados (verde), endereço (azul) e controle (vermelho). Na parte superior da figura são apresentados os sinais de *write-back* enquanto na parte inferior são exibidos os sinais de controle.

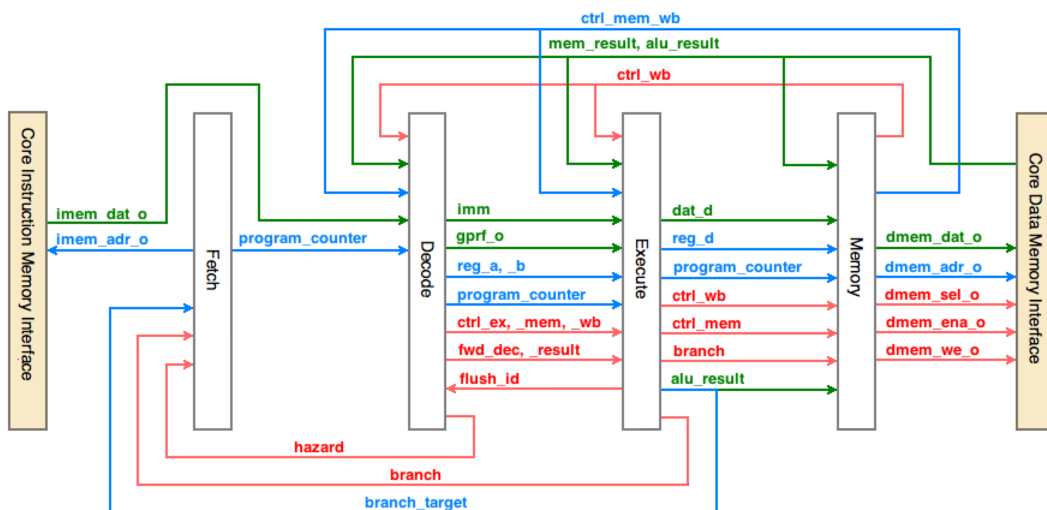


Figura 4.3: Estrutura do MB-Lite

O MB-Lite inclui uma interface de 32-bits para comunicação com um módulo de memória RAM. A interface consiste de um barramento de endereços, dois barramentos para dados (leitura e escrita), um sinal para habilitação de módulo e habilitação de escrita e, ainda, um sinal de 4-bits para indicar quais bytes da palavra de 32-bits devem ser considerados. Adicionalmente existe um sinal para "pausar" (*halt*) o pro-

[1] A arquitetura do IF e PC implicará diretamente na forma de implementação dos módulos de Vídeo e de memória ROM, como serão apresentados nas seções 4.6 e 4.9

cessador. A implementação desta interface é flexível para permitir a adaptação de um barramento padronizado como AMBA ou Wishbone, por exemplo.

Com o objetivo de simplificar a ligação de diversos componentes como memória, coprocessadores, adaptadores, entre outros periféricos, o MB-Lite inclui um decodificador de endereços configurável. Este decodificador de endereços pode ser conectado diretamente à interface da memória de dados para multiplexar o acesso às diferentes interfaces secundárias. O decodificador de endereços é responsável por decodificar os endereços com base num mapeamento de memória genérico e encaminhar os sinais de controle, endereço e dados para as portas de saída apropriadas (figura 4.4). O decodificador pode ser conectado diretamente ao barramento de dados sem introduzir atraso adicional significativo.

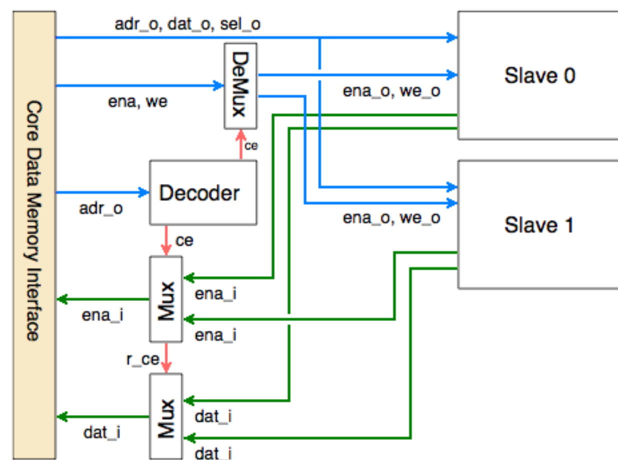


Figura 4.4: Estrutura do Decodificador de Endereços

Após o endereço ter sido decodificado, um dos módulos externos é ativado pelos correspondentes sinais de habilitação e habilitação de escrita. Os outros sinais que são conectados a todos os módulos (`dat_o`, `adr_o` e `sel_o`) podem ser conectados diretamente pois eles não influenciam no estado dos módulos.

Na seção a seguir será avaliado o uso de um barramento padronizado para interligar todos os componentes do sistema. Embora o MB-Lite inclua uma interface eficiente para acesso à memória e um decodificador de endereços, estes poderão ser substituídos por uma solução padronizada para facilitar a integração com os demais módulos do sistema. Embora, seja bom ter em mente, que a utilização de uma solução externa pode aumentar a quantidade de ciclos necessários para concluir uma transação entre o processador e os demais dispositivos do sistema.

4.2 Barramento de Comunicação

Para que os módulos de um sistema computacional possam se comunicar é necessário que exista uma estrutura de interconexão entre cada um dos componentes. A esta estrutura ou caminho de conexão é dado o nome de barramento.

Uma característica básica dos barramentos é o compartilhamento do meio de transmissão, ou seja, vários dispositivos podem estar conectados a um mesmo barramento. Desta forma, um sinal enviado por um componente pode ser recebido por todos os demais componentes que pertencem a esta mesma conexão. Consequentemente, se dois ou mais dispositivos decidirem enviar sinais ao barramento no mesmo instante, estes sinais irão se sobrepor e serão corrompidos. Dessa forma, deve ser permitido que apenas um dispositivo transmita sinais pelo barramento a cada instante.

Normalmente, o barramento de um sistema possui diversas linhas de conexão distintas. Cada uma dessas linhas tem uma função específica dentro da estrutura de interconexão. No entanto, é possível classificar estas linhas em 3 grupos funcionais: *dados*, *endereços* e *controle* [25].

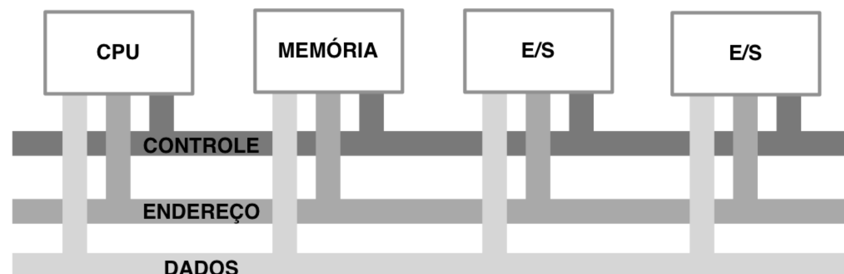


Figura 4.5: Esquema de Interconexão de Barramento

Barramento de Dados

Este barramento é usado para transferir dados entre os módulos do sistema. Em um sistema computacional de 32-bits, normalmente, este barramento também é composto por 32 linhas (largura do barramento). Como cada linha consegue transmitir apenas um bit por vez, a quantidade de linhas determina quantos bits podem ser transferidos de uma vez. O desempenho do sistema sofre forte influência da largura do barramento de dados. Por exemplo, se o barramento de dados for composto por apenas 16 linhas de largura e cada instrução possuir tamanho igual a 32-bits, o processador precisará acessar 2 vezes o módulo de memória em cada ciclo de instrução.

Barramento de Endereços

A função básica deste barramento é designar o destino dos dados que estão

sendo transferidos pelo barramento. Um fato interessante é que a largura do barramento de endereços determina a capacidade máxima da memória. Por exemplo, supondo que um barramento de endereços seja de 24-bits, dos quais 16-bits são utilizados para representar um endereço de memória, portanto, é possível endereçar 2^{16} posições de memória com este tamanho de endereço. Normalmente, as linhas de endereços também são utilizadas para endereçar um módulo de E/S, onde, os bits mais significativos identificam o módulo e os bits menos significativos identificam a porta de E/S ou uma posição de memória deste componente.

Barramento de Controle

As linhas de controle são importantes para controlar o acesso aos barramentos de endereço e de dados, uma vez que o uso deles é compartilhado entre os módulos do sistemas. Os sinais de controle podem ser destinados a definir operações ou comandos e informações de temporização. Os sinais de temporização indicam a validade das informações.

Os barramentos em um dispositivo programável podem operar da mesma forma que um barramento convencional e compartilhar da mesma estrutura funcional. De qualquer maneira, é importante que esse barramento e a própria forma de comunicação entre os módulos seja baseada em algum padrão para contornar o problema de integração de dispositivos construídos por diferentes desenvolvedores.

Existem algumas arquiteturas disponíveis de barramentos, sendo algumas abertas para uso e outras, apesar de gratuitas, necessitam de licença. Entre estas alternativas, é possível citar: **WISHBONE** (Silicore), **AMBA** (ARM), **Avalon** (Altera) e **CoreConnect** (IBM).

Todas essas arquiteturas mencionadas possuem pontos bastantes favoráveis ao seu uso e não serão comentados neste trabalho. Optou-se, para o desenvolvimento do protótipo do SoPC, pela escolha do barramento WISHBONE que possui uma especificação robusta o suficiente para assegurar a compatibilidade entre *IP cores*.

A empresa criadora do padrão, Silicore Corporation, tornou-o aberto ao domínio público e, em 2001, a organização OpenCores [26] adotou o Wishbone como padrão de conectividade por ser flexível, simples e de padrão completamente aberto. O OpenCores, além de manter seus próprios projetos de SoPC e ASICs, é responsável ainda por manter uma grande base de projetos (*IP cores*) de código aberto. Sendo esta uma excelente fonte de pesquisa para códigos HDL diversos.

4.2.1 Barramento Wishbone

O barramento Wishbone, formalmente chamado de *Arquitetura de Interconexão Wishbone para SoC*, é um barramento de dados destinado a efetuar uma ligação

padronizada entre diferentes *cores* utilizando uma solução flexível para facilitar o reuso e diminuir os problemas de integração. Isto é realizado através da criação de uma interface comum entre os módulos IP, obtendo-se maior portabilidade, confiabilidade e, consequentemente, menor tempo necessário para realizar a integração.

O padrão obtido não requer uma ferramenta, seja de software ou hardware, para o seu desenvolvimento. O Wishbone é destinado a ser um barramento lógico e, desta forma, não especifica nenhuma informação elétrica. Ao invés disso, a especificação é escrita em termos de sinais, ciclos de clock e níveis lógicos, alto e baixo. Resumidamente, o padrão destina-se a facilitar a comunicação e a integração de módulos ou projetos em VHDL, Verilog ou qualquer outra linguagem de descrição de hardware.

A arquitetura Wishbone foi fortemente influenciada por três fatores: (i) a necessidade de uma solução para a integração confiável de módulos de um SoC; (ii) necessidade de uma especificação de interface comum para facilitar o projeto estruturado em grandes projetos; e (iii) utilizar como referência, soluções tradicionais de integração de sistemas disponibilizadas para microcomputadores, como barramentos PCI e VME.

Uma das soluções para obter a flexibilidade desejada para o padrão, foi a criação de requisitos mínimos para garantir a compatibilidade entre as interfaces. Desta forma, a especificação do padrão utiliza cinco palavras chaves para descrever as operações previstas, são elas: *Regra*, *Recomendação*, *Sugestão*, *Permissão* e *Observação* [27].

Por exemplo, o padrão especifica como *Regra* que os nomes dos sinais devem obedecer às regras do ambiente de desenvolvimento e da HDL utilizada e, especifica como *Recomendação*, que as interfaces utilizem os nomes convencionados no padrão. Uma dessas convenções é utilizar "`_i`" e "`_o`" ao final do nome do sinal para indicar que o mesmo é um sinal de entrada (para o módulo) ou saída (do módulo).

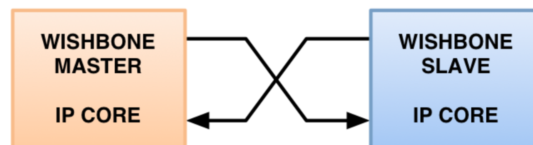


Figura 4.6: Interconexão Ponto-a-Ponto

O barramento Wishbone define dois tipos de interfaces: *Master* e *Slave*. As conexões entre as interfaces *Master* e *Slave* podem ser estabelecidas de diversas maneiras, desde uma simples conexão ponto-a-ponto (figura 4.6) até as formas mais complexas como conexões em matriz com a utilização de árbitros para acesso ao barramento. Seja qual for a topologia do barramento usado, sempre haverá um *Master* transferindo dados para um *Slave* [28].

A figura 4.7 apresenta a topologia utilizada no desenvolvimento do **CASoPC** para a conexão entre os módulos periféricos. Esta topologia de barramento compartilhado é uma estrutura facilmente encontrada em projetos de SoPC, porém, devido ao compartilhamento do barramento, não é a estrutura que oferece maior desempenho, contudo, é suficiente para a maioria dos sistemas computacionais mais simples.

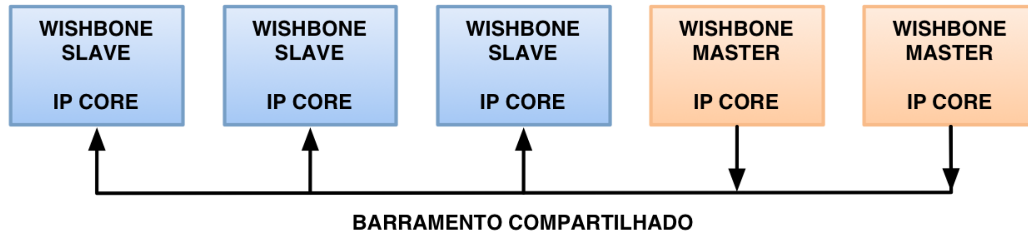


Figura 4.7: Interconexão com Barramento Compartilhado

A conexão entre os barramentos *Master* e *Slave* normalmente não é realizada diretamente, ou seja, a conexão é realizada utilizando-se um módulo adicional (no **CASoPC** é realizado por um mapeador de memória, descrito na seção 4.4). Omitindo, por hora, este módulo adicional de mapeamento, uma conexão básica entre os barramentos *Master* e *Slave* é apresentada na figura 4.8.

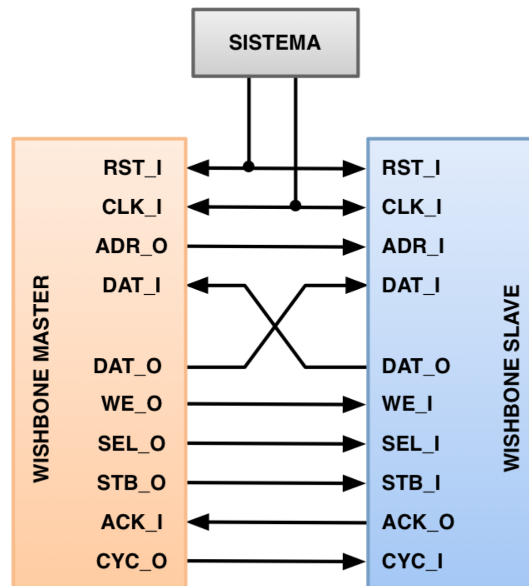


Figura 4.8: Conexão Padrão entre *Master*-*Slave*

Alguns dos sinais especificados no padrão podem ou não estar presentes em uma determinada interface, indicando que alguns sinais são opcionais. Os sinais das interfaces implementadas no **CASoPC** são descritos a seguir e estão divididos em três grupos: sinais comuns às interfaces *Master* e *Slave*, sinais exclusivos para o *Master* e exclusivos para o *Slave*.

1. Sinais das interfaces *Master* e *Slave*:

   `CLK_I`
   : Clock de entrada. Todos os sinais de saída são registrados na borda de subida do `CLK_I`.

   `DAT_I()`
   : Barramento para entrada de dados binários com tamanho máximo de 64-bits.

   `DAT_O()`
   : Barramento para a saída de dados binários com tamanho máximo de 64-bits.

   `RST_I`
   : O sinal de reset força a interface Wishbone a reiniciar. Todas as máquinas de estados internas deverão ser reinicializadas. Este sinal somente reinicia a interface Wishbone e não, necessariamente, reinicia os demais componentes do sistema, embora possa ser utilizado desta maneira.

2. Sinais exclusivos para o *Master*:

   `ACK_I`
   : A entrada do sinal de confirmação (*acknowledge*), quando em nível alto, indica o término normal de um ciclo de acesso ao barramento.

   `ADR_O`
   : O barramento de saída de endereço é usado para transmitir um endereço binário.

   `CYC_O`
   : O sinal de ciclo de acesso ao barramento indica que um ciclo válido está em andamento. O sinal é declarado para toda a duração do ciclo do barramento. Por exemplo, durante o ciclo de transferência de um bloco de dados, o sinal deve ser definido durante a primeira transferência e permanecer atribuído até a transferência do último dado.

   `SEL_O()`
   : O sinal de *select* indica a posição em que se encontra o dado válido no sinal `DAT_I` durante um ciclo de leitura ou `DAT_O` durante um ciclo de escrita. O tamanho do vetor do sinal `SEL_O()` é determinado pela granularidade de uma porta. Por exemplo, se é utilizada uma granularidade de 8-bits em uma porta de 32-bits, então, haverá 4 grupos de 8-bits e o sinal de *select* será definido como `SEL_O(3..0)`. Cada bit deste sinal é relacionado a um

dos 4 conjuntos de 8-bits. Definir o valor "0011" para este sinal, indica que o dado válido corresponde aos primeiros 16-bits de `DAT_I` ou `DAT_O`.

`STB_O`
O sinal de *strobe* indica um ciclo de transferência de dados válido e qualifica vários outros sinais na interface, como o `SEL_O()` por exemplo. O *Slave* definirá um sinal `ACK_I`, `ERR_I` ou `RTY_I` em resposta à definição do sinal `STB_O`.

`WE_O`
O sinal de escrita (*write enable*) indica se o ciclo é de leitura ou escrita. O valor de nível alto é atribuído durante escrita e nível "0" para leitura.

3. Sinais exclusivos para o *Slave*:

   `ACK_O`
   O módulo *Slave* define o sinal de saída de *acknowledgement* indicando o término normal de um ciclo de acesso ao barramento.

   `ADR_I()`
   O `ADR_I` é usado para o módulo receber um endereço binário.

   `CYC_I`, `SEL_I()`, `STB_I`, `WE_I`
   A descrição destes sinais é semelhante ao apresentado na interface *Master*, porém, com direção contrária. Na interface *Master* eles são definidos como saída e na interface *Slave* como entrada.

Nesta apresentação dos sinais é possível perceber o uso de parênteses em alguns nomes de sinais. Esta também é uma recomendação do padrão, ou seja, a indicação que um sinal é um vetor ou barramento utilizando os símbolos "( )".

Para entender o funcionamento dos ciclos de leitura e escrita, a figura 4.9 [29] apresenta uma operação de leitura seguida de uma operação de escrita executada pelo *Master* a um dispositivo de memória.

1. O *Master* inicia o ciclo definindo os sinais `CYC_O`, `STB_O`, `SEL_O` e `ADR_O`. Define também o sinal `WE_O = 0` indicando uma operação de leitura e, consequentemente, o sinal `DAT_O` não é definido.

2. *Master* recebe o sinal `ACK_I` e o dado a ser lido encontra-se disponível em `DAT_I`.

3. O dispositivo *Master* realiza um ciclo de escrita, ou seja, `WE_O = 1`. É definido também um novo endereço em `ADR_O` e o dado a ser gravado é colocado em `DAT_O`.

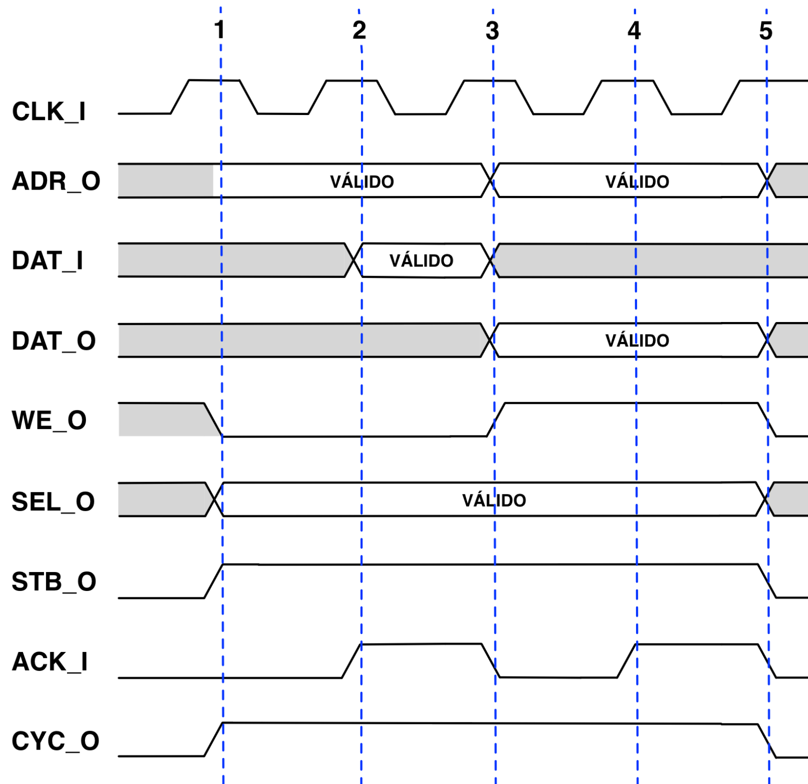


Figura 4.9: Ciclo de Leitura e Escrita

4. O *Slave* define o sinal *acknowledge* indicando ao *Master* que o dado foi recebido.

5. *Master* libera o barramento colocando os sinais `CYC_O` e `STB_O` em nível lógico "0".

No trecho de código VHDL a seguir, encontra-se a definição dos barramentos *Master* e *Slave* do **CASoPC**. Para facilitar a implementação, cada barramento foi divido em dois grupos, um contendo os sinais de entrada e o outro para os sinais de saída. Desta forma, o sistema desenvolvido utiliza quatro barramentos para a transferência de dados e todos os módulos *Slaves* deverão implementar dois barramentos Wishbone: *Slave* de entrada e *Slave* de saída.

```
-- WB-master inputs from the wb-slaves
type wb_mst_in_type is record
    clk_i : std_logic;                                    -- master clock input
    rst_i : std_logic;                                    -- synchronous active high reset
    dat_i : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0); -- databus input
    ack_i : std_logic;                                    -- buscycle acknowledge input
    int_i : std_logic;                                    -- interrupt request input
end record;

-- WB-master outputs to the wb-slaves
type wb_mst_out_type is record
    adr_o : std_logic_vector(CFG_DMEM_SIZE - 1 downto 0);  -- address bits
```

```
    dat_o : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);   -- databus output
    we_o  : std_logic;                                       -- write enable output
    stb_o : std_logic;                                       -- strobe signals
    sel_o : std_logic_vector(3 downto 0);                    -- select output array
    cyc_o : std_logic;                                       -- valid BUS cycle output
end record;

-- WB-slave inputs, from the WB-master
type wb_slv_in_type is record
    clk_i : std_logic;                                       -- master clock input
    rst_i : std_logic;                                       -- synchronous active high reset
    adr_i : std_logic_vector(CFG_DMEM_SIZE - 1 downto 0);    -- address bits
    dat_i : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);   -- Databus input
    we_i  : std_logic;                                       -- Write enable input
    stb_i : std_logic;                                       -- strobe signal
    sel_i : std_logic_vector(3 downto 0);                    -- select output array
    cyc_i : std_logic;                                       -- valid BUS cycle input
end record;

-- WB-slave outputs to the WB-master
type wb_slv_out_type is record
    dat_o : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);   -- Databus output
    ack_o : std_logic;                                       -- Bus cycle acknowledge output
    int_o : std_logic;                                       -- interrupt request output
end record;

type wb_slv_in_array_type is array(natural range <>) of wb_slv_in_type;
type wb_slv_out_array_type is array(natural range <>) of wb_slv_out_type;
```

Como especificado pelo processador em uso, os barramentos de endereço e de dados são de 32-bits. Para completar o código acima, falta a declaração das constantes `CFG_DMEM_SIZE` e `CFG_DMEM_WIDTH`.

```
-- Memory parameters
constant CFG_DMEM_SIZE   : positive := 32;      -- Data memory bus size in 2LOG # elements
constant CFG_DMEM_WIDTH  : positive := 32;      -- Data memory width in bits
```

4.3 PLL

Um PLL (*Phase-Locked Loop*) é um sistema de controle que gera um sinal de saída cuja fase está relacionada com a fase do sinal de entrada. Entre as suas funções destacam-se: multiplicação e divisão de clock, deslocamento de fase e alteração da razão de ciclo de trabalho (*duty cycle*). Simplificadamente, o PLL é um ciruito de malha fechada contendo três componentes principais conforme são apresentados na figura 4.10 e podem ser descritos como:

- **Detector de Fase** fornece uma tensão de saída (tensão de erro) cuja componente contínua é proporcional a diferença de fase entre o sinal de entrada e o sinal do VCO.

- **Filtro Passa-Baixo** tem como função básica, eliminar a componente de alta frequência na saída do detector de fase e extrair somente a componente contínua que serve de tensão de controle para o VCO, ou seja, possui a função de integrador.

- **Oscilador Controlado por Tensão** ou *Voltage-Controlled Oscillator* (VCO) gera um sinal cuja frequência depende da tensão de controle.

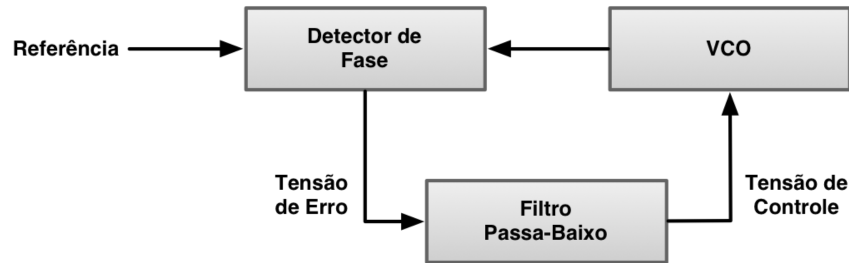


Figura 4.10: Diagrama de um PLL Básico

Os dispositivos da família Cyclone II fornecem uma rede interna para distribuição dos sinais de clock que pode fornecer o sinal de clock para todos os recursos do dispositivo, como os elementos de E/S, elementos lógicos, multiplicadores e blocos de memória interna. As características relacionadas ao sinal de clock deste dispositivo incluem também:

- Rede hierárquica de clock para obter desempenho até 402 MHz;

- Quatro (4) PLLs por dispositivo que permitem a realização de operações com o sinal de clock, como multiplicação e divisão, alteração de fase e ciclo de trabalho e a saída para alimentar um clock externo, permitindo o gerenciamento dos sinais de clock e o controle de sincronismo; e

- Permite até 16 linhas globais na rede de clock.

O PLL é o único módulo utilizado no CASoPC dependente da tecnologia utilizada, ou seja, adotando-se uma outra família de FPGA é necessário substituir o módulo do PLL em uso. Considerando que os ambientes de desenvolvimento disponibilizados pelos fabricantes de FPGA, incluem funções para facilitar a configuração e a inclusão de PLLs, este não torna-se, na prática, um fator complicador.

Para os dispositivos fabricados pela Altera e utilizando o seu software de desenvolvimento Quartus II, é possível utilizar a *megafunction* chamada **altpll** para habilitar os PLLs. A figura 4.11 apresenta as portas disponíveis nos PLLs dos dispositivos Cyclone II e na tabela 4.2 está a descrição de cada um desses sinais.

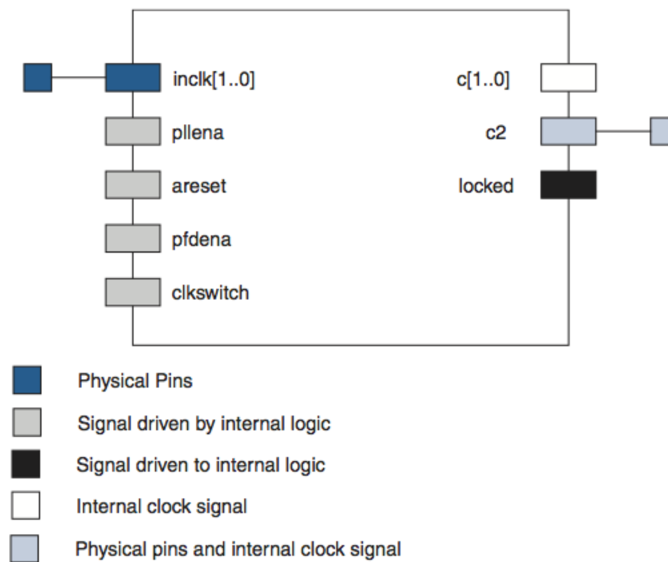


Figura 4.11: Sinais dos PLLs do Cyclone II

Porta	Descrição
`inclk[1..0]`	Entradas primária e secundária do clock
`pllena`	Ativo em nível alto, possui a função de habilitar o PLL
`areset`	Ativo em nível alto, reinicializa os contadores do PLL para seus valores iniciais
`pfdena`	Quando o PFD (*Phase Frequency Detector*) é desabilitado, o PLL deixa de monitorar as mudanças no clock de entrada. A saída do PLL continua a comutar na última frequência, porém, poderá ocorrer desvios na frequência
`clkswitch`	Realiza a comutação manual dos clocks de entrada
`c[2..0]`	Saída dos sinais de clock
`locked`	Indica o estado do PLL *lock*, ou seja, quando o clock de referência e o clock realimentado estão alinhados

Tabela 4.2: Descrição dos Sinais do PLL

Porém, nem todos esses sinais descritos são de uso obrigatório. A instanciação do módulo `altpll` no CASoPC é apresentada a seguir para ilustrar o uso do PLL em um dispositivo da Altera.

```
ALTPLL_INST: work.casopc_pll port map (
    areset      => rst_sys,
    inclk0      => clk_sys,
    c0          => clk_sdram,
    c1          => clk_ssram,
    c2          => clk_sdram_skewed,
    locked      => pll_locked
);
```

Como pode ser visto no código VHDL anterior, o CASoPC utiliza 4 sinais de clock: clock principal do sistema, clock para a memória SSRAM e dois clocks para

a memória SDRAM. A seguir é apresentada uma breve descrição desses sinais e a motivação para a configuração de cada sinal de clock será apresentada nas seções referentes a cada módulo.

`clk_sys`
: `clk_sys` é o sinal de clock externo que será utilizado como referência para os demais clocks internos. `clk_sys` possui frequência de 50 MHz e é utilizado em todo o sistema, exceto nos dispositivos que possui o seu clock gerado no PLL.

`clk_ssram`
: Sinal de clock utilizado na memória SSRAM e possui valor de 200 MHz.

`clk_sdram` e `clk_sdram_skewed`
: São sinais usados nas duas memórias SDRAM com frequência de 100 MHz, porém, o clock `clk_sdram_skewed` possui um atraso de 3 ns.

4.4 Mapeador de Memória

Os processadores podem utilizar as suas interfaces com as memórias para realizar a comunicação com dispositivos de entrada e saída (E/S), desta forma, o processador acessa um dispositivo de E/S utilizando os barramentos de endereço e de dados da mesma maneira que os utilizaria para acessar uma memória. Este método de comunicação com os dispositivos é chamado de *E/S mapeado em memória*, onde uma parte do espaço de endereçamento é dedicado aos dispositivos de E/S. Se para cada um destes dispositivos é atribuído um ou mais endereços, então, uma operação de escrita em um endereço específico envia um dado para o dispositivo. De forma análoga, uma operação de leitura recebe um dado do dispositivo.

Em um sistema com E/S mapeado em memória, uma operação de leitura ou escrita pode acessar tanto uma memória quanto um dispositivo de entrada e saída. A figura 4.12 apresenta um modelo básico de conexão entre o processador, uma memória e dois dispositivos de E/S mapeados em memória [30]. Um decodificador de endereços determina qual dispositivo se comunicará com o processador. É usado o barramento de endereços e o sinal `WE` (*write enable*) para gerar os sinais de controle para o restante do hardware. O decodificador de endereços, entre outros sinais, controla um multiplexador para habilitar a saída do elemento desejado.

Neste modelo, as mesmas instruções do processador usadas para acessar a memória podem também ser usadas para acessar os dispositivos. Para acomodar os dispositivos de entrada e saída, algumas áreas de endereços utilizadas pela CPU devem ser reservadas para E/S e não devem estar disponíveis para as memórias físicas.

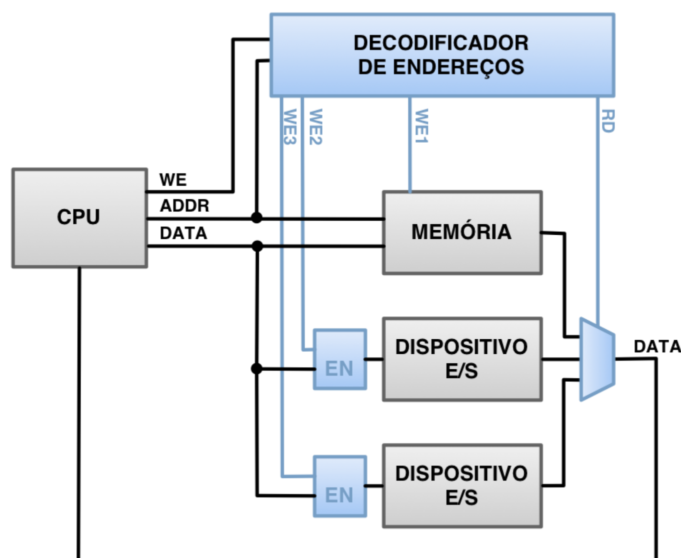


Figura 4.12: E/S Mapeado em Memória

Como vantagem, é possível citar que o processador não necessita de elementos lógicos extras para realizar o acesso aos dispositivos de E/S, resultado em processadores com menor área, mais rápidos e com consumo menor. Outra vantagem é que, como são usadas as mesmas instruções de acesso à memória para endereçar os dispositivos, todos os modos de endereçamento da CPU também estão disponíveis para E/S. Da mesma forma, as instruções que executam uma operação na ULA (Unidade Lógica Aritmética) com um operando na memória, também podem ser usadas com os dispositivos de E/S.

Em contrapartida, como os barramentos de endereço e dados são compartilhados e, normalmente, os dispositivos periféricos são muito mais lentos que a memória principal, as operações de entrada e saída podem diminuir o desempenho de todo o sistema.

Para contornar este problema, alguns processadores adotam barramentos dedicados para os dispositivos de entrada e saída, são as chamadas *Portas de E/S*. Contudo, este não é caso do processador MB-Lite e, na ausência de portas específicas para a conexão dos dispositivos, foi criado um *mapeador de memória* para possibilitar este acesso.

O processador MB-Lite possui barramento de endereços de 32-bits e, utilizando um esquema simples de mapeamento direto, há disponível 2^{32} endereços. Ou seja, uma quantidade muito superior ao que é realmente necessário pelo sistema para endereçar as memórias físicas e os dispositivos de E/S.

Como apresentado na seção 4.2.1 (Barramento Wishbone), todos os módulos *Slaves* do **CASoPC** implementam duas interfaces Wishbone (uma contendo os sinais de entrada e outra para os sinais de saída). Da mesma forma, o MB-Lite contém

duas interfaces Wishbone, porém do tipo *Master*. Então, o mapeador de memória para este sistema é bastante semelhante ao esquema tradicional porém, neste caso, será realizada a conexão entre as interfaces *Master* e as interfaces *Slave* do dispositivo que o processador deseja acessar. O mapeamento em memória dos dispositivos é realizado utilizando apenas um decodificador de endereços.

O código VHDL a seguir, apresenta a atribuição de um ID numérico a cada um dos dispositivos do sistema. Os endereços destes dispositivos são definidos pelo vetor `CFG_MEMORY_MAP`.

```
-------------------------------------------------------------------------------------------
-- BUS PARAMETERS & MEMORY MAPPER
-------------------------------------------------------------------------------------------
type memory_map_type is array(natural range <>) of
     std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);
constant CFG_NUM_SLAVES : positive := 14;

--
-- MAIN RAM (0000_0000 - 7FFF_FFFF) [2Gb]
-- DEVICES  (8000_0000 - FFFF_FFFF) [2Gb]
--
constant CFG_MEMORY_MAP : memory_map_type(0 to CFG_NUM_SLAVES) := (
    X"0000_0000",        -- ID  0: RAM         (0000_0000 - 00FF_FFFF) [16 Mb]
    X"0100_0000",        -- ID  1: NOT USED
    X"0200_0000",        -- ID  2: ROM         (0200_0000 - 02FF_FFFF) [16 Mb]
    X"0300_0000",        -- ID  3: NOT USED
    X"8001_0000",        -- ID  4: HEX7        (8001_0000 - 8001_FFFF)
    X"8002_0000",        -- ID  5: LCD         (8002_0000 - 8002_FFFF)
    X"8003_0000",        -- ID  6: SERIAL      (8003_0000 - 8003_FFFF)
    X"8004_0000",        -- ID  7: INTERRUPT   (8004_0000 - 8004_FFFF)
    X"8005_0000",        -- ID  8: TIMER       (8005_0000 - 8005_FFFF)
    X"8006_0000",        -- ID  9: BUTTONS     (8006_0000 - 8006_FFFF)
    X"8007_0000",        -- ID 10: LEDS        (8007_0000 - 8007_FFFF)
    X"8008_0000",        -- ID 11: NOT USED
    X"8010_0000",        -- ID 12: VIDEO       (8010_0000 - 803F_FFFF) [2Mb]
    X"8040_0000",        -- ID 13: NOT USED
    X"FFFF_FFFF"
);

constant MMAPPER_RAM_ID          : integer := 0;
constant MMAPPER_NOTUSE1         : integer := 1;
constant MMAPPER_ROM_ID          : integer := 2;
constant MMAPPER_NOTUSE2         : integer := 3;
constant MMAPPER_HEX7SEG_ID      : integer := 4;
constant MMAPPER_LCD16x2_ID      : integer := 5;
constant MMAPPER_UART_ID         : integer := 6;
constant MMAPPER_INTC_ID         : integer := 7;
constant MMAPPER_TIMER_ID        : integer := 8;
constant MMAPPER_BUTTONS_ID      : integer := 9;
constant MMAPPER_LED_ID          : integer := 10;
constant MMAPPER_NOTUSE3         : integer := 11;
constant MMAPPER_VIDEO_ID        : integer := 12;
constant MMAPPER_NOTUSE4         : integer := 13;
```

No próximo trecho de código é apresentada uma função do mapeador de memória onde um endereço encaminhado pelo processador (Wishbone *Master*) é compa-

rado com os endereços definidos em CFG_MEMORY_MAP (localmente chamado de WB_-MEMORY_MAP). Esta função retornará um vetor contendo apenas os valores "0" e "1", onde o único valor "1" indicará a posição relativa ao dispositivo que será acessado pelo processador.

```
------------------------------------------------------------------------------------------
-- Decodes the address based on the memory map.
-- Returns "1" if slave is attached.
------------------------------------------------------------------------------------------
function decode_address(adr : std_logic_vector) return std_logic_vector is
    variable result : std_logic_vector(WB_NUM_SLAVES - 1 downto 0);
begin
    if WB_NUM_SLAVES > 1 and notx(adr) then
        for i in WB_NUM_SLAVES - 1 downto 0 loop
            if (adr >= WB_MEMORY_MAP(i) and adr < WB_MEMORY_MAP(i+1)) then
                result(i) := '1';
            else
                result(i) := '0';
            end if;
        end loop;
    else
        result := (others => '1');
    end if;

    return result;

end function;
```

O código a seguir exemplifica como é realizada a conexão entre os barramentos *Slave* (entrada) e *Master* (saída). Neste código pode-se observar que os sinais we_-i (*write enable*), stb_i (*strobe*) e cyc_i (*cycle bus*) estarão habilitados apenas na interface que o processador deseja acessar. Este vetor, chamado ce_vector, é aquele retornado pela função decode_address apresentada anteriormente.

```
-- MASTER TO SLAVES
-- CE dos slaves
ce_vector <= decode_address(wb_master_o.adr_o);

CON: for i in WB_NUM_SLAVES - 1 downto 0 generate
begin
    wb_slave_i(i).clk_i <= clk_i;
    wb_slave_i(i).rst_i <= rst_cpu;
    wb_slave_i(i).adr_i <= wb_master_o.adr_o;
    wb_slave_i(i).dat_i <= wb_master_o.dat_o;
    wb_slave_i(i).we_i  <= wb_master_o.we_o and ce_vector(i);
    wb_slave_i(i).stb_i <= wb_master_o.stb_o and ce_vector(i);
    wb_slave_i(i).sel_i <= wb_master_o.sel_o;
    wb_slave_i(i).cyc_i <= wb_master_o.cyc_o and ce_vector(i);
end generate;
```

O próximo código ilustra como cada módulo pode ser incluído ou excluído facilmente do SoPC. A instanciação de cada módulo é condicional ao valor de cada constante listada neste trecho de código. Verifica-se também a existência de uma

constante para o uso da memória ROM em modo teste, desta forma, é possível acessar o conteúdo desta memória pelo barramento Wishbone. Em modo de operação normal, a ROM é conectada diretamente ao processador e não é acessível pelo barramento, desta forma, o seu espaço de endereços alocado no vetor `CFG_MEMORY_MAP` é ignorado pelo mapeador de memória.

```
------------------------------------------------------------------------------------------
-- DEVICES
------------------------------------------------------------------------------------------
    constant CASOPC_USE_HEX7         : boolean := true;
    constant CASOPC_USE_LCD16x2      : boolean := true;
    constant CASOPC_USE_UART         : boolean := true;
    constant CASOPC_USE_INTC         : boolean := true;
    constant CASOPC_USE_TIMER        : boolean := true;
    constant CASOPC_USE_BUTTONS      : boolean := true;
    constant CASOPC_USE_LED          : boolean := true;
    constant CASOPC_USE_VIDEO        : boolean := true;

    -- Uses a WB bus to read the ROM content. For testing purpose only
    -- and should not be enabled when the system is in production.
    constant CASOPC_USE_ROM_TESTING  : boolean := false;
```

4.5 Memória RAM

Este módulo implementa o controlador da memória que será utilizada pelo processador para armazenar dados. O funcionamento deste módulo é bastante simples e a complexidade resume-se ao controlador da memória SDRAM.

São utilizadas duas memórias SDRAM IS42S16160B [16] que possuem arquitetura interna de 4M x 16 x 4 *banks*, ou seja, possui 4 bancos de 4M por 16-bits, resultando em 16M por 16-bits. A figura 4.13 apresenta um diagrama funcional desta memória. Como o barramento de dados do sistema é de 32-bits e aproveitando a capacidade da memória em utilizar *burst*[2], são realizadas duas leituras consecutivas para obter os 32-bits requisitados pelo processador. Desta forma, na prática, obtém-se uma memória de 8M por 32-bits. Colocando o segundo módulo de memória na sequência de endereços, o sistema passa a perceber um total de 16M de memória RAM (*Random Access Memory*).

Como apresentado na seção 4.4, Mapeador de Memória, a faixa de endereço destinada à memória RAM é de 0x0000_0000 até 0x00FF_FFFF, ou seja, 16M de memória.

Como pode-se observar pelo diagrama da figura 4.13, o funcionamento deste tipo de memória é complexo, resultando em um código VHDL também complexo, sendo necessário o tratamento e controle de diversas temporizações, inclusive com a utilização de dois sinais de clock. O código VHDL para controlar as memórias SDRAM

[2]Termo genérico de computação que indica uma situação na qual um dispositivo transmite dados repetidamente sem verificar a entrada

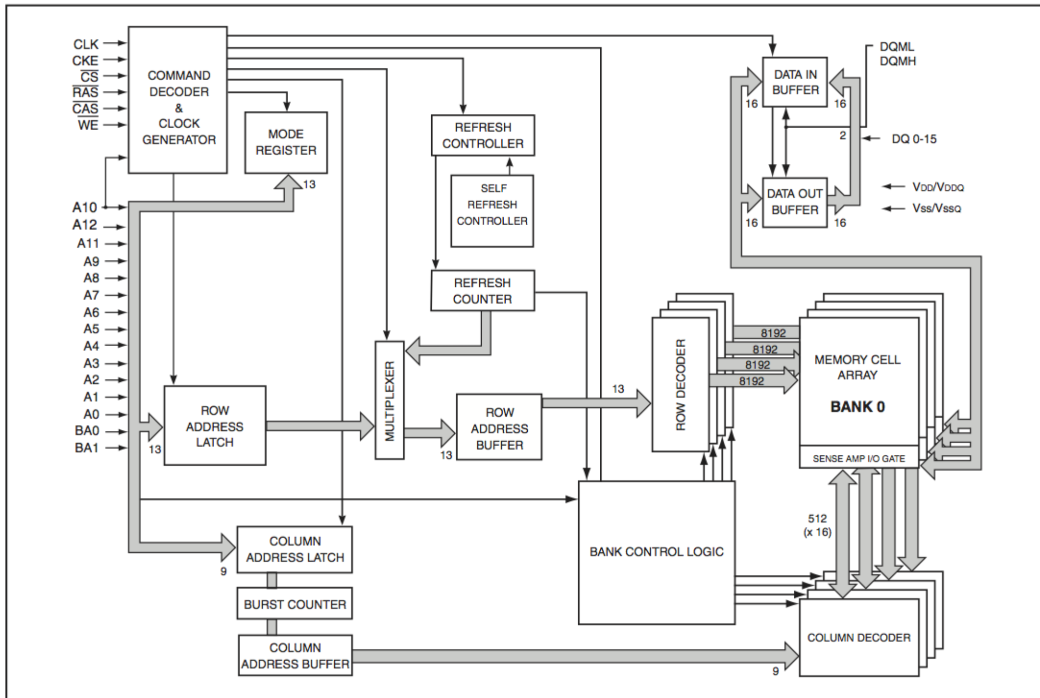


Figura 4.13: Diagrama Funcional da Memória SDRAM

IS42S16160B, disponível na plataforma de desenvolvimento DE2-70, encontra-se listado no Apêndice B para facilitar futuras referências.

O código deste módulo merece, futuramente, atenção para que sejam realizadas algumas otimizações. Mesmo a leitura da segunda metade do dado ser realizada em modo *burst*, esta metade estará disponível um ciclo de clock após a obtenção da primeira metade. Como são utilizados dois módulos de memória de 16-bits, é possível utilizar estas memórias em paralelo, ou seja, os primeiros 16-bits são acessados na primeira memória enquanto os outros 16-bits serão lidos da segunda. Reduzindo o tempo de acesso em 1 ciclo de clock.

As constantes para a configuração dos acessos à este módulo e para as suas interfaces estão definidos no trecho de código a seguir:

```
-------------------------------------------------------------------------------------------
-- SDRAM
-------------------------------------------------------------------------------------------
constant SDRAM_ADDR_WIDTH    : natural := 13;        -- 8M x 16
constant SDRAM_DATA_WIDTH    : natural := 16;

--
-- SDRAM TYPES
--
type sdram_io_type is record
    DQ      : std_logic_vector(SDRAM_DATA_WIDTH - 1 downto 0);
end record;

type sdram_out_type is record
    CLK     : std_logic;                                 -- Clock
    CKE     : std_logic;                                 -- Clock Enable
```

```
    RAS_n   : std_logic;                                    -- Row Address Strobe
    CAS_n   : std_logic;                                    -- Column Address Strobe
    WE_n    : std_logic;                                    -- Write Enable
    CS_n    : std_logic;                                    -- Chip Select
    BA_0    : std_logic;                                    -- Bank Address
    BA_1    : std_logic;                                    -- Bank Address
    ADDR    : std_logic_vector(SDRAM_ADDR_WIDTH-1 downto 0); -- SDRAM Address
    UDQM    : std_logic;                                    -- Data mask Upper Byte
    LDQM    : std_logic;                                    -- Data mask Lower Byte
end record;
```

4.6 Memória ROM

Este módulo implementa o controlador da memória a ser utilizada pelo processador como memória de instruções ou ROM (*Read-Only Memory*). De forma semelhante ao módulo de RAM, o funcionamento deste módulo é bastante simples, porém, devido a algumas limitações dos dispositivos de memória da plataforma de desenvolvimento, tornou-se necessária a realização de modificações no projeto.

O dispositivo de memória mais adequado para utilização como ROM seria uma memória *flash*, especificamente, a memória *Flash* S29GL064N [17] disponível na plataforma. Esta memória de 64 Mbits pode ser organizada para operar em dois modos: modo com 4M words (16-bits) ou 8Mbytes. Desta forma, o controlador poderia ser implementado para realizar duas leituras de 16-bits e então retornar a instrução de 32-bits para o processador.

Verificando-se as características AC no manual desta memória, percebe-se que o tempo de um ciclo de leitura possui duração de 90 ns. A leitura da segunda metade da instrução poderia ser realizada, usando o modo de acesso por páginas, no ciclo de clock seguinte, ou seja, 20 ns depois. Portanto, seria possível realizar a leitura de uma instrução completa em 110 ns. Porém, este tempo seria, pelo menos, 5 vezes inferior ao tempo necessário.

Conforme apresentado na arquitetura do processador MB-Lite, o PC (*program counter*) é incrementado a cada ciclo de clock e a interface entre o processador e a memória de instrução é bastante elementar conforme código apresentado a seguir, ou seja, não existe nenhum artifício para que o processador "espere" por uma nova instrução. Desta maneira, é necessário que o controlador de memória seja capaz de fornecer uma nova instrução a cada ciclo de clock, ou seja, a cada 20 ns.

```
-- Memory parameters
constant CFG_IMEM_SIZE    : positive := 14;      -- Instruction memory bus size (2LOG #)
constant CFG_IMEM_WIDTH   : positive := 32;      -- Instruction memory width

-- From instruction memory to processor
type imem_in_type is record
    dat_i : std_logic_vector(CFG_IMEM_WIDTH - 1 downto 0);
end record;
```

```
-- From processor to instruction memory
type imem_out_type is record
    adr_o : std_logic_vector(CFG_IMEM_SIZE - 1 downto 0);
    ena_o : std_logic;
end record;
```

Para tentar contornar esta limitação da memória, poderia ser utilizada a leitura por páginas disponível na *flash*. Este tipo de leitura possui um tempo de latência inicial grande, mas os acessos seguintes possuem um tempo de leitura menor, porém, este tempo ainda seria da ordem de 25 ns. Para realizar a leitura de uma instrução, são necessárias duas operações de leitura, portanto, o controlador utilizaria o artifício de incluir instruções NOP (*No Operation*) no *pipeline* do processador até que uma nova instrução estivesse disponível. De qualquer maneira, a solução incluiria perdas significativas no desempenho do sistema.

Os conectores de expansão, existentes na plataforma de desenvolvimento, não possuem conexões suficientes com o FPGA para permitir a utilização de um módulo externo de memória. Então, embora a memória SSRAM estivesse reservada para uso com o módulo de vídeo, a sua utilização como memória de instruções foi a melhor solução encontrada.

Na figura 4.14 é apresentado o diagrama da arquitetura do módulo de ROM.

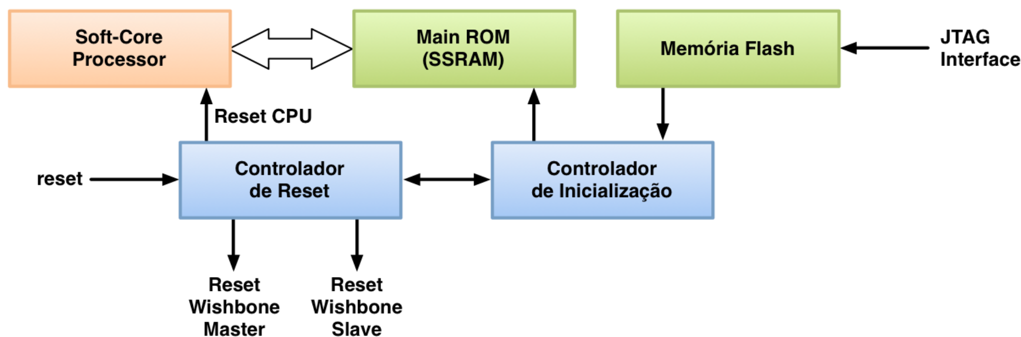


Figura 4.14: Diagrama dos Módulos da ROM

Como a memória SSRAM é uma memória volátil, torna-se necessário utilizar algum dispositivo para armazenar as instruções que serão copiadas para a SSRAM durante a inicialização do sistema. Portanto, foi implementado um controlador que carrega o software da memória *flash* e o transfere para a memória SSRAM que será acessada pelo processador como ROM. Este módulo também gerencia o circuito de reset criado para permitir a transferência do software antes da inicialização dos demais componentes.

O sistema ao ser inicializado ou reiniciado, manterá os sinais de reset do processador e dos demais dispositivos até a conclusão da transferência dos dados da memória *flash* para a memória SSRAM. Após a conclusão desta fase, a memória

SSRAM estará disponível como memória ROM para o processador e os sinais de reset são liberados para que o sistema opere normalmente.

4.7 Controlador de Interrupção

Uma interrupção é um evento assíncrono gerado pelo hardware (dispositivos periféricos) indicando a necessidade de atenção por parte do processador. Esta é uma forma de iniciar rotinas de software em resposta a eventos assíncronos do hardware. Estes eventos são sinalizados para o processador através de pedidos ou requisição de interrupção, do inglês, *Interrupt Request* (IRQ). O processamento da interrupção compõe uma troca de contexto da CPU para um rotina de software especificamente escrita para tratar a interrupção. Essa rotina é chamada *rotina de serviço de interrupção* ou *manipulador de interrupção* e os endereços destas rotinas são chamados de *vetores de interrupção*.

Existem dois tipos principais de interrupção gerada pelo hardware:

- **Interrupção Mascarável** (IRQ) pode ser configurada através de um bit em um determinado registrador para indicar que ela deve ser ignorada; e
- **Interrupção não-Mascarável** (NMI) é a interrupção que não pode ser ignorada ou desabilitada. As NMIs normalmente são utilizadas por *timers*, especialmente por *watchdog*.

O uso de interrupções é uma forma de evitar o desperdício de tempo computacional onde o software fica em *loop* aguardando que eventos sejam disparados pelos dispositivos. Desta forma, o processador fica disponível para realizar outros trabalhos até o momento em que uma interrupção avisa o processador que um evento ocorreu. Quando um sinal de interrupção é ativado, os seguintes passos serão realizados:

1. As interrupções são desabilitadas para evitar que o manipulador da interrupção seja atrapalhado por uma outra interrupção;
2. O estado atual da CPU é armazenado para que a execução normal do programa possa ser restaurada após o atendimento à interrupção;
3. O manipulador de interrupção é chamado para verificar qual foi a interrupção ocorrida, realizar as ações previstas e registrar o atendimento; e
4. Retornar à execução do programa restaurando o estado da CPU e habilitando novamente as interrupções.

O processador MB-Lite, assim como o próprio MicroBlaze, possui apenas uma linha de requisição de interrupção (uma linha de IRQ). Desta forma, faz-se necessária a criação de um elemento para controlar e multiplexar as requisições de interrupção. Este elemento é chamado de *Programmable Interrupt Controller* (PIC) ou, simplesmente, Controlador de Interrupção. O uso deste controlador para combinar várias fontes de interrupção em uma ou mais linhas de interrupção disponíveis na CPU facilita, também, que níveis de prioridade sejam atribuídos. Um diagrama básico de um PIC pode ser visualizado na figura 4.15. O estado da interrupção é acessado pela CPU utilizando o barramento.

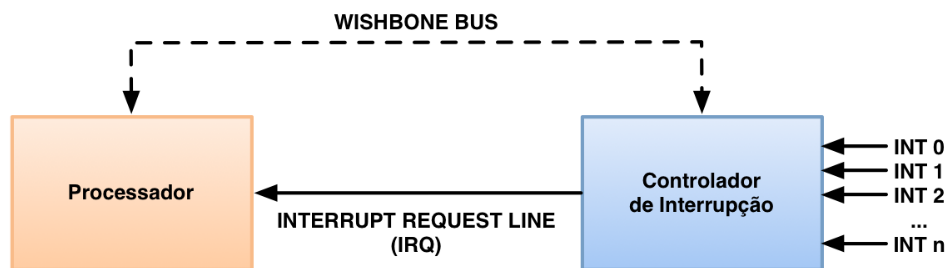


Figura 4.15: Controlador de Interrupção

Os registradores implementados no PIC do CASoPC adotam a mesma nomenclatura utilizada pela Xilinx nas documentações do MicroBlaze, são eles:

ISR (*Interrupt Service Register*)
: Registrador usado para indicar quais interrupções estão pendentes, ou seja, estão ativas mas ainda não foram atendidas.

IER (*Interrupt Enable Register*)
: Registrador usado para habilitar ou desabilitar determinadas interrupções.

IAR (*Interrupt Acknowledge Register*)
: Usado para indicar que uma determinada interrupção foi reconhecida. Com o reconhecimento de uma interrupção, o ISR também será modificado.

MER (*Master Enable Register*)
: Registrador de 1-bit utilizado para habilitar ou desabilitar as interrupções, ou seja, habilita ou não a saída de IRQ do controlador de interrupção (PIC).

A definição destes registradores foi incluída em um arquivo de cabeçalho, chamado `casopc.h`, para facilitar a programação do software usando as linguagens C ou C++. Os registradores poderão ser acessados para leitura ou escrita utilizando os nomes definidos nesse arquivo ou diretamente por meio dos seus endereços de memória. A definição dos registradores é apresentada na tabela 4.3.

Nome do Registrador	Endereço	Função	R/W
CASOPC_INTC_STATUS	0x8004_0000	ISR	R
CASOPC_INTC_ENABLE	0x8004_0004	IER	R/W
CASOPC_INTC_ACK	0x8004_0008	IAR	W
CASOPC_INTC_MASTER	0x8004_000C	MER	R/W

Tabela 4.3: Registradores do Controlador de Interrupção

Os registradores implementados no PIC, exceto o MER que possui tamanho de apenas um 1-bit, podem possuir tamanho de até 32-bits. Na prática, para contribuir com a economia de recursos do FPGA, o tamanho dos registradores é definido durante o processo de síntese do projeto de acordo com o número de fontes de interrupção existentes. Atualmente são 5 fontes de IRQ e a composição dos registradores é definida na tabela 4.4.

Bit #	Dispositivo
31 .. 5	Não usado
4	Chaves tipo *on/off*
3	Botões tipo *push-button*
2	*Touch Screen* do LTM
1	Timer 2
0	Timer 1

Tabela 4.4: Composição dos Registradores

Na ocorrência de uma interrupção gerada pelo pressionamento de um *push-button*, o registrador de status ISR possuirá o valor hexadecimal 0x08 (binário "00001000"). Após verificar o ISR, a rotina de manipulação de interrupção deverá consultar o controlador do respectivo periférico para conhecer qual foi o botão pressionado.

No MB-Lite, a ocorrência de uma interrupção é verificada no estágio *Instruction Decode* (ID). Neste estágio é verificado se a interrupção pode ser manipulada imediatamente ou se o seu atendimento deve ser atrasado. Se a interrupção pode ser atendida, uma instrução de salto é executada para o endereço 0x10 (vetor de interrupção). O endereço de retorno, ou seja, o endereço da próxima instrução que será executada após o atendimento da interrupção é registrada no endereço 0x14. Após o término da rotina de interrupção, a execução do programa continua normalmente.

4.8 UART

Uma UART (*Universal Asynchronous Receiver and Transmitter*) é um circuito utilizado para enviar dados paralelos através de um canal serial. UARTs normalmente são utilizadas em conjunto com o padrão RS-232 ou RS-485, que especificam as

características elétricas, mecânicas e funcionais dos equipamentos envolvidos na comunicação de dados [31]. Como os níveis de tensão definidos pelo RS-232 são diferentes daqueles existentes nas portas de entrada e saída do FPGA, faz-se necessária a utilização de um circuito para realizar esta conversão.

A plataforma de desenvolvimento adotada (DE2-70) inclui uma porta RS-232 com um conector padrão de 9 pinos e o componente para realizar a conversão de tensão e configurar os sinais de controle. Porém, esta porta está reservada para uso com o módulo de *debug* que será apresentado na seção 4.13. De qualquer forma, a implementação do *Tilt Test* exige, na sua forma mais básica, duas portas seriais para que a inclinação de duas plataformas possam ser comparadas. Da experiência e da metodologia adotada para o *Tilt Test Eletrônico*, são necessárias ao menos 4 portas seriais operando simultaneamente, podendo ser expandido para 8 portas simultâneas.

Conforme apresentado na seção 3.4, Plataforma de Desenvolvimento, existem dois conectores disponíveis para expansão contendo 72 pinos que estão conectados diretamente às portas de E/S do FPGA. Inicialmente, foram utilizados 8 destes pinos para conectar quatro interfaces seriais (outros pinos foram reservados para permitir, futuramente, expansão para mais 4 portas).

Estes 8 pinos, conectados diretamente ao FPGA, permitem a implementação dos sinais de *Tx* e *Rx*. Na figura 4.16 pode ser visualizado o diagrama do módulo RS-232 [32], o diagrama esquemático deste módulo consta no Apêndice A [33]. Embora os sinais de controle de fluxo estejam implementados nos conversores seriais, eles não são necessários pois os sensores adotados não utilizam este controle.

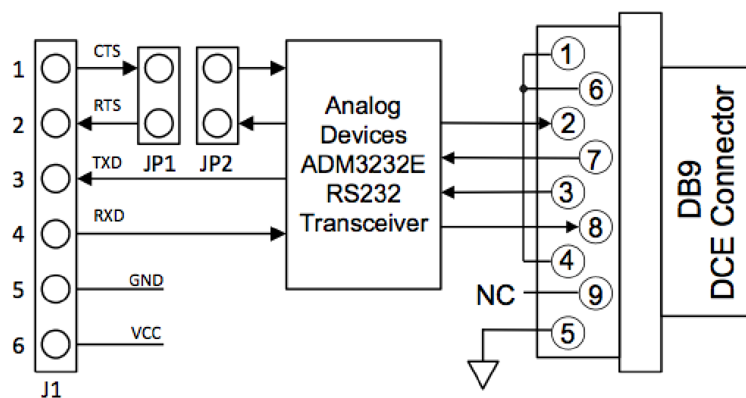


Figura 4.16: Diagrama do Módulo RS-232

Uma UART é composta basicamente por um transmissor e um receptor. O transmissor é essencialmente um *shift register* que recebe um dado paralelo e o transmite executando um deslocamento bit a bit em uma determina taxa de velocidade. A linha serial possui valor "1" quando está ociosa. A transmissão inicia com um *start bit* (bit de partida) cujo valor é "0", seguido pelos bits de dados e um bit opcional

de paridade. A transmissão de um byte termina com os chamados *stop bits* (bit de parada), que possuem valor “1”.

O número de bits de dados pode ser configurado para 6, 7 ou 8. O bit opcional de paridade é usado para detecção de erro. Para paridade *par*, o bit de paridade é definido como 0 quando os bits de dados possuem uma quantidade par de bits “1”. Para paridade *ímpar*, este bit é definido como 0 quando o número de bits “1” nos bits de dados é ímpar. O número de *stop bits* pode ser 1, 1.5 ou 2.

Uma transmissão com 8 bits de dados, sem paridade e com 1 bit de parada é exemplificada na figura 4.17. O bit LSB do dado é transmitido primeiro.

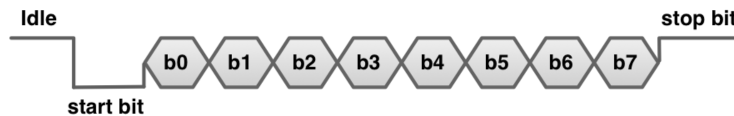


Figura 4.17: Transmissão de um Byte

Como nenhuma informação de clock ou sincronismo é transmitida pela conexão serial, o transmissor e receptor devem possuir a mesma configuração, como *baud rate* (número de bits por segundo), número de bits de dados e de parada e o uso ou não do bit de paridade.

4.8.1 Módulo de Recepção

A recepção serial utiliza um método chamado de *oversampling* para estimar os pontos centrais dos bits transmitidos e, desta forma, fazer a amostragem nestes pontos. As comunicações seriais, comumente, utilizam uma taxa de amostragem de 16 vezes o valor do *baud rate*, o que significa que cada bit será amostrado 16 vezes. A seguir são enumerados os passos do fluxo de recepção usando este esquema de sobreamostragem.

1. O receptor aguarda em modo de espera até a chegada de um bit “0”, indicando um bit de partida (*start bit*), e inicia o contador de amostragem.

2. Quando o contador alcançar o valor “7”, indicando o ponto central do bit de partida, o contador é zerado e a contagem é reiniciada.

3. Quando o contador alcançar o valor “15”, indicando, agora, o ponto central do primeiro bit de dado, o valor deste bit é lido e inserido no *shift register*. O contador é reiniciado.

4. O passo (3) é repetido até que todos os bits de dados sejam lidos (6, 7 ou 8).

5. Se o bit de paridade é utilizado, o passo (3) é repetido mais uma vez para realizar a leitura deste bit.

6. Repete o passo (3) para ler os bits de parada.

Este fluxo pode ser melhor visualizado pelo diagrama da figura 4.18.

O esquema de *oversampling* basicamente executa a função de um sinal de clock. Ao invés de utilizar a borda de subida para indicar que o sinal de entrada é válido, é utilizado um contador para estimar o ponto central de cada bit. Como o receptor não possui informação sobre o momento exato do início da transmissão, esta estimativa oferece uma forma de detectá-la no tempo de $\frac{1}{16}$ da taxa de *baud rate* utilizada.

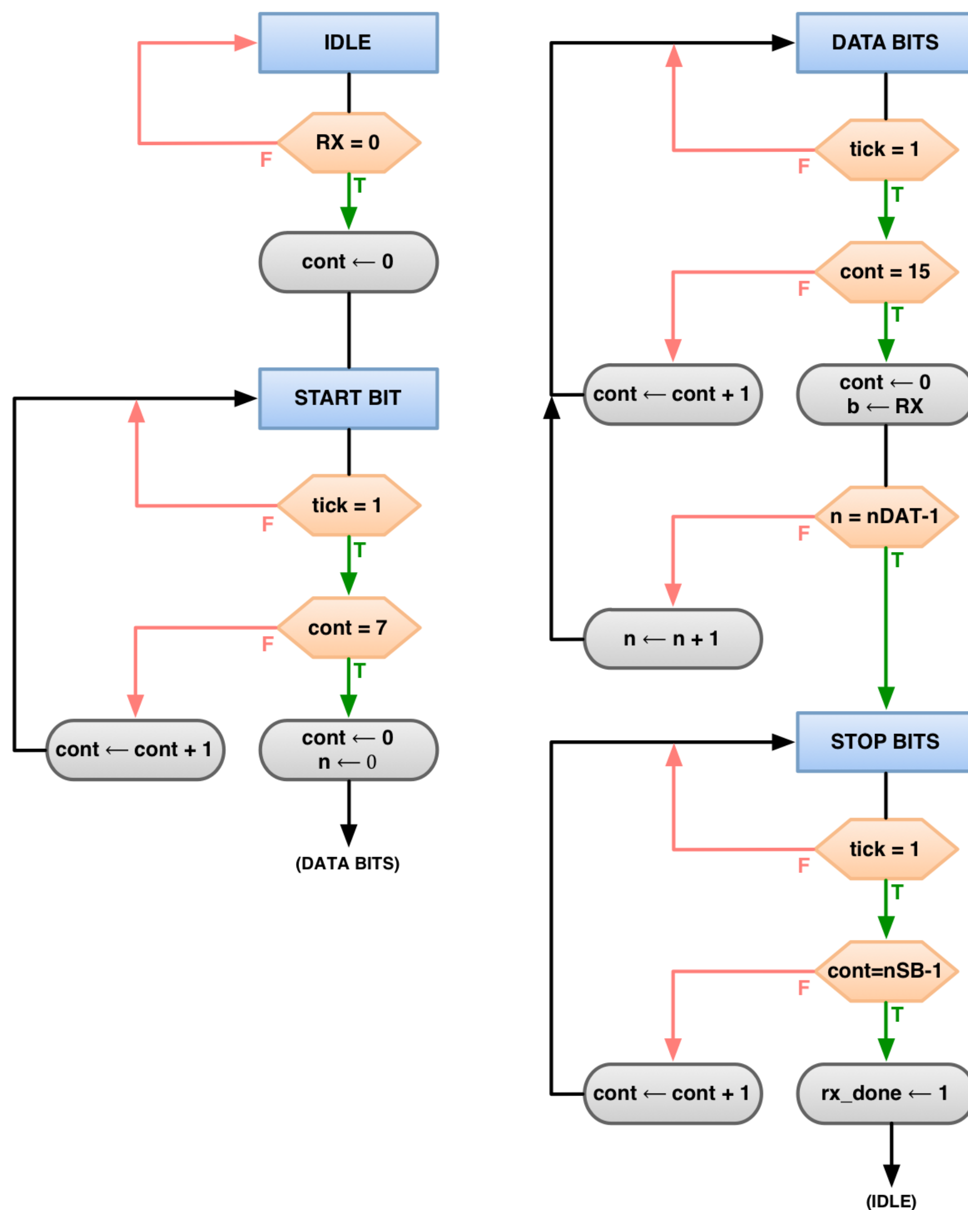


Figura 4.18: Receptor UART

4.8.2 Gerador de *Baud Rate*

O gerador de *baud rate* gera um sinal de amostragem cuja frequência é exatamente 16 vezes o valor do *baud rate*. Para evitar a criação de um novo clock, o sinal de amostragem deve funcionar como um gerador de pulso para o receptor.

Para uma taxa de 19200 bps, por exemplo, a taxa de amostragem deve ser de $19200 * 16 = 307200$. Como o clock do sistema é de 50 MHz, o gerador de *baud rate* necessita de um contador mod-163 ($\frac{50*10^6}{307200} = 163$), ou seja, um pulso será gerado a cada 163 ciclos de clock.

No trecho de código a seguir, pode ser verificada a definição das contantes utilizadas na geração do *baud rate*. Onde `CLK_FREQ = 50`.

```
-----------------------------------------------------------------------------------------
-- CONSTANTS
-----------------------------------------------------------------------------------------
constant BAUD_1200   : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / (  1200 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_2400   : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / (  2400 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_4800   : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / (  4800 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_9600   : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / (  9600 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_19200  : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / ( 19200 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_38400  : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / ( 38400 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_57600  : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / ( 57600 * 16)),
               BAUD_DIVIDER_SIZE));
constant BAUD_115200 : std_logic_vector(BAUD_DIVIDER_SIZE - 1 downto 0) :=
               std_logic_vector(to_unsigned(integer(CLK_FREQ * 1_000_000 / (115200 * 16)),
               BAUD_DIVIDER_SIZE));
```

4.8.3 Módulo de Transmissão

O módulo de transmissão é essencialmente um *shift register* que desloca os bits para a saída em uma taxa de transmissão específica. Esta taxa pode ser controlada utilizando um pulso oriundo do gerador de *baud rate*. Como a técnica de *oversampling* não é necessária para a transmissão, a frequência dos pulsos será 16 vezes menor do que do módulo de recepção. Para evitar a criação de um novo módulo, o transmissor compartilha o mesmo gerador de *baud rate*, porém, utilizando o contador interno para gerar um bit a cada 16 pulsos.

O diagrama do módulo de transmissão é semelhante ao do módulo de recepção (figura 4.18). Após a transmissão ser habilitada, a máquina de estados carrega o byte de dado e, então, gradualmente progride através dos estados *start bit*, *data bits* e *stop bits* para transmitir os bits correspondentes.

4.8.4 Módulo Completo de UART

O módulo de UART, além dos módulos de transmissão, recepção e gerador de *baud rate*, também é composto por um controle de interrupção e dois FIFOs, um para transmissão e outro para recepção.

O FIFO de recepção garante um espaço de *buffer* e reduz a chance de que dados recebidos sejam sobrescritos antes que o processador execute a sua leitura. Já o FIFO de transmissão tenta aproveitar a largura do barramento de dados de 32-bits para diminuir as operações de escritas realizadas pelo processador.

No diagrama da figura 4.19 está representado o módulo UART implementado no **CASoPC**. A escrita dos dados a serem transmitidos e a leitura dos dados recebidos são realizados através dos FIFOs. As configurações da porta e de interrupção são realizadas pelos demais registradores.

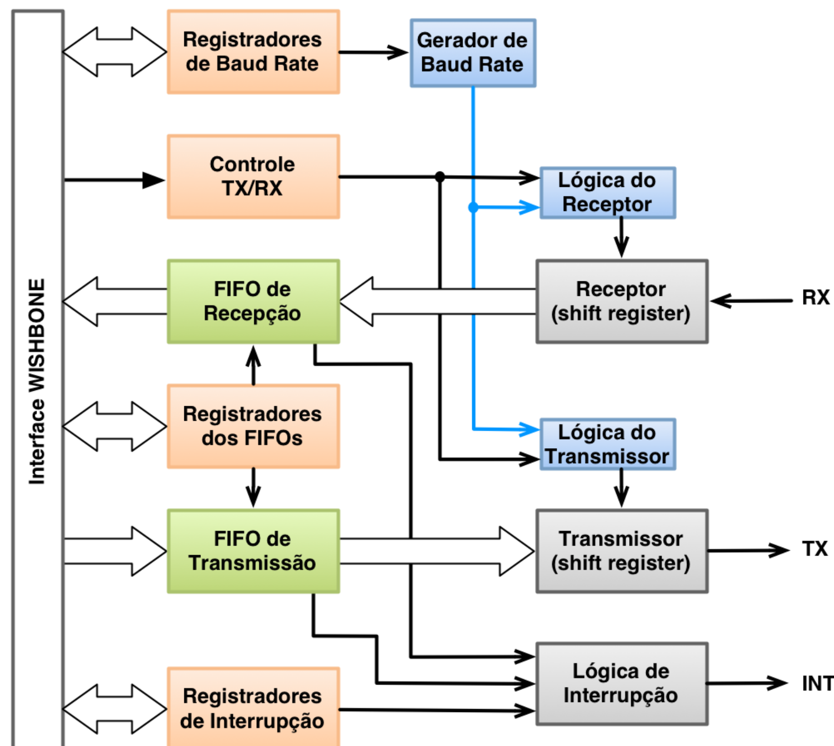


Figura 4.19: Diagrama em Blocos do Módulo UART

O trecho de código a seguir configura, em tempo de síntese do projeto, o número total de módulos UARTs que devem ser instanciados. O tamanho dos FIFOs, em

bytes, é especificado como $2^{FIFO_TX_SIZE}$ e $2^{FIFO_RX_SIZE}$, ou seja, pela configuração atual, ambos os FIFOs possuem tamanho de 32 bytes.

A constante BAUD_DIVIDER_SIZE define o número de bits necessários para armazenar as constantes de *baud rate* apresentadas no código anterior. A maior constante, BAUD_1200, possui o valor calculado como $\frac{50*1000000}{1200*16} = 2604$, ocupa 12 bits.

```
----------------------------------------------------------------------------------------
-- SERIAL PORT
----------------------------------------------------------------------------------------
constant UART_NUM_PORTS      : positive := 4;        -- Number of serial ports
                                                      -- (except onboard port - for debug)

-- FIFO #BYTES = 2 ^ SIZE
constant FIFO_TX_SIZE        : positive := 5;        -- Transmitter FIFO size
constant FIFO_RX_SIZE        : positive := 5;        -- Receiver FIFO size

constant BAUD_DIVIDER_SIZE   : positive := 12;       -- # bits of BAUD constants
```

O código VHDL a seguir pertence à entidade principal do projeto, onde os módulos UART são instanciados. Para endereçar as diversas portas seriais, foi elaborado um esquema de mapeamento semelhante ao apresentado na seção 4.4, Mapeador de Memória.

Esta forma de definir e instanciar os módulos seriais facilita a configuração e manutenção do sistema, porém, em contra-partida, insere atraso de um ciclo para a decodificação do endereço da porta serial solicitada pelo processador.

```
----------------------------------------------------------------------------------------
-- UART
----------------------------------------------------------------------------------------
GEN_UART: if (CASOPC_USE_UART = true) generate
begin
    GEN_NODE: for i in UART_NUM_PORTS downto 1 generate
    begin
        UART: uart_wb
        port map (
            wb_o        => uart_slave_o(i),
            wb_i        => wb_slave_i(MMAPPER_UART_ID),
            ce          => ce_vector(i),
            uart_tx     => uart_o_hw(i).txd,
            uart_rx     => uart_i_hw(i).rxd,
            uart_int_o  => int_uart_s
        );
    end generate;
end generate;


----------------------------------------------------------------------------------------
-- UART WB OUTPUT AND CHIP ENABLE SELECT ("MAPPER")
----------------------------------------------------------------------------------------
WB_UART_MAPPER: wb_uart_submapper generic map (
    UART_COUNT    => UART_NUM_PORTS
)
port map (
    wb_o          => wb_slave_o(MMAPPER_UART_ID),
```

```
        wb_i          => wb_slave_i(MMAPPER_UART_ID),
        ce            => ce_vector,
        uart_slave_i  => uart_slave_o
    );
```

As funções definidas para os módulos seriais estão relacionadas na tabela 4.5, seguida por uma breve descrição destas funções e registradores. O índice n indica o número da porta serial, onde, $n > 0$. Por exemplo, se o sistema é sintetizado com 4 portas seriais, a primeira porta será a de número "1", desta forma, para escrever um byte na porta serial #1, pode-se utilizar o registrador de nome `CASOPC_UART1_TX` ou, ainda, escrever diretamente no endereço 0x8003_1000.

Para atender melhor a aplicação do *Tilt Test*, onde a diferença entre os tempos de leitura dos sensores é fundamental, foi definido no decodificador de endereços das portas seriais um endereçamento para permitir a escrita em todas as portas ao mesmo tempo. Este endereço é o 0x8003_0000, porém, é válido apenas para as operações de escrita. Desta forma, para transmitir um byte por todas as portas seriais simultaneamente, pode-se escrever no endereço 0x8003_0000 ou utilizar o registrador `CASOPC_UART0_TX`. Ressalta-se, novamente, que esta é a única função atendida no índice "0", as operações de leitura, verificação de estado ou configuração deverão ser realizadas individualmente em cada porta.

`CASOPC_UART`n`_RX`

Lê um dado (byte) do FIFO de recepção.

`CASOPC_UART`n`_TX`

Escreve um dado (byte ou dword[3]) no FIFO de transmissão.

`CASOPC_UART`n`_CFG_UART`

Configura a porta serial. O formato do dado é:

31..8	7..3	2..0
Não usado	Reservado	BAUD RATE

Bits reservados: (*implementação futura*)

7..6 - tamanho do dado (6, 7 ou 8 bits)

5 - número de bits de parada (1 ou 2 bits)

4..3 - paridade par, ímpar ou nenhuma

Baud Rates suportados:

0 - 1200 bps

1 - 2400 bps

[3]dword é uma abreviação comum para *double word* e equivale a 32-bits.

Nome do Registrador	Endereço	Função	R/W
`CASOPC_UART0_TX`	0x8003_0000	Transmite um dado em todas as portas seriais	W
CASOPC_UARTn_RX	0x8003_n000	Recebe um dado na porta serial n	R
CASOPC_UARTn_TX	0x8003_n000	Transmite um dado na porta serial n	W
CASOPC_UARTn_CFG_UART	0x8003_n004	Configuração da porta serial n	R/W
CASOPC_UARTn_CFG_INT	0x8003_n008	Configuração de interrupção	R/W
CASOPC_UARTn_CFG_WB	0x8003_n00C	Configura a transferência de dados (byte ou *dword*)	W
CASOPC_UARTn_RESET	0x8003_n010	Reset dos FIFOs	W
CASOPC_UARTn_STATUS_RX	0x8003_n014	Verifica o uso do FIFO de recepção	R
CASOPC_UARTn_STATUS_TX	0x8003_n018	Verifica o uso do FIFO de transmissão	R
CASOPC_UARTn_STATUS_INT	0x8003_n01C	Verifica as interrupções	R

Tabela 4.5: Registradores da UART

2 - 4800 bps
3 - 9600 bps
4 - 19200 bps
5 - 38400 bps
6 - 57600 bps
7 - 115200 bps

CASOPC_UARTn_CFG_INT

Habilita e configura as interrupções.

31..8	7..5	4..2	1	0
Não usado	RX TRIGGER	Reservado	RX ENA	TX ENA

RX TRIGGER: gera uma interrupção quando existirem x bytes no FIFO de recepção (esta fonte de interrupção será ignorada se o bit `RX_ENA` estiver desabilitado). Os valores atualmente suportados são:

0 - desabilitado
1 - 1 byte
2 - 4 bytes
3 - 16 bytes

4 - 24 bytes
5 - 32 bytes

RX ENA: habilita/desabilita as interrupções para recepção
0 - desabilita
1 - habilita

TX ENA: habilita/desabilita as interrupções para transmissão
0 - desabilita
1 - habilita

`CASOPC_UART`n`_CFG_WB`

Configura a transferência de dados para o FIFO de transmissão (0=byte, 1=dword)

`CASOPC_UART`n`_RESET`

Esta função é usada para limpar os FIFOs de transmissão e/ou recepção. Esta função não é registrada e o bit 0 corresponde ao FIFO de transmissão e o bit 1 corresponde ao FIFO de recepção. Definindo o valor "1" em qualquer um destes bits, limpa o FIFO correspondente.

`CASOPC_UART`n`_STATUS_RX` e `CASOPC_UART`n`_STATUS_TX`

Estas funções retornam a quantidade de bytes armazenadas nos FIFOs de recepção e transmissão.

`CASOPC_UART`n`_STATUS_INT`

Este registrador armazena a origem da interrupção gerada.

31..4	3	2	1	0
Não usado	FF_RX_TR	FF_RX_OF	TX_TO	FF_TX_OF

FF_RX_TR	- Tamanho do FIFO de recepção alcançou o número de bytes definido em `RX_TRIGGER`
FF_RX_OF	- *Overflow* do FIFO de recepção
TX_TO	- *Timeout* do FIFO de transmissão
FF_TX_OF	- *Overflow* do FIFO de transmissão

4.9 Controlador de Vídeo

Um controlador de vídeo é responsável em gerar sinais de sincronismo e os pixels na forma serial que serão encaminhados a um monitor. A figura 4.20 apresenta um diagrama simplificado de um controlador de vídeo contendo um circuito de sincronização e um circuito de geração de pixel.

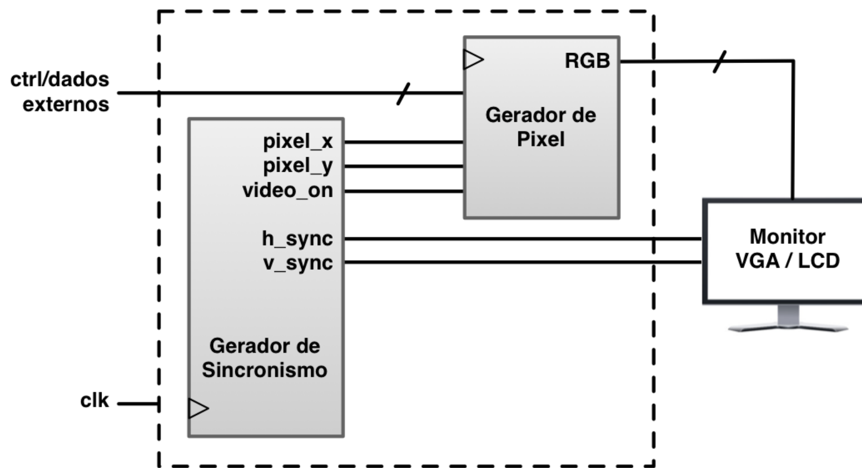


Figura 4.20: Diagrama Simplificado de um Controlador de Vídeo

O gerador de sincronismo transmite os sinais de temporização e de sincronismo. Os sinais *hsync* e *vsync* são conectados diretamente ao monitor padrão VGA ou a um módulo de LCD para controlar as varreduras horizontal e vertical. Os dois sinais são decodificados por contadores internos para gerar os sinais *pixel_x* e *pixel_y*. Estes sinais indicam a posição relativa da varredura e especificam a posição do ponto atual, estes sinais também são utilizados para gerar o sinal *video_on* que indica se os pontos (*pixels*) encontram-se em uma área visível ou não do vídeo.

O circuito de gerador de pixel implementa as três componentes de cores, coletivamente chamada de RGB (*Red, Green and Blue*) para cada ponto. Um valor de cor é obtido de acordo com as coordenadas do pixel (*pixel_x* e *pixel_y*) e os sinais de controle e dados externos.

4.9.1 Circuito Gerador de Sincronismo

O circuito de sincronização gera o sinal *hsync* que especifica o tempo necessário para percorrer uma linha e o sinal *vsync* que especifica o tempo necessário para percorrer uma tela inteira. A tela de um monitor, seja ele CRT ou LCD, inclui uma pequena borda preta, como apresentado na figura 4.21 [31]. A área central é a área visível ou quadro. As coordenadas do eixo vertical aumentam de cima para baixo. Considerando a resolução de 800x480 do LCD adotado no projeto, as coordenadas da borda superior esquerda e da borda inferior direita são (0,0) e (799,479), respectivamente.

Ainda com relação ao mesmo LCD, o período do sinal *hsync* possui 1056 pixels e pode ser dividido em quatro regiões:

- *Área Visível* ou *Quadro*: região onde os pixels são desenhados na tela. O comprimento desta região é de 800 pixels.

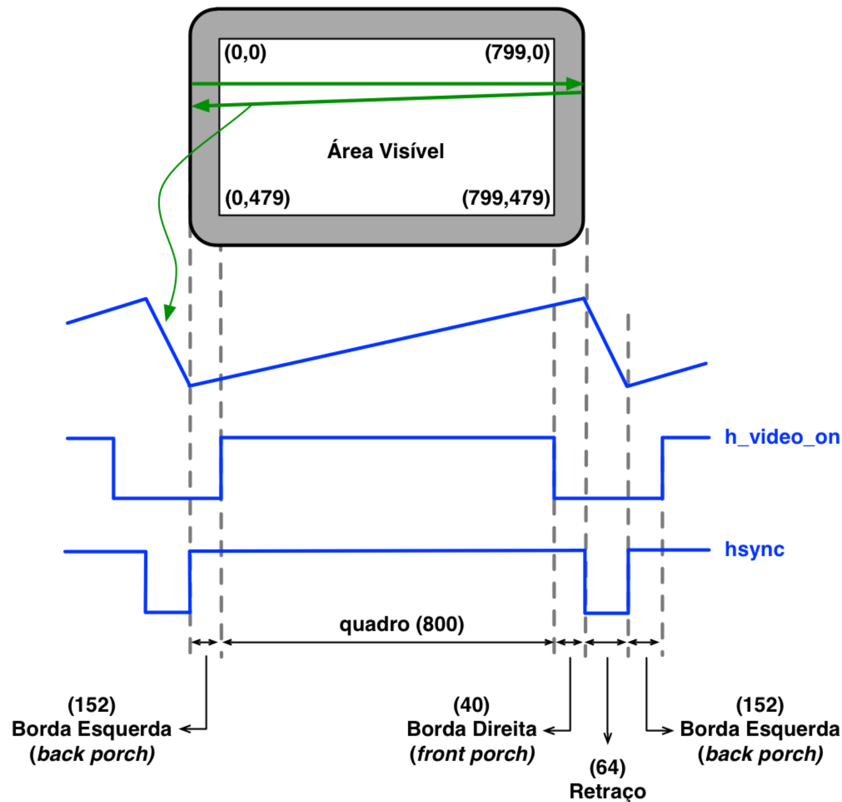


Figura 4.21: Diagrama de Temporização da Varredura Horizontal

- *Retraço*: região na qual o feixe[4] retorna para a margem esquerda. O sinal de vídeo é desabilitado e esta região possui 64 pixels.

- *Borda Direita*: região formada à direita do quadro. Esta região também é conhecida como *front porch*. O sinal de vídeo é desabilitado e o comprimento desta área é de 40 pixels.

- *Borda Esquerda*: região à esquerda do quadro, também conhecida como *back porch*. O vídeo é desabilitado e o seu comprimento é de 152 pixels.

O sinal *hsync* pode ser obtido utilizando um contador mod-1056. O sinal *display_on* estará definido apenas na região de área visível, ou seja, entre os pixels 216 (retraço + *back porch*) e 1016. Entre 1016 e 1056 encontra-se a região de *front porch*. O sinal *pixel_x* também só será válido se *display_on* = 1.

Os valores apresentados acima correspondem ao módulo LCD utilizado [18]. Os valores para resolução de 640x480, comumente chamado de *modo VGA* estão relacionados na tabela 4.6 juntamente com os valores do modo SVGA (Super VGA) de 800x600 pixels.

[4]Embora o texto refira-se a um módulo LCD, é comum verificar a utilização dos mesmos termos empregados nos antigos CRT. Porém, as regiões, ao invés de serem especificados em tempo, são especificadas em pontos (*pixels*).

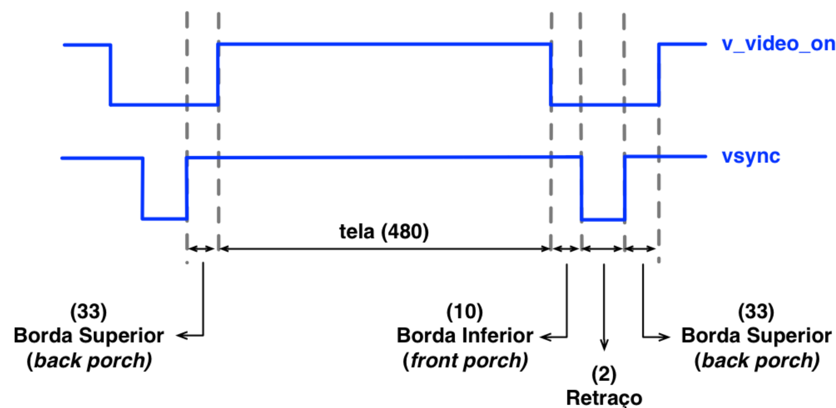


Figura 4.22: Diagrama de Temporização da Varredura Vertical

Parâmetro	**Resolução VGA (pixel)**	**Resolução SVGA (pixel)**
Quadro (horizontal)	640	800
Retraço horizontal	96	120
Front Porch horizontal	16	64
Back Porch horizontal	48	56
Quadro (vertical)	480	600
Retraço vertical	2	6
Front Porch vertical	10	23
Back Porch vertical	33	37

Tabela 4.6: Temporização dos Modos VGA e SVGA

Durante a varredura vertical, a tela é desenhada de cima para baixo. O formato do sinal *vsync* é semelhante ao *hsync* como pode ser verificado na figura 4.22 [31]. A unidade de tempo do movimento de desenho da tela é representado em termos de linhas horizontais. Um período do sinal *hsync* possui 525 linhas e pode ser divido em quatro regiões:

- *Área Visível* ou *Quadro*: região onde as linhas horizontais são exibidas na tela. O comprimento desta região é de 480 linhas.
- *Retraço*: região que o feixe retorna para o topo da tela. O sinal de vídeo é desabilitado e o comprimento desta região é de 2 linhas.
- *Borda Inferior*: região formada ao término do quadro, também conhecida como *front porch*. O sinal de vídeo é desabilitado e o seu comprimento é de 10 linhas.
- *Borda Superior*: também conhecida como *back porch*, é a região formada acima do quadro. O vídeo é desabilitado e possui comprimento de 33 linhas.

O sinal *vsync* pode ser obtido utilizando um contador mod-525. O sinal *display_-on* estará definido, conjuntamente com o sinal *hsync*, entre as linhas 35 (retraço + *back porch*) e 515. Entre as linhas 515 e 525 encontra-se a região de *front porch.* O sinal *pixel_y* somente será válido se *display_on* = 1.

A obtenção do clock necessário ao módulo de vídeo pode ser conseguido utilizando a seguinte fórmula:

$$pixel\ rate = p * l * fps$$

onde, p é o número de pixels horizontais e l representa o número total de linhas.

O parâmetro *fps* especifica quantos quadros devem ser desenhados a cada segundo, do inglês, *frames per second*. Para o olho humano, esta taxa de *refresh* deve ser de pelo menos 30 fps para que os movimentos pareçam contínuos. Para reduzir a oscilação, normalmente, os monitores utilizam taxas mais altas, como 60 quadros por segundo. Então, a taxa de pixels pode ser calculada como:

$$pixel\ rate = 1056 * 525 * 60$$
$$pixel\ rate \approx 33.2\text{M pixels/segundo}$$

Desta forma, para que os contadores gerem os pulsos de sincronismo no tempo correto e seja possível desenhar 60 quadros por segundo, o clock do módulo de vídeo deverá ser de 33.2 MHz.

4.9.2 Circuito Gerador de Pixel

O circuito gerador de pixel é responsável por gerar, no caso do **CASoPC**, um sinal RGB de 24-bits, sendo 8-bits para cada componente de cor. Os sinais de controle e dados externos especificam o conteúdo da tela e os sinais *pixel_x* e *pixel_y*, originados no gerador de sincronismo, fornecem as coordenadas do ponto atual.

Existem três tipo comuns de circuitos geradores de pixel, são eles:

- *Mapeamento de Bits*;
- *Mapeamento de Tiles*; e
- *Mapeamento de Objetos.*

No esquema de mapeamento de bits, uma memória de vídeo é usada para armazenar os dados que serão exibidos no vídeo. Cada pixel é mapeado diretamente para uma posição na memória e os sinais *pixel_x* e *pixel_y* são usados para formar o endereço. O processador gráfico, continuamente, atualiza a tela e pode escrever dados relevantes na memória. Um outro circuito carrega os dados da memória, compondo o sinal RGB, para exibição no vídeo.

Este é um dos esquemas de mapeamento utilizado no **CASoPC** e é disponibilizado como um modo de vídeo (modo gráfico).

Para a resolução de 800x480, existem 384000 pixels na tela. Embora, cada pixel seja representado por 24-bits, para evitar atrasos na solução de desalinhamentos na memória, são utilizados 32-bits para cada pixel. Isto resulta em 1536000 bytes, ou seja, é necessário uma memória com pelo menos 1,5 Mbyte para armazenar uma tela.

É comum em processadores de vídeo, a existência de mais de uma página de vídeo. Assim, enquanto o conteúdo de uma página é exibido no vídeo, as outras páginas estão disponíveis para escrita. Este método necessita de um circuito chaveador adicional para alterar a página que será exibida. Porém, pelo modelo apresentado, seria necessário 1,5Mb para cada página de vídeo adicional. Como, neste projeto, a memória utilizada possui 2Mb, apenas uma página de vídeo é implementada.

Para reduzir os requisitos de memória, é possível utilizar o esquema de mapeamento de *tiles*. Neste esquema, uma coleção de bits são agrupados para formar um *tile*[5] que será tratado como uma unidade de dado. Por exemplo, definindo um quadrado de 8x8 pixels (64 pixels) como um *tile*, uma tela de 800x480 orientada a pixels poderia ser trabalhada como uma tela de 100x60 *tiles*. Neste caso, seriam necessárias apenas 6000 posições de memória, ao contrário das 384000 do modo de mapeamento de bits.

Para definir a quantidade de bits necessários para endereçar todas estas "posições de memória", é necessário conhecer o total de padrões de *tiles*. Por exemplo, existindo 256 padrões distintos de *tiles* são necessários 8-bits para endereçar todos os padrões. Este modo de mapeamento normalmente necessita de uma ROM para armazenar os padrões.

O mapeamento de *tiles* é o segundo tipo de circuito gerador de pixel utilizado no **CASoPC**. Este esquema é utilizado pelo controlador de vídeo para exibir os modos de texto.

4.9.3 Circuito Gerador de Caracteres

Um método para construir um gerador de caracteres é tratar os caracteres como *tiles* e implementar um esquema de mapeamento de *tiles*. Em um esquema de mapeamento de bits, o valor de um pixel é representado, por exemplo, por 24-bits, enquanto o valor de um *tile* é representado por um código específico, como um código ASCII.

[5]Embora não seja usual, poderia-se traduzir *tile* como ladrilho ou azulejo, pois é a ideia que o emprego da palavra quer sugerir.

Os padrões dos *tiles* correspondem às fontes de um conjunto de caracteres. Uma variedade de fontes podem ser utilizadas e, nesta implementação, foi utilizada uma fonte 8x16, ou seja, cada caracter é representado por um padrão de 8x16 pixels. O conjunto de caracteres é armazenado em uma ROM e, atualmente, estão implementados 128 caracteres. Então, para armazenar este conjunto, são necessários 2048 bytes ou ainda $8 * 16 * 128 = 16384$ bits. O trecho de código a seguir exemplifica a definição dos caracteres "A" e "B" na ROM.

```
-- ROM definition
constant ROM : rom_type := (
(...)
-- code x41
"00000000", -- 0
"00000000", -- 1
"00010000", -- 2    *
"00111000", -- 3   ***
"01101100", -- 4  ** **
"11000110", -- 5 **   **
"11000110", -- 6 **   **
"11111110", -- 7 *******
"11000110", -- 8 **   **
"11000110", -- 9 **   **
"11000110", -- a **   **
"11000110", -- b **   **
"00000000", -- c
"00000000", -- d
"00000000", -- e
"00000000", -- f
-- code x42
"00000000", -- 0
"00000000", -- 1
"11111100", -- 2 ******
"01100110", -- 3  **  **
"01100110", -- 4  **  **
"01100110", -- 5  **  **
"01111100", -- 6  *****
"01100110", -- 7  **  **
"01100110", -- 8  **  **
"01100110", -- 9  **  **
"01100110", -- a  **  **
"11111100", -- b ******
"00000000", -- c
"00000000", -- d
"00000000", -- e
"00000000", -- f
```

Utilizando estes caracteres de 8x16 em um vídeo de 800x480, obtém-se uma tela com 100x30 *tiles* ($\frac{800}{8}$ e $\frac{480}{16}$). Para armazenar todos os *tiles* desta tela é necessário um *buffer* contendo 3000 posições $(100 * 30)$. Foram definidos 128 caracteres, ou seja, 2^7, então, o tamanho deste *buffer* seria de 3000 x 7 bits.

Como o gerador de caracteres implementado permite a utilização de cores individuais para cada caracter, as informações de cor do caracter e do fundo são armazenadas no mesmo *buffer*, ocupando mais 8-bits. Para manter o alinhamento

da memória, cada caracter será armazenado utilizando 16-bits. Então, recalculando, o *buffer* necessário para armazenar todos os caracteres de uma tela deve possuir 3000 * 16 bits, ou ainda, 6Kbytes.

As fontes podem ser redimensionadas para um tamanho maior. Por exemplo, a fonte de 8x16 pode ser escalonada para 16x32, aumentado o tamanho original 4 vezes. Ou seja, um pixel é expandido para quatro pixels. Desta forma, obtém-se uma tela com 50x15 caracteres que, dependendo do dispositivo de vídeo, serão mais legíveis.

O circuito de endereçamento dos caracteres no *buffer* é bastante simples, pois é possível utilizar as coordenadas *pixel_x* e *pixel_y*. Primeiramente, os sinais `pixel_-x(9 downto 3)` e `pixel_y(8 downto 4)` fornecem a posição no *buffer* relativo à coordenada atual do pixel na tela e o circuito de geração de caracteres carrega o *tile* desta posição. O valor do *tile* corresponde ao código ASCII do caracter.

Na segunda etapa, o código ASCII (7-bits) é transformado no MSB (*Most Significant Bit*) do endereço que especifica a posição deste caracter na ROM. Os quatro bits LSB (*Least Significant Bit*) são obtidos da coordenada `pixel_y(3 downto 0)` que indicarão qual a linha do caracter deve ser lida. A figura 4.23 pode ajudar a esclarecer este modo de endereçamento.

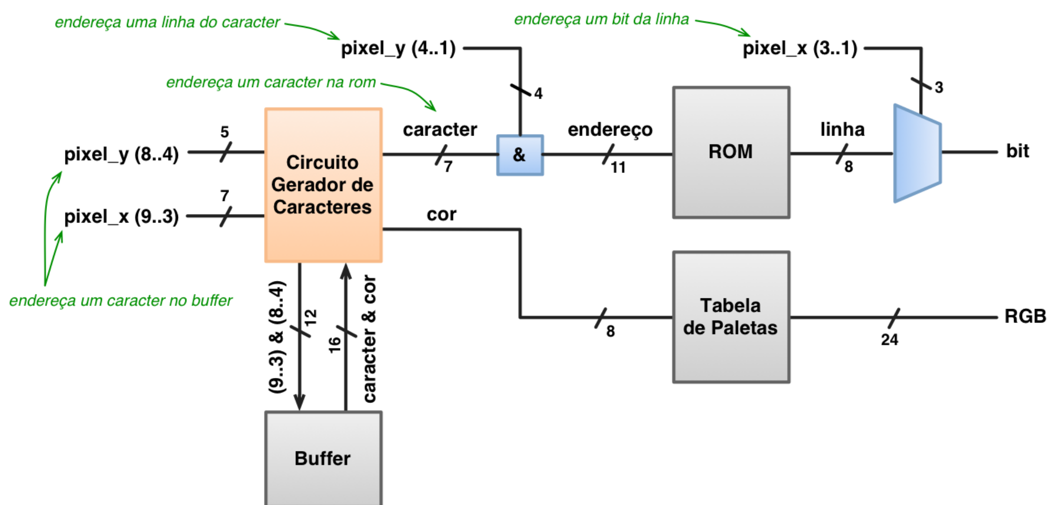


Figura 4.23: Circuito de Geração de Caracteres

Este endereçamento é válido para os caracteres de 8x16 na tela de 100x30. Para a segunda resolução adotada, alguns bits devem ser deslocados. O trecho de código a seguir, exemplifica as duas formas de endereçamento.

```
-- TEXT 50x15 (character 16x32)
elsif display_on = '1' and pixel_x(3 downto 0) = "0000" and reg_video_mode = TEXT_50x15 then
    read_char := '1';
    txtbuf_char_addr(11 downto 10) <= "00";
    txtbuf_char_addr(9 downto 0)   <= pixel_y(8 downto 5) & pixel_x(9 downto 4);
```

```
-- TEXT 100x30 (character 8x16)
elsif display_on = '1' and pixel_x(2 downto 0) = "000" and reg_video_mode = TEXT_100x30 then
    read_char := '1';
    txtbuf_char_addr(11 downto 0) <= pixel_y(8 downto 4) & pixel_x(9 downto 3);
end if;

if read_char = '1' then
    txtbuf_char_read   <= '1';
    caracter           <= txtbuf_char_dat(6 downto 0);
    caracter_color     <= txtbuf_char_dat(15 downto 8);
end if;
```

O endereçamento da linha e do bit de um caracter também é diferente para os dois modos de texto e podem ser visualizados pelo código:

```
-----------------------------------------------------------------------------------------
-- FONT ROM INTERFACE
-----------------------------------------------------------------------------------------
row_addr  <= pixel_y_i(4 downto 1) when video_mode = "000" else pixel_y_i(3 downto 0);

rom_addr  <= char_addr & row_addr;

bit_addr  <= pixel_x_i(3 downto 1) when video_mode = "000" else pixel_x_i(2 downto 0);

font_bit  <= font_word(to_integer(unsigned(not bit_addr)));
```

4.9.4 Controlador com *Frame Buffer*

O esquema de mapeamento de bits utiliza uma memória para armazenar os dados (componentes de cores) que serão exibidos no vídeo. Um *Frame Buffer* é o nome dado ao segmento de memória reservado para armazenar a imagem mapeada em bits.

O sistema pode ser configurado para suportar dois formatos de informação:

- Modo 8-bits paletizado; e
- Modo 24-bits *true color*.

No modo de 8-bits, a informação armazenada no *frame buffer* corresponde a um índice da tabela de paletes. Esta tabela de paletes também é armazenada na memória e, sequencialmente, define as componentes R, G e B para cada cor de 0 a 255. Como somente um byte é armazenado para cada pixel, a imagem pode conter no máximo 256 cores, porém, o usuário possui a liberdade para definir a palete de todas estas cores. Este modo foi implementado como teste, necessitando de melhorias e otimização, portanto, o seu uso não é recomendado por não oferecer benefícios em termos de economia de memória ou desempenho.

O modo 24-bits, também chamado de modo *true color*, armazena diretamente as componentes R, G e B de cada pixel na memória. Desta forma, 3 bytes são

necessários para cada pixel. Como o acesso à memória, no **CASoPC**, é realizado em 32-bits, então, são utilizados 4 bytes para cada pixel e este quarto byte é ignorado. Utilizando apenas 3 bytes, contribuiria para otimizar o uso da memória, porém, aumentaria a lógica computacional necessária para resolver o problema de alinhamento e, consequentemente, prejudicaria o desempenho.

A figura 4.24 apresenta o diagrama em blocos do controlador de vídeo com *frame buffer*.

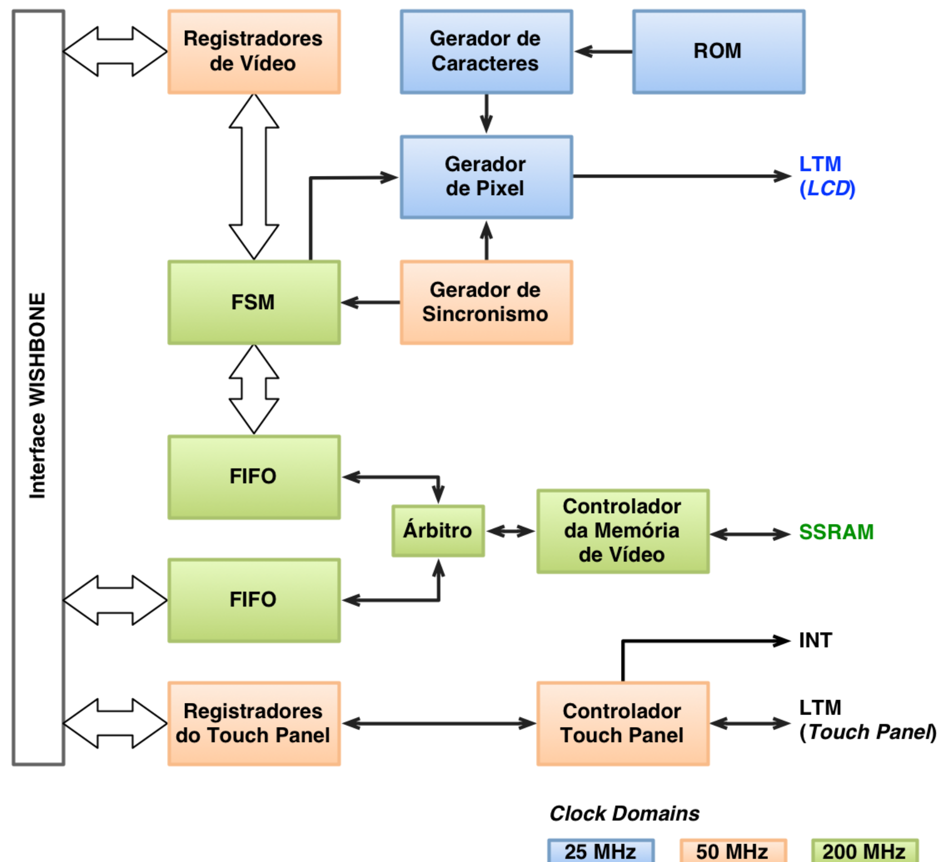


Figura 4.24: Diagrama do Controlador de Vídeo usando SSRAM

Ao contrário do apresentado na seção anterior, o gerador de sincronismo utiliza um clock de 25 MHz. Esta alteração facilita a implementação e o controle do sincronismo entre os componentes do controlador de vídeo, uma vez que, todos os clocks envolvidos no controlador serão múltiplos uns dos outros.

Esta alteração de clock apresenta como consequência a redução de FPS, que passou a ser de 45 quadros por segundo.

A memória utilizada como *frame buffer* é uma memória SSRAM [15]. O controlador implementando para esta memória possui uma FSM (*Finite-State Machine*) que permite que um dado (32-bits) seja lido a cada 4 ciclos de clock. Este é o fator determinante para acessar a memória a 200 MHz. Como esta memória possui apenas uma porta (leitura/escrita), um esquema para multiplexar o acesso teve que

ser criado, inclusive para resolver os problemas com a troca de informações entre componentes contidos em domínios diferentes de clock.

O código VHDL do controlador da memória SSRAM [34] é apresentado no Apêndice C para facilitar futuras referências.

Os problemas para transmitir um sinal de controle de um módulo mais lento (50 MHz) para um módulo mais rápido (200 MHz) pode ser resolvido utilizando um circuito contendo um sincronizador e um detector de borda conforme apresentado na figura 4.25. O clock utilizado neste circuito é o clock do domínio mais rápido, ou seja, 200 MHz.

Figura 4.25: Regeneração de um Sinal Lento

Os problemas na transmissão de dados, podem ser resolvidos com o emprego de FIFOs que implementam domínios de clock diferentes para os circuitos de escrita e de leitura. O emprego destes FIFOs contribui para a solução de um outro problema: a concorrência no acesso à memória, necessitando apenas incluir um árbitro. Uma breve descrição do fluxo de informações do controlador de memória, considerando o diagrama da figura 4.24, é apresentada a seguir:

1. A FSM recebe o clock do gerador de sincronismo e a posição atual do pixel. A máquina de estados aguarda o início de um quadro.

2. Com o início do quadro, o endereço de memória é calculado utilizando os valores de *pixel_x* e *pixel_y* e a FSM insere no FIFO uma solicitação de leitura da memória de vídeo. Para exemplificar o uso do árbitro, considerar-se-á que uma requisição de escrita do processador é inserida no outro FIFO.

3. Como o FIFO conectado à FSM, responsável por ler os dados e encaminhá-los para o vídeo, é sempre prioritário no acesso à memória, a requisição de leitura é enviada para a memória. Uma vez que o dado é lido, ele é retornado para a FSM.

4. O árbitro, verificando que o FIFO conectado à FSM encontra-se vazio, pode atender o outro FIFO, retirando a solicitação de escrita e transferindo para a memória.

Esta implementação ficou comprometida com a necessidade de utilizar, como ROM, a única memória SSRAM disponível. Sem disponibilidade de memória, foi necessário simplificar o controlador para implementá-lo integralmente no FPGA sem o auxílio de componentes externos. Esta implementação simplificada é apresentada na seção 4.9.5 juntamente com a descrição dos demais registradores.

4.9.5 Controlador Básico

Este controlador é uma versão simplificada do controlador utilizando *frame buffer*. Nesta implementação, foi removida a memória SSRAM e os componentes envolvidos no seu acesso. Como esta implementação não conta com o auxílio de memória externa e todo o controlador deve estar contido no FPGA, tornou-se necessária a remoção dos modos gráficos.

Com a retirada da SSRAM e dos demais módulos que operavam a 200 MHz, a complexidade do controlador foi reduzida consideravelmente. A memória, agora necessária apenas para o *buffer* de texto, foi implementada utilizando os recursos do próprio FPGA. Com a possibilidade de implementar uma DPRAM (*Dual-Port RAM*), a concorrência de acesso à memória pode ser desconsiderada, uma vez que cada componente que necessita acessar a memória (barramento Wishbone e controlador interno) está conectado a uma das portas da memória.

O diagrama deste controlador básico é apresentado na figura 4.26.

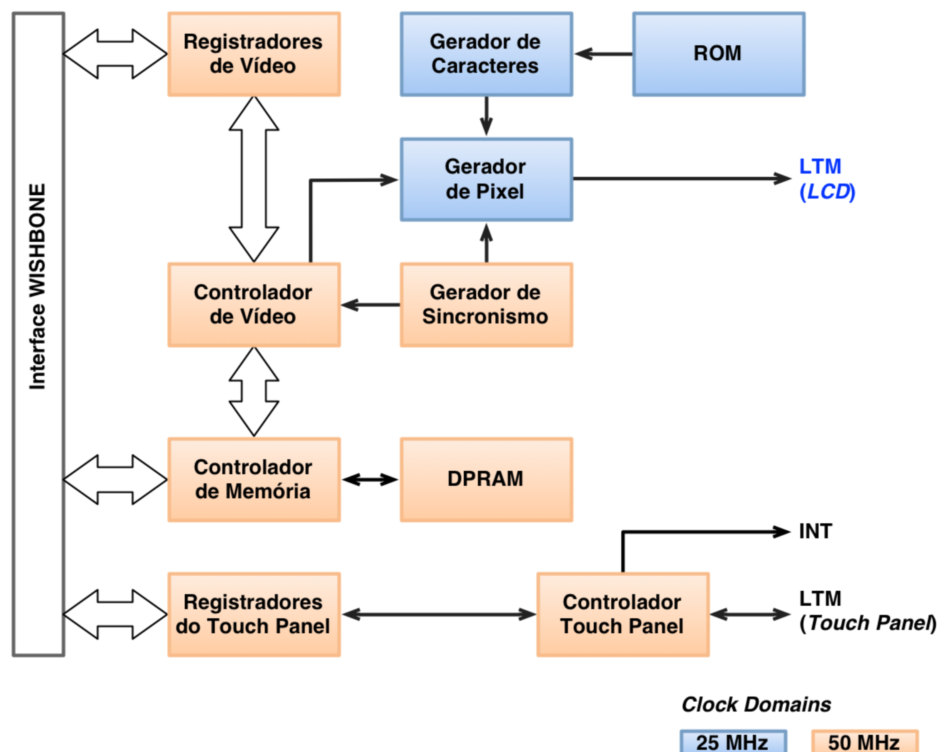


Figura 4.26: Diagrama do Controlador de Vídeo usando DPRAM

Nome do Registrador	Endereço	Função	R/W
CASOPC_VIDEO_MODE	0x8010_0024	Define o modo de texto	R/W
CASOPC_VIDEO_CLEAR	0x8010_0028	Limpa a tela	W
CASOPC_VIDEO_FORECOLOR	0x8010_0030	Define a cor da fonte	W
CASOPC_VIDEO_BACKGROUND	0x8010_0034	Define a cor de fundo	W
CASOPC_VIDEO_PALETTE	0x8010_0040	Configura as paletes de cores	W
CASOPC_VIDEO_BUFFER	0x8020_0000	*Buffer* de vídeo	W

Tabela 4.7: Registradores de Vídeo

CASOPC_VIDEO_MODE

Configura o modo de vídeo.

Atualmente somente os modos de texto são suportados.
0 - Modo texto 50x15
1 - Modo texto 100x30

CASOPC_VIDEO_CLEAR

Limpa a tela ao escrever o valor "1" neste registrador. A tela será preenchida com a cor definida por CASOPC_VIDEO_BACKGROUND.

CASOPC_VIDEO_FORECOLOR

Define a cor da fonte. Valores suportados: 0 a 15.

CASOPC_VIDEO_BACKGROUND

Define a cor do fundo da tela. Valores suportados: 0 a 15.

CASOPC_VIDEO_PALETTE

Buffer que armazena as definições das paletas das 16 cores suportadas pelo modo texto.

Ao inicializar o sistema, a paleta de cores é copiada para o *buffer*, no endereço 0x8010_0040, para que o usuário possa alterá-la. A definição *default* das cores é apresentada no código a seguir:

```
-- Default Color Palette (for text mode)
type list_colors_type is array(integer range <>) of std_logic_vector(23 downto 0);
constant DEFAULT_COLORS : list_colors_type(0 to 15) := (
    X"000000",        --  0 - BLACK:           0,   0,   0
    X"0000AA",        --  1 - BLUE:            0,   0, 170
    X"00AA00",        --  2 - GREEN:           0, 170,   0
    X"00AAAA",        --  3 - CYAN:            0, 170, 170
    X"AA0000",        --  4 - RED:           170,   0,   0
    X"AA00AA",        --  5 - MAGENTA:       170,   0, 170
    X"AA5500",        --  6 - BROWN:         170,  85,   0
    X"AAAAAA",        --  7 - LIGHT GRAY:    170, 170, 170
```

Cor #	Nome	Endereço
0	BLACK	0x8010_0040
1	BLUE	0x8010_0044
2	GREEN	0x8010_0048
3	CYAN	0x8010_004C
4	RED	0x8010_0050
5	MAGENTA	0x8010_0054
6	BROWN	0x8010_0058
7	LIGHT GRAY	0x8010_005C
8	DARK GRAY	0x8010_0060
9	LIGHT BLUE	0x8010_0064
10	LIGHT GREEN	0x8010_0068
11	LIGHT CYAN	0x8010_006C
12	LIGHT RED	0x8010_0070
13	LIGHT MAGENTA	0x8010_0074
14	YELLOW	0x8010_0078
15	WHITE	0x8010_007C

```
        X"555555",        --  8 - DARK GRAY:      85,  85,  85
        X"5555FF",        --  9 - LIGHT BLUE:     85,  85, 255
        X"55FF55",        -- 10 - LIGHT GREEN:    85, 255,  85
        X"55FFFF",        -- 11 - LIGHT CYAN:     85, 255, 255
        X"FF5555",        -- 12 - LIGHT RED:     255,  85,  85
        X"FF55FF",        -- 13 - LIGHT MAGENTA: 255,  85, 255
        X"FFFF55",        -- 14 - YELLOW:        255, 255,  85
        X"FFFFFF"         -- 15 - WHITE:         255, 255, 255
    );
```

`CASOPC_VIDEO_BUFFER`

Buffer que armazena os caracteres que serão exibidos no vídeo. Os dados possuem o formato:

31..16	15..12	11..8	7	6..0
Não usado	Cor da fonte	Cor do fundo	Não usado	Caracter

Para as cores da fonte e do fundo podem ser utilizados os mesmos valores apresentados anteriormente (valores: 0 a 15).

O valor do caracter corresponde ao código ASCII (valores: 0 a 127).

4.9.6 Touch Panel

O controlador do *touch panel* é basicamente independente do módulo de vídeo. A sua implementação está incluída no controlador de vídeo por questão de conveniência, uma vez que, a interface do *touch panel* é parte integrante do dispositivo do LCD.

O módulo de *touch panel* inclui uma interface à parte da interface do LCD, conforme apresenta na figura 4.27.

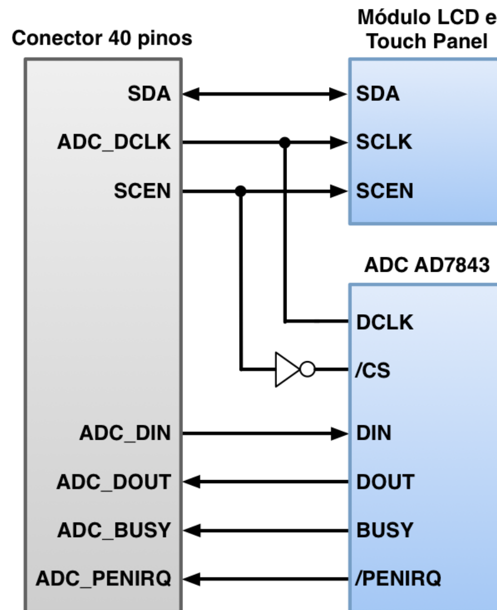


Figura 4.27: Interface Serial do Módulo LCD e *Touch Panel*

O LTM (*LCD Touch Panel Module*) inclui um ADC de 12-bits para digitalizar as coordenadas X e Y dos pontos pressionados. Estas coordenadas são armazenadas no módulo e poderão ser lidas utilizando uma interface serial. O LTM inclui um sinal, `ADC_PENIRQ_n`, de interrupção. Embora não esteja presente no diagrama anterior, o sinal de interrupção está conectado a um resistor de *pull-up*. Quando a tela é pressionada em algum ponto, este sinal passará para nível "0" indicando ao FPGA a ocorrência de um pressionamento.

A figura 4.28 apresenta um diagrama de operação da interface serial do ADC. Como os sinais de clock (`ADC_DCLK`) e *chip enable* (`SCEN`) da interface serial são compartilhados com o módulo de LCD, estes dois módulos, LCD e ADC, não devem ser operador simultaneamente. Para evitar este conflito, o sinal de *enable* do ADC é invertido em relação ao módulo do LCD.

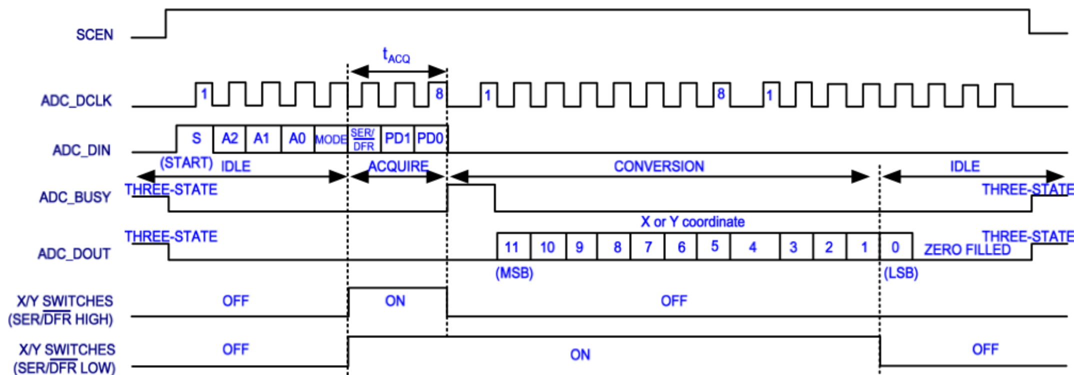


Figura 4.28: Temporização da Interface Serial

O código VHDL do controlador do *Touch Panel* encontra-se listado no Apêndice

D para facilitar futuras pesquisas e referências.

Na tabela 4.8 são apresentadas as funções implementadas no **CASoPC** e é seguida por uma breve descrição destas funções.

Nome do Registrador	**Endereço**	**Função**	**R/W**
`CASOPC_TOUCH_ENABLE`	0x8010_0000	Habilita o *Touch Panel*	R/W
`CASOPC_TOUCH_INT_ENABLE`	0x8010_0004	Habilita o uso de interrupção	R/W
`CASOPC_TOUCH_INT_CLEAR`	0x8010_0008	Limpa o registrador de interrupção	W
`CASOPC_TOUCH_COORD_XY`	0x8010_000C	Retorna as coordenadas X e Y	R
`CASOPC_TOUCH_COORD_X`	0x8010_0010	Retorna a coordenada X	R
`CASOPC_TOUCH_COORD_Y`	0x8010_0014	Retorna a coordenada Y	R

Tabela 4.8: Registradores do *Touch Panel*

`CASOPC_TOUCH_ENABLE`

Habilita(1) ou desabilita(0) o *touch panel*.

`CASOPC_TOUCH_INT_ENABLE`

Habilita(1) ou desabilita(0) a geração de interrupção pelo *touch panel*.

`CASOPC_TOUCH_INT_CLEAR`

Limpa o registrador de interrupção.

`CASOPC_TOUCH_COORD_XY`

Registrador que armazena as últimas coordenadas X e Y do local do pressionamento do *touch panel*. O valor retornado possui o formato:

31..28	27..16	15..12	11..0
Não usado	Coordenada Y	Não usado	Coordenada X

`CASOPC_TOUCH_COORD_X`

Registrador que armazena a coordenada X do pressionamento.

`CASOPC_TOUCH_COORD_Y`

Registrador que armazena a coordenada Y do pressionamento.

4.10 Controlador do LCD 16x2

Com a intenção de possuir mais uma alternativa para exibição de informações ao usuário, foi implementado um módulo para controlar o LCD de 16 caracteres e 2 linhas disponível na plataforma de desenvolvimento.

Este LCD, de modelo CFAH1602B [35] fabricado pela Crystalfontz, possui uma interface simples e disponibiliza dois registradores internos de 8-bits, um registrador de instrução (IR) e um registrador de dados (DR).

O IR armazena os códigos de instruções, como limpar a tela ou deslocar o cursor. O registrador de dados armazena temporariamente os dados que serão escritos nas suas memórias internas.

Como mencionado anteriormente, a interface com o LCD é simples porém deve-se respeitar algumas temporizações conforme apresentado na figura 4.29 e tabela 4.9. A figura apresenta o diagrama para as operações de escrita.

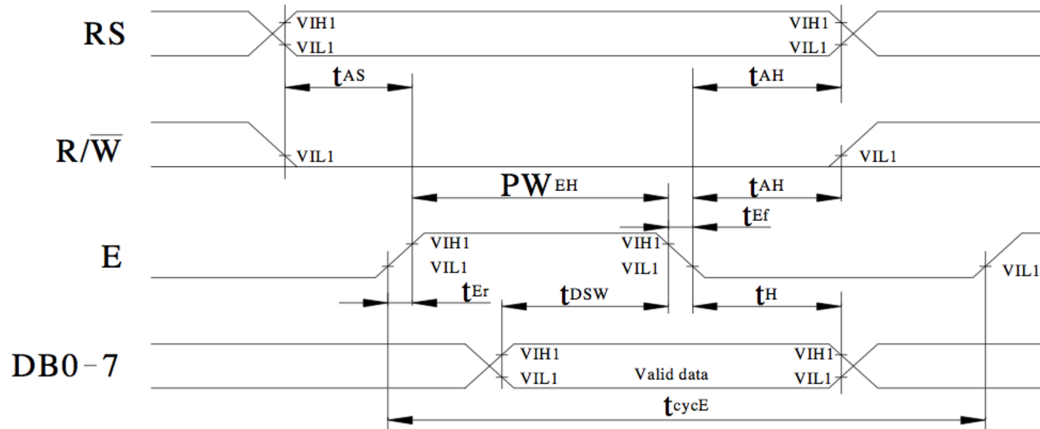


Figura 4.29: Temporização do LCD para Escrita

Item	Símbolo	Min	Máx	Unidade
Enable cycle time	t_{cycE}	500	-	ns
Enable pulse width (high level)	PW_{EH}	230	-	ns
Enable rise/fall time	t_{Er}, t_{Ef}	-	20	ns
Address setup time (RS, R/W to E)	t_{AS}	40	-	ns
Address hold time	t_{AH}	10	-	ns
Data setup time	t_{DSW}	80	-	ns
Data hold time	t_H	10	-	ns

Tabela 4.9: Temporização do LCD

O código VHDL do controlador do LCD 16x2 implementando uma máquina de estados e o controle das temporizações requeridas é listado no Apêndice E para facilitar futuras pesquisas e referências.

Na tabela 4.10 são listadas as funções implementadas no **CASoPC** com uma breve descrição apresentada na sequência.

Nome do Registrador	Endereço	Função	R/W
`CASOPC_LCD_ENABLE`	0x8002_0000	LCD on/off	W
`CASOPC_LCD_MODE`	0x8002_0004	Define modo byte/dword	W
`CASOPC_LCD_CLEAR`	0x8002_0008	Limpa uma linha do LCD	W
`CASOPC_LCD_SWITCH_LINES`	0x8002_000C	Inverte as linhas do LCD	W
`CASOPC_LCD_LINE_1`	0x8002_0010	*Buffer* da linha 1 do LCD	W
`CASOPC_LCD_LINE_2`	0x8002_0020	*Buffer* da linha 2 do LCD	W

Tabela 4.10: Registradores do Módulo LCD 16x2

`CASOPC_LCD_ENABLE`

Liga(1) ou desliga(0) o LCD.

`CASOPC_LCD_MODE`

Configura a transferência de dados para os *buffers* do LCD como byte(0) ou dword(1).

`CASOPC_LCD_CLEAR`

Limpa uma linha do LCD, esta função (não registrada), possui o seguinte formato:

31..2	1	0
Não usado	Linha 2	Linha 1

`CASOPC_LCD_SWITCH_LINES`

Este registrador controla a apresentação das linhas. Com o valor "0" a apresentação será normal e com valor "1" a apresentação das duas linhas será invertida. Os *buffers* de cada linha não serão alterados, somente a apresentação no LCD.

`CASOPC_LCD_LINE_1` e `CASOPC_LCD_LINE_2`

Buffers das linhas do LCD contendo 16 bytes. Este *buffer* pode ser endereçado byte a byte se `CASOPC_LCD_MODE=0` ou endereçado de 4 em 4 bytes em modo *dword*.

4.11 Timers

Conforme apresentado nos capítulos iniciais, para a execução do *Tilt Test* é necessário comparar a leitura de diferentes sensores ao mesmo tempo. Porém, para quantificar este "ao mesmo tempo" faz-se necessário o uso de contadores. Embora esteja prevista a inclusão futura de um RTC (*Real-Time Clock*), para a versão atual, onde é necessária apenas a medição de tempos relativos, foram incluídos dois temporizadores simples.

Os contadores implementados possuem 32-bits e possuem funcionamento bastante simples. Inicialmente eles são zerados e, quando habilitados, são incrementados a cada ciclo de clock. Quando o valor do contador alcançar um valor de *threshold*, um sinal de interrupção é gerado ao processador, o contador é então zerado e uma nova contagem se inicia.

Obtendo o período de tempo de cada ciclo de clock (50 MHz), chega-se ao valor de 20 ns. Desta forma, cada contador é capaz de realizar contagem até $2^{32} * 20ns$, o que resulta em aproximadamente 85 segundos.

Na tabela 4.11 são apresentados os registradores do módulo dos temporizadores.

Nome do Registrador	Endereço	Função	R/W
`CASOPC_TIMER1_CONTROL`	0x8005_0000	Set/Reset contador	W
`CASOPC_TIMER1_THRESHOLD`	0x8005_0004	Limite do contador	R/W
`CASOPC_TIMER1_COUNTER`	0x8005_0008	Valor do contador	R
`CASOPC_TIMER2_CONTROL`	0x8005_000C	Set/Reset contador	W
`CASOPC_TIMER2_THRESHOLD`	0x8005_0010	Limite do contador	R/W
`CASOPC_TIMER2_COUNTER`	0x8005_0014	Valor do contador	R

Tabela 4.11: Registradores dos Contadores

Os registradores de contagem (somente leitura) e *threshold* (leitura/escrita) possuem 32-bits. O registrador de controle possui apenas 2-bits e suas funções são apresentadas na tabela 4.12.

Bit #	Função
31 .. 2	Não usado
1	Reset(1) do Timer
0	Timer (1)ligado/(0)desligado

Tabela 4.12: Composição do Registrador de Controle

4.12 Periféricos Básicos de E/S

A plataforma de desenvolvimento dispõe de alguns dispositivos extras como LEDs, botões tipo chave e tipo *push-button* e displays de 7 segmentos.

A seguir serão apresentados de maneira breve a implementação dos módulos de controle destes periféricos.

4.12.1 LEDs

Existem disponíveis 18 LEDs de cor vermelha e 9 LEDs verdes. As funções implementadas para acender ou apagar cada um desses LEDs são apresentadas na tabela 4.13.

Nome do Registrador	**Endereço**	**Função**	**R/W**
`CASOPC_LED_RED`	0x8007_0000	Liga(1)/Desliga(0) os LEDs vermelhos	W
`CASOPC_LED_GREEN`	0x8007_0004	Liga(1)/Desliga(0) os LEDs verdes	W

Tabela 4.13: Registradores dos LEDs

Cada bit dos registradores apresentados na tabela corresponde a um LED. A quantidade de LEDs e, consequentemente, o tamanho destes registradores é definida pelo trecho de código apresentado a seguir.

```
-----------------------------------------------------------------------------------------
-- LEDS
-----------------------------------------------------------------------------------------
constant LED_RED_WIDTH        : natural := 18;
constant LED_GREEN_WIDTH      : natural := 9;
```

4.12.2 Hex 7 Segmentos

O módulo desenvolvido para controlar os 8 *displays* de 7 segmentos, implementa as funções apresentadas na tabela 4.14.

Nome do Registrador	**Endereço**	**Função**	**R/W**
`CASOPC_HEX_MASK`	0x8001_0000	Define a máscara de escrita	W
`CASOPC_HEX_DATA`	0x8001_0004	Lê/Escreve um valor em hexadecimal nos displays	R/W

Tabela 4.14: Registradores dos Displays de 7 Segmentos

O registrador `CASOPC_HEX_MASK` possui 8-bits e cada bit corresponde a um dos displays de 7 segmentos. Este registrador define a máscara utilizada nas operações de escrita. Ou seja, somente os displays com o bit correspondente habilitado no registrador de máscara, terá o seu valor atualizado pelo registrador `CASOPC_HEX_DATA`.

O formato do registrador de dados é apresentado a seguir:

31..28	27..24	23..20	19..16	15..12	11..8	7..4	3..0
Hex 7	Hex 6	Hex 5	Hex 4	Hex 3	Hex 2	Hex 1	Hex 0

4.12.3 Botões

Existem dois tipos de botões disponíveis, chaves liga/desliga e botões de pressão (*push-button*). A configuração do número de chaves e botões e, consequentemente, o tamanho dos seus respectivos registradores é definida pelo trecho de código apresentado a seguir:

```
------------------------------------------------------------------------------------------
-- SWITCHES & KEYS
------------------------------------------------------------------------------------------
constant BTN_SWITCH_WIDTH    : natural := 18;      -- Switches ON/OFF
constant BTN_KEY_WIDTH       : natural := 3;       -- Push buttons

constant KEY_OPEN_LEVEL      : std_logic := '1';   -- Value of the KEY when not pressed
                                                   -- (as the button is physically connected)
```

Como pode-se perceber pelo código apresentado, os botões *push-button* podem estar conectados fisicamente ao FPGA como "normalmente aberto" ou "normalmente fechado". Pelo diagrama esquemático da figura 4.30, verifica-se que as chaves estão ligadas a resistores de *pull-up* que os tornam "normalmente fechados". Quando pressionados, os respectivos pinos do *transceiver* de barramento (74HC245) são aterrados.

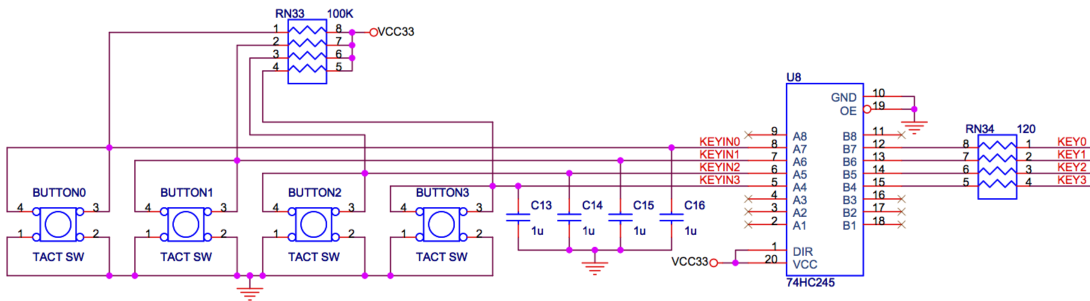


Figura 4.30: Diagrama das Chaves *Push-Button*

A tabela 4.15 apresenta as funções implementadas para as chaves e botões. O pressionamento de um botão ou a mudança de estado de uma chave gera uma interrupção ao processador. Para verificar qual botão ou chave sofreu alguma alteração, deve-se consultar os registradores `CASOPC_SW_STATUS` e `CASOPC_KEY_STATUS`.

4.13 Módulo de *Debug*

Durante o desenvolvimento de um sistema, é de vital importância a realização de testes para verificar se o funcionamento e comportamento estão conforme o esperado.

Nome do Registrador	Endereço	Função	R/W
CASOPC_SW_INT_ENABLE	0x8006_0000	Habilita(1) interrupção	W
CASOPC_SW_INT_MASK	0x8006_0004	Define a máscara de interrupção	W
CASOPC_SW_STATUS	0x8006_0008	Retorna o estado das chaves	R
CASOPC_SW_INT_VECTOR	0x8006_000C	Retorna as chaves que sofreram alteração (geraram interrupção)	R
CASOPC_KEY_INT_ENABLE	0x8006_0010	Habilita(1) interrupção	W
CASOPC_KEY_INT_MASK	0x8006_0014	Define a máscara de interrupção	W
CASOPC_KEY_STATUS	0x8006_0018	Retorna o estado dos botões	R
CASOPC_KEY_INT_VECTOR	0x8006_001C	Retorna os botões que foram pressionados (geraram interrupção)	R
CASOPC_SW_CLEAR	0x8006_0020	Limpa o registrador de estado/interrupção das chaves	W
CASOPC_KEY_CLEAR	0x8006_0024	Limpa o registrador de estado/interrupção dos botões	W

Tabela 4.15: Registradores das Chaves e Botões

Quando o sistema envolve o desenvolvimento de hardware e software, essa verificação torna-se, por vezes, difícil de ser realizada.

A verificação de alguns módulos pode ser realizada utilizando de simulações, porém, o funcionamento integrado dos módulos, bem como, o sistema como um todo, torna-se de difícil simulação. É mais adequada a realização de testes diretamente no hardware.

Para facilitar a execução destes testes, foi incluído um módulo capaz de acessar o barramento do sistema e, desta forma, acessar os diversos periféricos. Este módulo é inserido como um *Master* no barramento Wishbone e permite a escrita e leitura dos registradores dos componentes do sistema. Utilizando um emulador de terminal (como o *minicom* no Linux ou *HyperTerminal* no Windows) conectado, via porta serial, ao módulo **RS232_syscon** [36], o usuário é capaz de enviar comandos ao sistema e verificar os resultados retornados.

O módulo RS232_syscon, originalmente, compartilha o barramento *Master* com o

processador principal do sistema e implementa um protocolo para requisitar acesso ao barramento. Contudo, como a utilização de diversos dispositivos mestres não estava prevista na implementação do protótipo, os sinais do tipo `grant` e a lógica associada para permitir o compartilhamento do barramento não foram implementadas. Desta forma, o módulo RS232_syscon é ativado somente com o sistema em *modo debug* e, neste caso, o processador principal estará "desconectado" do barramento e não poderá ser utilizado.

O esquema de conexão deste módulo ao sistema é simples e é apresentado na figura 4.31.

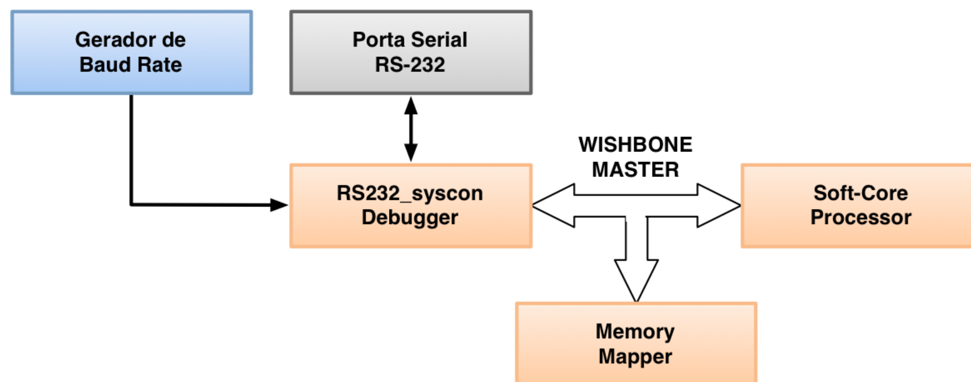


Figura 4.31: Conexão do Módulo RS232_syscon ao Barramento Mestre

Os sinais `ack_i` e `err_i` são usados para determinar se os ciclos de barramento foram corretamente executados. O módulo inclui ainda um *watchdog* para determinar se o sinal `ack_i` foi recebido muito tarde e envia uma indicação de erro ao terminal remoto. O valor de *timeout* do *watchdog* pode ser configurado para atender os requisitos do sistema em teste.

O *debug* possui três comandos: leitura, escrita e reset. O envio do caracter "*Enter*" encerra o comando e inicia o processo de verificação e execução do comando. Os valores devem ser escritos em hexadecimal.

A sintaxe dos comandos suportados são:

Comando	**Sintaxe**
Escrita	`w` *endereço dado [repetição]*
Leitura	`r` *endereço [repetição]*
Reset	`i`

Tabela 4.16: Comandos do Módulo de *Debug*

O parâmetro *repetição* (opcional) indica a quantidade de repetições que devem ser realizadas para o comando. Utilizando os comando `w` e `r` sem o uso de parâmetros, o último comando será repetido.

A seguir são apresentados alguns exemplos:

`w 80070000 01`

Escreve o valor "00000001" (lembrando que o barramento de dados do sistema é de 32-bits) no endereço 0x8007_0000.

`r 80050008`

Lê o valor do endereço 0x8005_0008.

A tabela 4.17 apresenta as mensagens que podem ser enviadas ao terminal remoto em resposta a um comando.

Resposta	Significado
OK	O comando foi interpretado e executado com sucesso
?	Comprimento da linha foi excedido
A?	Erro de interpretação no campo de endereço
D?	Erro de interpretação no campo de dados
Q?	Erro de interpretação no campo de repetição
A?	O sistema habilitou o sinal `err_i` ou *timeout* do *watchdog* (módulo de debug não recebeu o *ACK*)
B!	*Timeout* do *watchdog* na requisição de acesso ao barramento mestre

Tabela 4.17: Respostas Geradas pelo Módulo de Debug

O *watchdog* possui duas funcionalidades na execução de um comando, determinar a ocorrência de um *timeout* na obtenção de acesso ao barramento mestre e verificar o tempo de recebimento do sinal de *ACK* transmitido pelo módulo requisitado ao término de um ciclo de barramento.

Capítulo 5

Conclusões

Este trabalho buscou, entre seus objetivos, verificar a viabilidade de implementação de um sistema computacional utilizando plataformas reconfiguráveis com desempenho suficiente para a execução do *Tilt Test* Naval.

A conclusão desta implementação pode ser dividida em duas partes, a verificação do sistema na plataforma reconfigurável e a capacidade do seu emprego para o *Tilt Test* Naval e demais sistemas de interesse.

Foi desenvolvido um sistema computacional completo embarcado em um FPGA com a disponibilidade de alguns dispositivos periféricos. Convém ressaltar que, devido a algumas limitações da plataforma utilizada, alguns recursos ficaram comprometidos. Desta forma, o sistema não encontra-se totalmente funcional. A seguir são apresentadas algumas considerações sobre a implementação e os componentes que necessitarão de atenção no futuro.

Processador *soft-core*
: A escolha do *MB-Lite* foi acertada com relação ao desempenho, flexibilidade e facilidades de implementação e programação, porém, com a integração do módulo de *debug*, foi percebida a ausência de sinais de arbitragem para as interfaces *Master*. Uma solução intermediária, exequível em um tempo curto e sem grandes alterações na arquitetura do sistema, foi incluir uma diretiva para habilitar apenas um mestre em tempo de síntese. Desta forma, não é possível utilizar as facilidades do módulo de *debug* enquanto o processador estiver em funcionamento. A solução definitiva encontra-se na alteração da interface do processador para suportar mais de um *Master* no mesmo barramento. Por outro lado, a utilização de um outro processador avaliado, *SecretBlaze*, além de oferecer um ligeiro aumento de desempenho, já oferece a interface com os demais sinais necessários implementados.

Plataforma de desenvolvimento
: As memórias disponíveis na plataforma adotada, não atendem completamente

o requerido pelo sistema. Ressalta-se a necessidade de utilização da memória SSRAM como memória ROM, destinada inicialmente para o controlador de vídeo. Este artifício poderia ser contornado caso houvesse a disponibilidade de pinos de entrada e saída para a conexão de módulos externos de memória. Porém, como a maior parte das conexões de E/S do FPGA estão direcionadas para os dispositivos periféricos existentes na plataforma de desenvolvimento, os pinos restantes não são em número suficiente.

Controlador de Vídeo

Este módulo, para ser considerado totalmente funcional, necessita a conexão com uma memória de vídeo para permitir o uso do modo gráfico, importante para exibir os resultados do alinhamento. Outra necessidade é a disponibilidade de sinais padrão VGA para permitir a ligação de um monitor. Entretanto, como a forma de geração dos sinais de sincronismo para este padrão é semelhante à forma utilizada para o LCD empregado, esta adaptação não implica em grandes dificuldades.

Módulo UART

Pode-se dizer que as portas seriais são os elementos principais do *Tilt Test* pois, através delas, são realizadas as leituras dos sensores. Então, torna-se essencial aumentar a eficiência e precisão deste módulo. Lembrando que um dos propósitos do *Tilt Test* Naval é permitir a realização de alinhamento com o navio atracado ou navegando, ou seja, com os efeitos do balanço e caturro, então, os dados devem ser obtidos com o mínimo de atraso entre as leituras de cada sensor. Desta forma, considera-se importante a inclusão de um *timer* no módulo de UART para controlar e verificar a validade dos dados. Executando a chamada filtragem de sincronização diretamente no hardware de forma mais precisa.

Controlador USB

A existência de um dispositivo de armazenamento é essencial para permitir a gravação dos dados do *Tilt Test* para análises posteriores e geração de relatórios. Um módulo para o controle de dispositivos USB foi implementado e testado, porém, o suporte disponibilizado ao hardware permite apenas leituras em modo *raw*, ou seja, utilizando um endereçamento direto por setores. Para permitir o uso deste módulo é imprescindível a criação de uma camada de software para acesso a sistemas de arquivos como, por exemplo, FAT32 (formato mais comum encontrado em dispositivos tipo *pendrives*). Como segunda linha de ação, encontra-se o uso de um sistema operacional pois, desta forma, os métodos de acesso a alguns periféricos tornam-se transparentes ao programador do software aplicativo. A execução desta abordagem, requer um

estudo dos sistema operacionais disponíveis e das padronizações de hardware exigidas.

Considerando os resultados alcançados nos primeiros testes, o sistema embarcado no FPGA, mesmo com as limitações presentes, é capaz de desenvolver o controle e a leitura dos sensores e executar o processamento dos dados de *tilt* utilizando uma programação de software simples e com bom desempenho (figura 5.1).

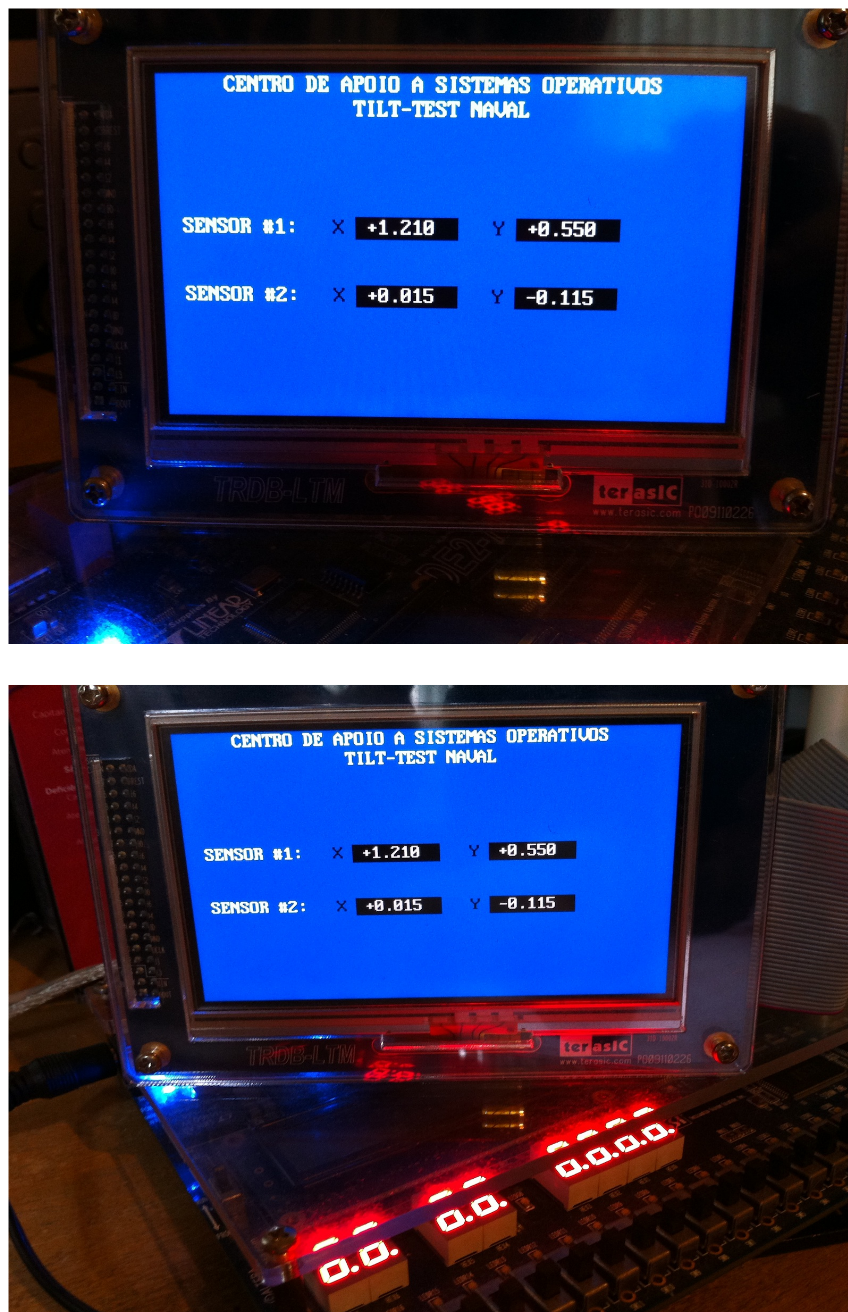


Figura 5.1: *Tilt Test* Naval

Embora o sistema ainda necessite ser submetido a testes mais rigorosos para permitir o conhecimento pleno das suas capacidades e limitações, o resultado obtido

é motivador para a continuação e conclusão do trabalho iniciado. Contudo, ainda é necessário apresentar algumas considerações adicionais.

Normalmente, os *kits* de desenvolvimento são utilizados para abrigar o projeto inicial e, após as correções e adaptações, migrá-lo para um sistema confeccionado especificamente para suportar a aplicação. Porém, deve-se levar em consideração os custos envolvidos no projeto e confecção de placas de circuitos complexos e na aquisição dos componentes. Convém realçar que o FPGA utilizado neste trabalho possui 896 pinos, ou seja, um grande complicador para o trabalho de roteamento e projeto da placa de circuito que, provavelmente, possuirá mais de 2 camadas, elevando ainda mais o custo de produção.

Considerando que o projeto indica a confecção de algumas poucas unidades, deve ser considerada a utilização do próprio *kit* de desenvolvimento como plataforma final para o sistema em produção.

Da mesma forma, deve-se considerar a necessidade de utilização de processadores *soft-cores* que, embora apresentem bom desempenho por MHz, acabam sendo limitados pelas frequências utilizadas pelo FPGA e demais componentes da plataforma. Atenta a este segmento de mercado, a indústria de FPGA, disponibiliza atualmente plataformas de desenvolvimento contendo, além do FPGA, um processador *hard-core* de alto desempenho e baixo consumo. Estes processadores, possuindo dois ou quatro núcleos, normalmente são utilizados em *tablets* e *smartphones* e, consequentemente, são otimizados para alcançar excelente *performance* com baixo consumo de energia. Como exemplo, é possível citar uma plataforma que embarca, além de um FPGA contendo 100K LE e controlador interno de memórias, um processador ARM Cortex^{TM}-A9 de dois núcleos operando a 800 MHz e acelerador gráfico.

Para finalizar, a viabilidade apresentada com a implementação do *Tilt Test* Naval utilizando uma plataforma reconfigurável é fator importante de conhecimento e experiência, motivando alcançar um sistema completamente funcional, porém, utilizando uma plataforma que ofereça desempenho superior, provavelmente, para abrigar também a versão final do projeto.

Referências Bibliográficas

[1] *Manual de Alinhamento das Fragatas MK-10 (ALT-1).* Centro de Apoio a Sistemas Operativos, Marinha do Brasil, Rio de Janeiro, 1991.

[2] *Manual de Alinhamento das Corvetas Classe "Inhaúma"(MAC).* Centro de Apoio a Sistemas Operativos, Marinha do Brasil, Rio de Janeiro, 1996.

[3] APOSTOL, T. M., MNATSAKANIAN, M. A. *New Horizons in Geometry.* Washington, The Mathematical Association of America, 2012.

[4] *Relatório do Projeto de Automação das Medidas dos Ângulos de "Tilt Test".* Relatório técnico, Centro de Apoio a Sistemas Operativos, Marinha do Brasil, Rio de Janeiro, 2011.

[5] *T7 Communication User Guide*, 1.3 ed. US Digital, Washington, 2009.

[6] *T7 Networked Absolute Inclinometer.* US Digital, Washington, 2009.

[7] *ACA2000 Digital Type Dual-Axis Inclinometer - Product Specifications.* Rion Sensor Technology, 2011.

[8] HUFFMIRE, T., IRVINE, C., NGUYEN, T. D., et al. *Handbook of FPGA Design Security.* New York, Springer, 2010.

[9] GULATI, K., KHATRI, S. P. *Hardware Acceleration of EDA Algorithms.* New York, Springer, 2010.

[10] SASS, R., SCHMIDTH, A. G. *Embedded Systems Design with Platform FPGAs.* Massachusetts, Morgan Kaufmann, 2010.

[11] PEDRONI, V. A. *Circuit Design with VHDL.* Massachusetts, Massachusetts Institute of Technology, 2004.

[12] CHU, P. P. *RTL Hardware Design Using VHDL.* New Jersey, John Wiley & Sons, 2006.

[13] HAMBLEN, J. O., HALL, T. S., FURMAN, M. D. *Rapid Prototyping of Digital Systems: Quartus II Edition.* New York, Springer, 2006.

[14] *Cyclone II Device Handbook*. Altera Corporation, 2008.

[15] *IS61VPS51236A 18MBit Synchronous Pipelined, Single Cycle Static Ram*. Integrated Silicon Solution (ISSI), 2007.

[16] *IS42S16160B 256MBit Synchronous DRAM*. Integrated Silicon Solution (ISSI), 2007.

[17] *S29GL064A 64MBit Page Mode Flash Memory*. Spansion Inc, 2007.

[18] *Digital Touch Panel Development Kit User Guide*. Terasic Technologies, 2007.

[19] JOSÉ, W. M. *Sistema de Multiprocessamento numa FPGA*. Tese de Mestrado, Universidade Técnica de Lisboa, 2011.

[20] KRANENBURG, T. *Design of a Portable and Customizable Microprocessor for Rapid System Prototyping*. Tese de Mestrado, Delft University of Technology, 2009.

[21] "1-Core Technologies". . Disponível em: <`http://www.1-core.com/library/digital/soft-cpu-cores/`>.

[22] BARTHE, L., CARGNINI, L. V., BENOIT, P., et al. "The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor", *25th IEEE International Parallel & Distributed Processing Symposium*, pp. 310–313, 2011.

[23] PLAVEC, F. *Soft-Core Processor Design*. Tese de Mestrado, University of Toronto, 2004.

[24] *MicroBlaze Processor Reference Guide*, 12.0 ed. Xilinx, 2011.

[25] STALLINGS, W. *Arquitetura e Organização de Computadores*. 5 ed. São Paulo, Pearson Prentice Hall, 2003.

[26] "OpenCores". . Disponível em: <`http://opencores.org`>.

[27] *Wishbone SoC Interconnection Architecture for Portable IP Cores*, b4 ed. OpenCores, 2010.

[28] HERVEILLE, R. *Combining Wishbone Interface Signals - Application Note*, 0.2 ed. OpenCores, 2001.

[29] AIRES, J. M. S. *Migração de Funcionalidades de um Sistema Operativo de Tempo Real para Gateware de um Processador*. Tese de Mestrado, Universidade do Minho, 2012.

[30] HARRIS, D. M., HARRIS, S. L. *Digital Design and Computer Architecture.* California, Morgan Kaufmann, 2007.

[31] CHU, P. P. *FPGA Prototyping by VHDL Examples.* New Jersey, John Wiley & Sons, 2008.

[32] *PmodRS232 Converter Module Board Reference Manual.* Digilent Inc., 2012.

[33] *PmodRS232 Peripheral Module.* Digilent Inc., 2011.

[34] OSMAN, A. *aoOCS Specification*, 2010.

[35] *CFAH1602B-TMC-JP Character LCD Module.* Crystalfontz America.

[36] CLAYTON, J. *RS232_syscon Users Guide*, 2001.

[37] PEDRONI, V. A. *Digital Electronics and Design with VHDL.* Massachusetts, Elsevier, 2008.

[38] SANDIGE, R. S., SANDIGE, M. L. *Fundamentals of Digital and Computer Design with VHDL.* New York, McGraw-Hill, 2012.

[39] BROWN, S., VRANESIC, Z. *Fundamentals of Digital Logic with VHDL Design.* 3 ed. New York, McGraw-Hill, 2009.

[40] MATTSSON, D., CHRISTENSSON, M. *Evaluation of Synthesizable CPU Cores.* Tese de Mestrado, Chalmers University of Technology, 2004.

[41] TONG, J. G., ANDERSON, I. D. L., KHALID, M. A. S. “Soft-Core Processors for Embedded Systems”, *18th Internationa Conference on Microelectronics*, 2006.

[42] KRANENBURG, T., VAN LEUKEN, R. “MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture”, *European Design Automation Association*, 2010.

[43] DE QUEIROZ, D. C. *Implementação do barramento on-chip AMBA baseada em computação reconfigurável.* Tese de Mestrado, Universidade de São Paulo, 2005.

[44] MAXFIELD, C. *The Design Warrior’s Guide to FPGAs - Devices, Tools and Flows.* Oxford, UK, Newnes, 2004.

[45] DESCHAMPS, J.-P., BIOUL, G. J. A., SUTTER, G. D. *Synthesis of Arithmetic Circuits - FPGA, ASIC and Embedded Systems.* New Jersey, John Wiley & Sons, 2006.

[46] HAUCK, S., DEHON, A. *Reconfigurable Computing.* Massachusetts, Morgan Kaufmann, 2008.

[47] COFER, R. C., HARDING, B. F. *Rapid System Prototyping with FPGAs.* Oxford, UK, Newnes, 2006.

[48] NURMI, J. *Processor Design. System-on-Chip Computing for ASICs and FPGAs.* Dordrecht, The Netherlands, Springer, 2007.

[49] HASSAN, H., ANIS, M. *Low-Power Design of Nanometer FPGAs.* Massachusetts, Morgan Kaufmann, 2010.

[50] MAXFIELD, C. *FPGAs World Class Designs.* Oxford, UK, Newnes, 2009.

[51] MAXFIELD, C. *FPGAs Instant Access.* Oxford, UK, Newnes, 2008.

[52] WOODS, R., MCALLISTER, J., LIGHTBODY, G., et al. *FPGA-based Implementation of Signal Processing Systems.* West Sussex, United Kingdom, John Wiley & Sons, 2008.

[53] TINDER, R. F. *Engineering Digital Design.* 2 ed. California, Academic Press, 2000.

[54] BAILEY, D. G. *Design for Embedded Image Processing on FPGAs.* Singapore, John Wiley & Sons, 2011.

[55] FLOYD, T. L. *Digital Fundamentals.* 9 ed. USA, Pearson Prentice Hall, 2006.

[56] KARRIS, S. T. *Digital Circuit Analysis and Design with an Introduction to CPLDs and FPGAs.* USA, Orchard Publications, 2005.

[57] ZEIDMAN, B. *Designing with FPGAs and CPLDs.* Kansas, CMP Books, 2002.

[58] *ADV7123 Triple 10-Bit High Speed Video DAC.* Analog Devices.

[59] "Evolução do Tilt Test Eletrônico (Sensores de 2 Eixos)". Pesquisa Naval, 2012.

[60] *Procedimentos para Alinhamento dos Sistemas de Armas.* Centro de Apoio a Sistemas Operativos, Marinha do Brasil, Rio de Janeiro, 2009.

[61] *DE2-70 Development and Education Board User Manual.* Terasic Technologies, 2009.

Apêndice A

Módulo Serial RS-232

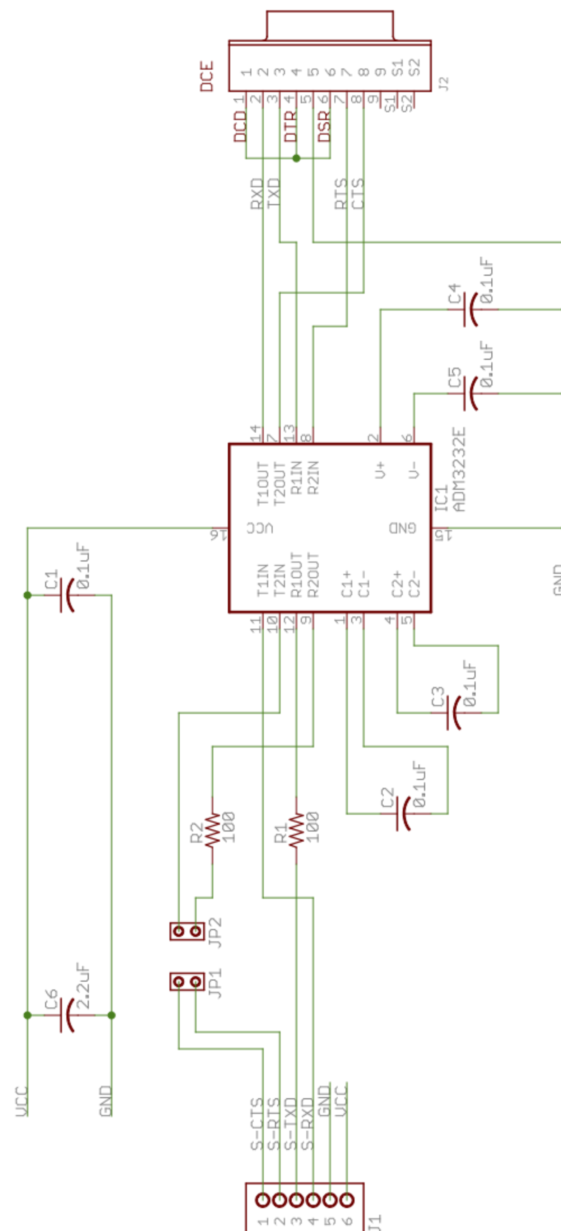


Figura A.1: Diagrama Esquemático do Módulo RS-232

Apêndice B

Controlador do Módulo SDRAM

```
-----------------------------------------------------------------------------------------------
--
--  UNIVERSIDADE FEDERAL DO RIO DE JANEIRO - UFRJ
--  INSTITUTO ALBERTO LUIZ COIMBRA DE POS-GRADUACAO E PESQUISA EM ENGENHARIA - COPPE
--  PROGRAMA DE ENGENHARIA ELETRICA - PEE
--
--  Mestrado em Microeletronica
--  2012,2013 - Alexandre Bazyl
--
--  CASoPC - CASOP System on Programmable Chip
--
-----------------------------------------------------------------------------------------------

--
-- Revision History
--
-- Version 0.2 11/04/2013 by Alexandre Bazyl
--     Adapted for IS42S16160B - 4M x 16 bits x 4 banks (found in the Terasic DE2-70)
--
-- Version 0.1 25/03/2013 by Alexandre Bazyl
--     Conversion to VHDL
--     Changed the interface
--     Changed clock to 100MHz
--
-- Original Verilog Version 03/02/2008 by lsilvest
-- Source code found in http://en.pudn.com/downloads150/sourcecode/embed/detail647139_en.html#
--

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

library casopc_lib;
use casopc_lib.casopc_pkg.all;
use casopc_lib.config_Pkg.all;

entity sdram_wb is port(
    clk_sdram    : in std_logic;              -- @ 100MHz
    clk_skewed   : in std_logic;              -- @ 100MHz - 3ns
    pll_locked   : in std_logic;
    wb_o         : out wb_slv_out_type;
    wb_i         : in wb_slv_in_type;
```

```
    sdram_o      : out sdram_out_type;
    sdram_io     : inout sdram_io_type
);
end sdram_wb;

architecture arch of sdram_wb is
    ----------------------------------------------------------------------------------------------
    -- CONSTANTS
    ----------------------------------------------------------------------------------------------
    -- M2-M0   - BURST LENGTH     (001 = 2)
    -- M3      - TYPE OF BURST    (  0 = sequential)
    -- M6-M4   - CAS LATENCY      (011 = 3)
    -- M8-M7   - OPERATION MODE   (  0 = standard operation)
    -- M9      - WRITE BURST MODE (  0 = programmed burst length)
    -- M12-M10 - RESERVED
    constant MODE_REGISTER   : std_logic_vector(SDRAM_ADDR_WIDTH - 1 downto 0) := "0000000110001";

    -- @ 100 MHz period is 10ns cycle
    -- tRC (Command Period REF-REF/ACT-ACT) = 67.5 ns => 67.5 / 10 => 7 cycles
    -- Time to wait after refresh and between two ACT commands
    constant TRC_CNTR_VALUE  : std_logic_vector(3 downto 0) := std_logic_vector(to_unsigned(7, 4));

    -- Need 8192 refreshes for every 64 ms
    -- So the # of cycles between refreshes is 64_000_000 / 8192 / 10 = 781
    constant RFSH_INT_CNTR_VALUE  : std_logic_vector(24 downto 0) :=
                                    std_logic_vector(to_unsigned(781, 25));

    -- tRCD (Active Command To Read / Write Command Delay Time)
    -- RAS to CAS delay = 20 ns => 2 cycles; will also be used for tRP and tRSC
    constant TRCD_CNTR_VALUE  : std_logic_vector(2 downto 0) := std_logic_vector(to_unsigned(2, 3));

    -- 200us => 20000
    constant WAIT_200us_CNTR_VALUE  : std_logic_vector(15 downto 0) :=
                                      std_logic_vector(to_unsigned(20000, 16));

    ----------------------------------------------------------------------------------------------
    -- TYPES
    ----------------------------------------------------------------------------------------------
    type parameter_state is (INIT_IDLE,     INIT_WAIT_200us, INIT_INIT_PRE, INIT_WAIT_PRE,
                             INIT_MODE_REG, INIT_WAIT_MODE_REG, INIT_DONE_ST);

    type cmd_state is (IDLE_ST, REFRESH_ST, REFRESH_WAIT_ST, ACT_ST,  WAIT_ACT_ST,
                       WRITE0_ST, WRITE1_ST, WRITE_PRE_ST,
                       READ0_ST, READ1_ST, READ2_ST, READ3_ST, READ4_ST,
                       READ_PRE_ST, PRE_ST, WAIT_PRE_ST);

    ----------------------------------------------------------------------------------------------
    -- SIGNALS
    ----------------------------------------------------------------------------------------------
    --
    -- REGISTERS
    --
    signal address_r            : std_logic_vector(22 downto 0);
    signal dram_addr_r          : std_logic_vector(SDRAM_ADDR_WIDTH - 1 downto 0);
    signal dram_bank_r          : std_logic_vector(1 downto 0);
    signal dram_dq_r            : std_logic_vector(SDRAM_DATA_WIDTH - 1 downto 0);
    signal dram_cas_n_r         : std_logic;
    signal dram_ras_n_r         : std_logic;
    signal dram_we_n_r          : std_logic;
```

```
    signal dat_o_r              : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);
    signal ack_o_r              : std_logic;
    signal dat_i_r              : std_logic_vector(CFG_DMEM_WIDTH - 1 downto 0);
    signal we_i_r               : std_logic;
    signal stb_i_r              : std_logic;
    signal oe_r                 : std_logic;

    signal current_state        : cmd_state;
    signal next_state           : cmd_state;
    signal current_init_state   : parameter_state;
    signal next_init_state      : parameter_state;

    signal init_done            : std_logic;
    signal init_pre_cntr        : std_logic_vector(3 downto 0);
    signal trc_cntr             : std_logic_vector(3 downto 0);
    signal rfsh_int_cntr        : std_logic_vector(24 downto 0);
    signal trcd_cntr            : std_logic_vector(2 downto 0);
    signal wait_200us_cntr      : std_logic_vector(15 downto 0);
    signal do_refresh           : std_logic;
    --
    -- ALIASES
    --
    alias clk_i                 : std_logic is clk_sdram;
    alias rst_i                 : std_logic is wb_i.rst_i;

begin
    ----------------------------------------------------------------------------------------------
    -- OUTPUTS
    ----------------------------------------------------------------------------------------------
    wb_o.dat_o       <= dat_o_r;
    wb_o.ack_o       <= ack_o_r;

    sdram_o.ADDR <= dram_addr_r;
    sdram_o.BA_0     <= dram_bank_r(0);
    sdram_o.BA_1     <= dram_bank_r(1);
    sdram_o.CAS_n    <= dram_cas_n_r;
    sdram_o.RAS_n    <= dram_ras_n_r;
    sdram_o.WE_n     <= dram_we_n_r;
    sdram_io.DQ      <= dram_dq_r when oe_r = '1' else (others => 'Z');
    sdram_o.CKE      <= '1';                     -- pll_locked
    sdram_o.CS_n     <= not(pll_locked);         -- chip select is always on in normal op
    sdram_o.CLK      <= clk_skewed;
    sdram_o.LDQM     <= '0';                     -- don't do byte masking
    sdram_o.UDQM     <= '0';                     -- don't do byte masking

    --
    -- Register the user command
    --
    process (clk_i)
    begin
        if (clk_i'event and clk_i = '1') then
            if (stb_i_r = '1' and current_state = ACT_ST) then
                stb_i_r   <= '0';
            elsif (wb_i.stb_i = '1' and wb_i.cyc_i = '1') then
                address_r <= wb_i.adr_i(23 downto 1);
                dat_i_r   <= wb_i.dat_i;
                we_i_r    <= wb_i.we_i;
                stb_i_r   <= wb_i.stb_i;
```

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

process (clk_i)
begin
    if(clk_i'event and clk_i = '1') then
        if (rst_i = '1') then
            wait_200us_cntr     <= (others => '0');
        else
            if (current_init_state = INIT_IDLE) then
                wait_200us_cntr <= WAIT_200us_CNTR_VALUE;
            else
                wait_200us_cntr <= std_logic_vector(unsigned(wait_200us_cntr) - 1);
            end if;
        end if;
    end if;
end process;

--
-- Control the interval between refreshes
--
process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if (rst_i = '1') then
            rfsh_int_cntr <= (others => '0');    -- immediately initiate new refresh on reset
        elsif (current_state = REFRESH_WAIT_ST) then
            do_refresh    <= '0';
            rfsh_int_cntr <= RFSH_INT_CNTR_VALUE;
        elsif (unsigned(rfsh_int_cntr) = 0) then
            do_refresh    <= '1';
        else
            rfsh_int_cntr <= std_logic_vector(unsigned(rfsh_int_cntr) - 1);
        end if;
    end if;
end process;

process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            trc_cntr <= (others => '0');
        elsif (current_state = PRE_ST OR current_state = REFRESH_ST) then
            trc_cntr <= TRC_CNTR_VALUE;
        else
            trc_cntr <= std_logic_vector(unsigned(trc_cntr) - 1);
        end if;
    end if;
end process;

--
-- Counter to control the activate
--
process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            trcd_cntr <= (others => '0');
```

```
        elsif (current_state = ACT_ST or current_init_state = INIT_INIT_PRE or
               current_init_state = INIT_MODE_REG) then
            trcd_cntr <= TRCD_CNTR_VALUE;
        else
            trcd_cntr <= std_logic_vector(unsigned(trcd_cntr) - 1);
        end if;
    end if;
end process;

process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            init_pre_cntr <= (others => '0');
        elsif (current_init_state = INIT_INIT_PRE) then
            init_pre_cntr <= std_logic_vector(unsigned(init_pre_cntr) + 1);
        end if;
    end if;
end process;

process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            init_done <= '0';
        elsif (current_init_state = INIT_DONE_ST) then
            init_done <= '1';
        end if;
    end if;
end process;

--
-- STATE CHANGE
--
process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            current_init_state <= INIT_IDLE;
        else
            current_init_state <= next_init_state;
        end if;
    end if;
end process;

process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            current_state <= IDLE_ST;
        else
            current_state <= next_state;
        end if;
    end if;
end process;

-- Initialization is fairly easy on this chip: wait 200us then issue
-- 8 precharges before setting the mode register
process(current_init_state)
```

```
begin
    case current_init_state is
        when INIT_IDLE =>
            if (init_done = '0') then
                next_init_state <= INIT_WAIT_200us;
            else
                next_init_state <= INIT_IDLE;
            end if;

        when INIT_WAIT_200us =>
            if (unsigned(wait_200us_cntr) = 0) then
                next_init_state <= INIT_INIT_PRE;
            else
                next_init_state <= INIT_WAIT_200us;
            end if;

        when INIT_INIT_PRE=>
            next_init_state <= INIT_WAIT_PRE;

        when INIT_WAIT_PRE =>
            if (unsigned(trcd_cntr) = 0) then -- this is tRP
                if (unsigned(init_pre_cntr) = 8) then
                    next_init_state <= INIT_MODE_REG;
                else
                    next_init_state <= INIT_INIT_PRE;
                end if;
            else
                next_init_state <= INIT_WAIT_PRE;
            end if;

        when INIT_MODE_REG =>
            next_init_state <= INIT_WAIT_MODE_REG;

        when INIT_WAIT_MODE_REG =>
            if (unsigned(trcd_cntr) = 0) then -- tRSC
                next_init_state <= INIT_DONE_ST;
            else
                next_init_state <= INIT_WAIT_MODE_REG;
            end if;

        when INIT_DONE_ST =>
            next_init_state <= INIT_IDLE;

        when others =>
            next_init_state <= INIT_IDLE;
    end case;
end process;

--
-- MAIN CONTROLLER LOGIC
--
process (current_state)
begin
    case current_state is
        when IDLE_ST =>
            if (init_done = '0') then
                next_state <= IDLE_ST;
            elsif (do_refresh = '1') then
                next_state <= REFRESH_ST;
```

```
    elsif (stb_i_r = '1') then
        next_state <= ACT_ST;
    else
        next_state <= IDLE_ST;
    end if;

when REFRESH_ST =>
    next_state <= REFRESH_WAIT_ST;

when REFRESH_WAIT_ST =>
    if (unsigned(trc_cntr) = 0) then
        next_state <= IDLE_ST;
    else
        next_state <= REFRESH_WAIT_ST;
    end if;

when ACT_ST =>
    next_state <= WAIT_ACT_ST;

when WAIT_ACT_ST =>
    if (unsigned(trcd_cntr) = 0) then
        if (we_i_r = '1') then
            next_state <= WRITE0_ST;
        else
            next_state <= READ0_ST;
        end if;
    else
        next_state <= WAIT_ACT_ST;
    end if;

when WRITE0_ST =>
    next_state <= WRITE1_ST;

when WRITE1_ST =>
    next_state <= WRITE_PRE_ST;

when WRITE_PRE_ST =>
    next_state <= PRE_ST;

when READ0_ST =>
    next_state <= READ1_ST;

when READ1_ST =>
    next_state <= READ2_ST;

when READ2_ST =>
    next_state <= READ3_ST;

when READ3_ST =>
    next_state <= READ4_ST;

when READ4_ST =>
    next_state <= READ_PRE_ST;

when READ_PRE_ST =>
    next_state <= PRE_ST;

when PRE_ST =>
    next_state <= WAIT_PRE_ST;
```

```
        when WAIT_PRE_ST =>
            -- if the next command was not another row activate in the same bank
            -- we could wait tRCD only; for simplicity but at the detriment of
            -- efficiency we always wait tRC
            if (unsigned(trc_cntr) = 0) then
                next_state <= IDLE_ST;
            else
                next_state <= WAIT_PRE_ST;
            end if;

        when others =>
            next_state <= IDLE_ST;
    end case;
end process;

--
-- ACK SIGNAL
--
process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            ack_o_r <= '0';
        elsif (current_state = READ_PRE_ST OR current_state = WRITE_PRE_ST) then
            ack_o_r <= '1';
        elsif (current_state = WAIT_PRE_ST) then
            ack_o_r <= '0';
        end if;
    end if;
end process;

--
-- DATA
--
process (clk_i)
begin
    if (clk_i'event and clk_i = '1') then
        if rst_i = '1' then
            dat_o_r     <= (others => '0');
            dram_dq_r   <= (others => '0');
            oe_r        <= '0';
        elsif (current_state = WRITE0_ST) then
            dram_dq_r   <= dat_i_r(31 downto 16);
            oe_r        <= '1';
        elsif (current_state = WRITE1_ST) then
            dram_dq_r   <= dat_i_r(15 downto 0);
            oe_r        <= '1';
        elsif (current_state = READ4_ST) then
            -- we should actually be reading this on READ3, but
            -- because of delay the data comes a cycle later...
            dat_o_r(31 downto 16) <= sdram_io.DQ;
            dram_dq_r   <= (others => 'Z');
            oe_r        <= '0';
        elsif (current_state = READ_PRE_ST) then
            dat_o_r(15 downto 0) <= sdram_io.DQ;
            dram_dq_r   <= (others => 'Z');
            oe_r        <= '0';
        else
```

```
                dram_dq_r   <= (others => 'Z');
                oe_r        <= '0';
            end if;
        end if;
    end process;

    --
    -- ADDRESS
    --
    process (clk_i)
    begin
        if (clk_i'event and clk_i = '1') then
            if (current_init_state = INIT_MODE_REG) then
                dram_addr_r  <= MODE_REGISTER;
            elsif (current_init_state = INIT_INIT_PRE) then
                dram_addr_r  <= "0010000000000";  -- precharge all (A10 set to 1)
            elsif (current_state = ACT_ST) then
                dram_addr_r  <= address_r(20 downto 8);
                dram_bank_r  <= address_r(22 downto 21);
            elsif (current_state = WRITE0_ST OR current_state = READ0_ST) then
                -- Enter column with bit A10 set to 1 indicating auto precharge
                -- A0-A8 (16 bits) provides the starting column location
                dram_addr_r  <= "0010" & address_r(8 downto 0);
                dram_bank_r  <= address_r(22 downto 21);
            else
                dram_addr_r  <= (others => '0');
                dram_bank_r  <= (others => '0');
            end if;
        end if;
    end process;

    --
    -- COMMANDS
    --
    process (clk_i)
    begin
        if (clk_i'event and clk_i = '1') then
            if (current_init_state = INIT_INIT_PRE OR current_init_state = INIT_MODE_REG OR
                current_state = REFRESH_ST OR current_state = ACT_ST) then
                dram_ras_n_r <= '0';
            else
                dram_ras_n_r <= '1';
            end if;
            if (current_state = READ0_ST OR current_state = WRITE0_ST OR
                current_state = REFRESH_ST OR current_init_state = INIT_MODE_REG) then
                dram_cas_n_r <= '0';
            else
                dram_cas_n_r <= '1';
            end if;
            if (current_init_state = INIT_INIT_PRE OR current_state = WRITE0_ST OR
                current_init_state = INIT_MODE_REG) then
                dram_we_n_r  <= '0';
            else
                dram_we_n_r  <= '1';
            end if;
        end if;
    end process;
end arch;
```

Apêndice C

Controlador do Módulo SSRAM

```
-----------------------------------------------------------------------------------------------
--
--  UNIVERSIDADE FEDERAL DO RIO DE JANEIRO - UFRJ
--  INSTITUTO ALBERTO LUIZ COIMBRA DE POS-GRADUACAO E PESQUISA EM ENGENHARIA - COPPE
--  PROGRAMA DE ENGENHARIA ELETRICA - PEE
--
--  Mestrado em Microeletronica
--  2012,2013 - Alexandre Bazyl
--
--  CASoPC - CASOP System on Programmable Chip
--
-----------------------------------------------------------------------------------------------

--
-- Revision History
--
-- Version 0.2 06/04/2013 by Alexandre Bazyl
--     Removed the WB interface, adjusted interface and output signals
--     Tested @200MHz
--
-- Version 0.1 05/04/2013 by Alexandre Bazyl
--     VHDL version
--     Removed Burst cycles for VGA and Video R/W requests
--     Adjusted the size of data path
--
-- Original Verilog version 20/12/2010 by Aleksander Osman (alfik@poczta.fm)
-- from The OpenCores aoOCS Soc (Amiga Original Chip Set - OCS)
-- http://opencores.org/project,aoocs
--

--
-- IS61LPS51236A 512K x 36
-- Synchronous pipelined SRAM driver with WISHBONE slave interface.
--

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

library casopc_lib;
use casopc_lib.casopc_pkg.all;
use casopc_lib.config_Pkg.all;
```

```
entity ssram is port(
    clk_i        : in std_logic;
    rst_i        : in std_logic;

    read_i       : in std_logic;
    write_i      : in std_logic;

    addr_i       : in std_logic_vector(SSRAM_ADDR_WIDTH + 1 downto 0);
    dat_i        : in std_logic_vector(SSRAM_DATA_WIDTH - 1 downto 0);

    dat_o        : out std_logic_vector(SSRAM_DATA_WIDTH - 1 downto 0);
    ack_o        : out std_logic;

    ssram_o      : out ssram_out_type;
    ssram_io     : inout ssram_io_type
);
end ssram;

architecture arch of ssram is

    ----------------------------------------------------------------------------------------------
    -- SIGNALS
    ----------------------------------------------------------------------------------------------

    --
    -- REGISTERS
    --
    signal reg_ack_o                  : std_logic;
    signal reg_dat_o                  : std_logic_vector(SSRAM_DATA_WIDTH - 1 downto 0);
    signal ssram_address              : std_logic_vector(SSRAM_ADDR_WIDTH - 1 downto 0);
    signal ssram_oe_n                 : std_logic;
    signal ssram_writeen_n            : std_logic;
    signal ssram_byteen_n             : std_logic_vector(3 downto 0);
    signal ssram_advance_n            : std_logic;
    signal ssram_adsc_n               : std_logic;
    signal ssram_data_oe              : std_logic;
    signal ssram_data_reg             : std_logic_vector(SSRAM_DATA_WIDTH - 1 downto 0);
    signal slv_sel                    : std_logic_vector(3 downto 0);

    type parameter is (S_IDLE,
                       S_R1, S_R2, S_R3,
                       S_PRE_IDLE);

    signal state                      : parameter;

begin
    ----------------------------------------------------------------------------------------------
    -- ASSIGN OUTPUTS
    ----------------------------------------------------------------------------------------------
    ack_o               <= reg_ack_o;
    dat_o               <= reg_dat_o;

    ssram_o.GLOBAL_WE_n  <= '1';
    ssram_o.ADSP_n       <= '1';
    ssram_o.CE1_n        <= '0';
    ssram_o.CE2          <= '1';
    ssram_o.CE3_n        <= '0';
    ssram_o.PARITY       <= (others => '0');
```

```
ssram_o.ADDR         <= ssram_address;
ssram_o.ADSC_n       <= ssram_adsc_n;
ssram_o.ADV_n        <= ssram_advance_n;
ssram_o.OE_n         <= ssram_oe_n;
ssram_o.WE_n         <= ssram_writeen_n;
ssram_o.BYTE_WE_n    <= ssram_byteen_n;

ssram_io.DQ  <= ssram_data_reg when ssram_data_oe = '1' else (others => 'Z');

slv_sel      <= (others => write_i);

process(clk_i, rst_i)
begin
    if clk_i'event and clk_i = '1' then
        if rst_i = '1' then
            ssram_address    <= (others => '0');
            ssram_adsc_n     <= '1';
            ssram_advance_n  <= '1';
            ssram_data_reg   <= (others => '0');
            ssram_data_oe    <= '0';
            ssram_oe_n       <= '1';
            ssram_writeen_n  <= '1';
            ssram_byteen_n   <= "1111";

            reg_ack_o        <= '0';
            reg_dat_o        <= (others => '0');

            state <= S_IDLE;
        else
            case state is

                when S_IDLE =>
                    reg_ack_o <= '0';

                    if (reg_ack_o = '0' and read_i = '1') then
                        -- address and byte enables output
                        ssram_address    <= addr_i(SSRAM_ADDR_WIDTH + 1 downto 2);
                        ssram_adsc_n    <= '0';
                        ssram_advance_n <= '1';
                        ssram_data_reg  <= (others => '0');
                        ssram_data_oe   <= '0';
                        ssram_oe_n      <= '1';
                        ssram_writeen_n <= '1';
                        ssram_byteen_n  <= "0000";

                        state <= S_R1;

                    elsif (reg_ack_o = '0' and write_i = '1') then
                        -- address, byte enables and write enables output
                        ssram_address   <= addr_i(SSRAM_ADDR_WIDTH + 1 downto 2);
                        ssram_adsc_n    <= '0';
                        ssram_advance_n <= '1';
                        ssram_data_reg  <= dat_i;
                        ssram_data_oe   <= '1';
                        ssram_oe_n      <= '1';
                        ssram_writeen_n <= '0';
                        ssram_byteen_n  <= not(slv_sel);
```

```
                            reg_ack_o       <= '1';

                            state <= S_PRE_IDLE;

                        end if;

                    when S_R1 =>
                        -- address and byte enables latched
                        ssram_adsc_n       <= '1';
                        ssram_advance_n    <= '1';

                        state <= S_R2;

                    when S_R2 =>
                        -- output enable
                        ssram_oe_n <= '0';

                        state <= S_R3;

                    when S_R3 =>
                        reg_dat_o          <= ssram_io.DQ(SSRAM_DATA_WIDTH - 1 downto 0);
                        reg_ack_o          <= '1';

                        ssram_address      <= (others => '0');
                        ssram_adsc_n       <= '1';
                        ssram_advance_n    <= '1';
                        ssram_data_reg     <= (others => '0');
                        ssram_data_oe      <= '0';
                        ssram_oe_n         <= '1';
                        ssram_writeen_n    <= '1';
                        ssram_byteen_n     <= "1111";

                        state <= S_IDLE;

                    when S_PRE_IDLE =>
                        reg_ack_o         <= '0';

                        ssram_address     <= (others => '0');
                        ssram_adsc_n      <= '1';
                        ssram_advance_n   <= '1';
                        ssram_data_reg    <= (others => '0');
                        ssram_data_oe     <= '0';
                        ssram_oe_n        <= '1';
                        ssram_writeen_n   <= '1';
                        ssram_byteen_n    <= "1111";

                        state <= S_IDLE;

                end case;
            end if;
        end if;
    end process;
end arch;
```

Apêndice D

Controlador do *Touch Panel*

```
--------------------------------------------------------------------------------
--
--  UNIVERSIDADE FEDERAL DO RIO DE JANEIRO - UFRJ
--  INSTITUTO ALBERTO LUIZ COIMBRA DE POS-GRADUACAO E PESQUISA EM ENGENHARIA - COPPE
--  PROGRAMA DE ENGENHARIA ELETRICA - PEE
--
--  Mestrado em Microeletronica
--  2012,2013 - Alexandre Bazyl
--
--  CASoPC - CASOP System on Programmable Chip
--
-------------------------------------------------------------------------------

--
-- Revision History
--
-- Version 0.1 28/03/2013 by Alexandre Bazyl
--
-- Original version 0.01 13/03/2009 by Tobias Rudloff
--

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

entity adc_spi_controller  is
    generic (
        SYS_CLK_FREQ     : integer
    );
    port(
        clk_i            : in std_logic;
        rst_n            : in std_logic;
        ADC_DIN_o        : out std_logic;
        ADC_DCLK_o       : out std_logic;
        ADC_CS_o         : out std_logic;
        ADC_DOUT_i       : in std_logic;
        ADC_BUSY_i       : in std_logic;      -- NOT USED
        ADC_PENIRQ_n_i   : in std_logic;
        TOUCH_IRQ_o      : out std_logic;
        TOUCH_X_o        : out std_logic_vector(11 downto 0);
        TOUCH_Y_o        : out std_logic_vector(11 downto 0)
```

```
    );
end adc_spi_controller;

architecture arch of adc_spi_controller is
    -----------------------------------------------------------------------------------------------
    -- CONSTANTS
    -----------------------------------------------------------------------------------------------
    constant SYSCLK_FRQ     : integer := SYS_CLK_FREQ * 1000000;        -- 50 * 1_000_000 (50 MHz)
    constant ADC_DCLK_FRQ   : integer:= 1000;                           -- 10kHz
    constant ADC_DCLK_CNT   : integer:= SYSCLK_FRQ/(ADC_DCLK_FRQ*2);

    -----------------------------------------------------------------------------------------------
    -- SIGNALS
    -----------------------------------------------------------------------------------------------
    signal d1_PENIRQ_n      : std_logic;
    signal d2_PENIRQ_n      : std_logic;
    signal dclk_cnt         : std_logic_vector(15 downto 0);
    signal transmit_en      : std_logic;
    signal spi_ctrl_cnt     : std_logic_vector( 7 downto 0);
    signal mdclk            : std_logic;
    signal pre_mdclk        : std_logic := '0';
    signal x_config_reg     : std_logic_vector( 7 downto 0);
    signal y_config_reg     : std_logic_vector( 7 downto 0);
    signal ctrl_reg         : std_logic_vector( 7 downto 0);
    signal mdata_in         : std_logic_vector( 7 downto 0);
    signal eof_transmition  : std_logic;
    signal x_coordinate     : std_logic_vector(11 downto 0);
    signal y_coordinate     : std_logic_vector(11 downto 0);
    signal load_data        : std_logic := '0';

begin
    x_config_reg <= X"92";
    y_config_reg <= X"D2";

    -----------------------------------------------------------------------------------------------
    -- PEN IRQ DETECT
    -----------------------------------------------------------------------------------------------
    process(rst_n, clk_i)
    begin
        if (rst_n = '0') then
            transmit_en <= '0';
            d1_PENIRQ_n <= '0';
            d2_PENIRQ_n <= '0';

        elsif (clk_i'event and clk_i = '1') then
            d1_PENIRQ_n <= ADC_PENIRQ_n_i;
            d2_PENIRQ_n <= d1_PENIRQ_n;

            if (((ADC_PENIRQ_n_i = '0' and d1_PENIRQ_n = '1') or
                 (ADC_PENIRQ_n_i = '0' and d2_PENIRQ_n = '0')) and
                  eof_transmition = '0') then
                transmit_en <= '1';
                load_data   <= '1';
            elsif (spi_ctrl_cnt = X"37") then
                transmit_en <= '0';
            else
                load_data   <= '0';
            end if;
        end if;
```

```
end process;

process(rst_n, clk_i)
begin
    if (rst_n = '0') then
        dclk_cnt <= (others => '0');
        mdclk    <= '0';
    elsif (clk_i'event and clk_i = '1') then
        if (transmit_en = '1' and spi_ctrl_cnt < X"37") then
            if (dclk_cnt = ADC_DCLK_CNT) then
                mdclk    <= not mdclk;
                dclk_cnt <= (others => '0');
            else
                dclk_cnt <= dclk_cnt + '1';
            end if;
        else
            dclk_cnt <= (others => '0');
            mdclk    <= '0';
        end if;
    end if;
end process;

process(rst_n, clk_i)
begin
    if (rst_n = '0') then
        mdata_in        <= (others => '0');
        x_coordinate    <= (others => '0');
        y_coordinate    <= (others => '0');
        pre_mdclk       <= '0';
        spi_ctrl_cnt    <= (others => '0');
        eof_transmition <= '0';
    elsif (clk_i'event and clk_i = '1') then
        ADC_CS_o        <= transmit_en;
        ADC_DIN_o       <= mdata_in(7);
        ADC_DCLK_o      <= mdclk;
        pre_mdclk       <= mdclk;

        if (load_data = '1') then
            ctrl_reg        <= x_config_reg;
            eof_transmition <= '1';
        else
            if (spi_ctrl_cnt = X"37") then
                spi_ctrl_cnt    <= (others => '0');
                eof_transmition <= '0';
            end if;

            if (mdclk = '0' and pre_mdclk = '1') then
                spi_ctrl_cnt <= spi_ctrl_cnt + '1';

                if (spi_ctrl_cnt >= X"08" and spi_ctrl_cnt <= X"0F") then      -- send_x_reg
                    ctrl_reg  <= ctrl_reg(6 downto 0) & '0';
                    mdata_in  <= ctrl_reg;
                elsif (spi_ctrl_cnt >= X"20" and spi_ctrl_cnt <= X"27") then   -- send_y_reg
                    ctrl_reg  <= ctrl_reg(6 downto 0) & '0';
                    mdata_in  <= ctrl_reg;
                elsif (spi_ctrl_cnt = X"35") then
                    TOUCH_IRQ_o <= '1';
                    TOUCH_X_o <= x_coordinate;
                    TOUCH_Y_o <= y_coordinate;
```

```
                else
                    mdata_in <= X"00";
                    TOUCH_IRQ_o <= '0';
                end if;
            end if;

            if (mdclk = '1' and pre_mdclk = '0') then
                if (spi_ctrl_cnt >= X"11" and spi_ctrl_cnt <= X"1C" ) then
                    x_coordinate <= x_coordinate(10 downto 0) & ADC_DOUT_i;
                    ctrl_reg     <= y_config_reg;
                elsif (spi_ctrl_cnt >= X"29" and spi_ctrl_cnt <= X"34") then
                    y_coordinate <= y_coordinate(10 downto 0) & ADC_DOUT_i;
                end if;
            end if;
        end if;
    end if;
end process;
end arch;
```

Apêndice E

Controlador do LCD 16x2

```
--------------------------------------------------------------------------------
--
--  UNIVERSIDADE FEDERAL DO RIO DE JANEIRO - UFRJ
--  INSTITUTO ALBERTO LUIZ COIMBRA DE POS-GRADUACAO E PESQUISA EM ENGENHARIA - COPPE
--  PROGRAMA DE ENGENHARIA ELETRICA - PEE
--
--  Mestrado em Microeletronica
--  2012,2013 - Alexandre Bazyl
--
--  CASoPC - CASOP System on Programmable Chip
--
-------------------------------------------------------------------------------

--
-- Revision History
--
-- Version 1.0.1 03/01/2013 by Alexandre Bazyl
-- Tested (stable) version
--
-- Original version 1.0 28/07/2012 by Stachelsau (stachelsau@T420)
-- from http://opencores.org/project,16x2_lcd_controller
--

library ieee;
use ieee.std_logic_1164.all;

entity lcd16x2_ctrl is
    generic (
        CLK_PERIOD_NS  : positive := 20     -- 50MHz
    );
    port (
        clk_i           : in std_logic;
        rst_i           : in std_logic;
        lcd_en          : out std_logic;
        lcd_rs          : out std_logic;
        lcd_rw          : out std_logic;
        lcd_db          : out std_logic_vector(7 downto 4);
        line1_buffer    : in std_logic_vector(127 downto 0);  -- 16 x 8-bits
        line2_buffer    : in std_logic_vector(127 downto 0)
    );
end entity lcd16x2_ctrl;
```

```
architecture rtl of lcd16x2_ctrl is

    ------------------------------------------------------------------------------------------------
    -- CONSTANTS
    ------------------------------------------------------------------------------------------------
    constant DELAY_15_MS        : positive :=   15 * 10**6 / CLK_PERIOD_NS + 1;
    constant DELAY_1640_US      : positive := 1640 * 10**3 / CLK_PERIOD_NS + 1;
    constant DELAY_4100_US      : positive := 4100 * 10**3 / CLK_PERIOD_NS + 1;
    constant DELAY_100_US       : positive :=  100 * 10**3 / CLK_PERIOD_NS + 1;
    constant DELAY_40_US        : positive :=   40 * 10**3 / CLK_PERIOD_NS + 1;

    constant DELAY_NIBBLE       : positive := 10**3 / CLK_PERIOD_NS + 1;
    constant DELAY_LCD_EN       : positive :=   230 / CLK_PERIOD_NS + 1;
    constant DELAY_SETUP_HOLD   : positive :=    40 / CLK_PERIOD_NS + 1;

    constant MAX_DELAY          : positive := DELAY_15_MS;

    ------------------------------------------------------------------------------------------------
    -- TYPES
    ------------------------------------------------------------------------------------------------

    -- this record describes one write operation
    type op_t is record
        rs      : std_logic;
        data    : std_logic_vector(7 downto 0);
        delay_h : integer range 0 to MAX_DELAY;
        delay_l : integer range 0 to MAX_DELAY;
    end record op_t;

    type op_state_t is (IDLE,
                        WAIT_SETUP_H, ENABLE_H, WAIT_HOLD_H, WAIT_DELAY_H,
                        WAIT_SETUP_L, ENABLE_L, WAIT_HOLD_L, WAIT_DELAY_L,
                        DONE);

    type state_t is (RESET, CONFIG,
                     SELECT_LINE1, WRITE_LINE1,
                     SELECT_LINE2, WRITE_LINE2);

    constant default         : op_t := (rs => '1', data => X"00",
                                        delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US);
    constant op_select_line1 : op_t := (rs => '0', data => X"80",
                                        delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US);
    constant op_select_line2 : op_t := (rs => '0', data => X"C0",
                                        delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US);

    -- init + config operations:
    -- write 3 x 0x3 followed by 0x2
    -- function set command
    -- entry mode set command
    -- display on/off command
    -- clear display
    type config_ops_t is array(0 to 5) of op_t;
    constant config_ops : config_ops_t := (
            5 => (rs => '0', data => X"33", delay_h => DELAY_4100_US, delay_l => DELAY_100_US),
            4 => (rs => '0', data => X"32", delay_h => DELAY_40_US, delay_l => DELAY_40_US),
            3 => (rs => '0', data => X"28", delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
            2 => (rs => '0', data => X"06", delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
            1 => (rs => '0', data => X"0C", delay_h => DELAY_NIBBLE, delay_l => DELAY_40_US),
            0 => (rs => '0', data => X"01", delay_h => DELAY_NIBBLE, delay_l => DELAY_1640_US)
```

```
    );

    ----------------------------------------------------------------------------------------------
    -- SIGNALS
    ----------------------------------------------------------------------------------------------
    signal this_op        : op_t;
    signal op_state       : op_state_t := DONE;
    signal next_op_state  : op_state_t;
    signal cnt            : natural range 0 to MAX_DELAY;
    signal next_cnt       : natural range 0 to MAX_DELAY;

    signal state          : state_t := RESET;
    signal next_state     : state_t;
    signal ptr            : natural range 0 to 15 := 0;
    signal next_ptr       : natural range 0 to 15;

begin

    proc_state : process(state, op_state, ptr, line1_buffer, line2_buffer) is
    begin
        case state is
            when RESET =>
                this_op    <= default;
                next_state <= CONFIG;
                next_ptr   <= config_ops_t'high;

            when CONFIG =>
                this_op    <= config_ops(ptr);
                next_ptr   <= ptr;
                next_state <= CONFIG;
                if op_state = DONE then
                    next_ptr <= ptr - 1;
                    if ptr = 0 then
                        next_state <= SELECT_LINE1;
                    end if;
                end if;

            when SELECT_LINE1 =>
                this_op  <= op_select_line1;
                next_ptr <= 15;
                if op_state = DONE then
                    next_state <= WRITE_LINE1;
                else
                    next_state <= SELECT_LINE1;
                end if;

            when WRITE_LINE1 =>
                this_op      <= default;
                this_op.data <= line1_buffer(ptr*8 + 7 downto ptr*8);
                next_ptr     <= ptr;
                next_state   <= WRITE_LINE1;
                if op_state = DONE then
                    next_ptr <= ptr - 1;
                    if ptr = 0 then
                        next_state <= SELECT_LINE2;
                    end if;
                end if;

            when SELECT_LINE2 =>
```

```
            this_op  <= op_select_line2;
            next_ptr <= 15;
            if op_state = DONE then
                next_state <= WRITE_LINE2;
            else
                next_state <= SELECT_LINE2;
            end if;

        when WRITE_LINE2 =>
            this_op      <= default;
            this_op.data <= line2_buffer(ptr*8 + 7 downto ptr*8);
            next_ptr     <= ptr;
            next_state   <= WRITE_LINE2;
            if op_state = DONE then
                next_ptr <= ptr - 1;
                if ptr = 0 then
                    next_state <= SELECT_LINE1;
                end if;
            end if;
    end case;
end process proc_state;

reg_state : process(clk_i)
begin
    if rising_edge(clk_i) then
        if rst_i = '1' then
            state <= RESET;
            ptr   <= 0;
        else
            state <= next_state;
            ptr   <= next_ptr;
        end if;
    end if;
end process reg_state;

-- we never read from the lcd
lcd_rw <= '0';

--lcd_hw.lcd_on <= display_on_off;

proc_op_state : process(op_state, cnt, this_op) is
begin
    case op_state is
        when IDLE =>
            lcd_db <= (others => '0');
            lcd_rs <= '0';
            lcd_en <= '0';
            next_op_state <= WAIT_SETUP_H;
            next_cnt      <= DELAY_SETUP_HOLD;

        when WAIT_SETUP_H =>
            lcd_db <= this_op.data(7 downto 4);
            lcd_rs <= this_op.rs;
            lcd_en <= '0';
            if cnt = 0 then
                next_op_state <= ENABLE_H;
                next_cnt      <= DELAY_LCD_EN;
            else
                next_op_state <= WAIT_SETUP_H;
```

```
            next_cnt      <= cnt - 1;
        end if;

    when ENABLE_H =>
        lcd_db <= this_op.data(7 downto 4);
        lcd_rs <= this_op.rs;
        lcd_en <= '1';
        if cnt = 0 then
            next_op_state <= WAIT_HOLD_H;
            next_cnt      <= DELAY_SETUP_HOLD;
        else
            next_op_state <= ENABLE_H;
            next_cnt      <= cnt - 1;
        end if;

    when WAIT_HOLD_H =>
        lcd_db <= this_op.data(7 downto 4);
        lcd_rs <= this_op.rs;
        lcd_en <= '0';
        if cnt = 0 then
            next_op_state <= WAIT_DELAY_H;
            next_cnt      <= this_op.delay_h;
        else
            next_op_state <= WAIT_HOLD_H;
            next_cnt      <= cnt - 1;
        end if;

    when WAIT_DELAY_H =>
        lcd_db <= (others => '0');
        lcd_rs <= '0';
        lcd_en <= '0';
        if cnt = 0 then
            next_op_state <= WAIT_SETUP_L;
            next_cnt      <= DELAY_SETUP_HOLD;
        else
            next_op_state <= WAIT_DELAY_H;
            next_cnt      <= cnt - 1;
        end if;

    when WAIT_SETUP_L =>
        lcd_db <= this_op.data(3 downto 0);
        lcd_rs <= this_op.rs;
        lcd_en <= '0';
        if cnt = 0 then
            next_op_state <= ENABLE_L;
            next_cnt      <= DELAY_LCD_EN;
        else
            next_op_state <= WAIT_SETUP_L;
            next_cnt      <= cnt - 1;
        end if;

    when ENABLE_L =>
        lcd_db <= this_op.data(3 downto 0);
        lcd_rs <= this_op.rs;
        lcd_en <= '1';
        if cnt = 0 then
            next_op_state <= WAIT_HOLD_L;
            next_cnt      <= DELAY_SETUP_HOLD;
        else
```

```
                next_op_state <= ENABLE_L;
                next_cnt      <= cnt - 1;
            end if;

        when WAIT_HOLD_L =>
            lcd_db <= this_op.data(3 downto 0);
            lcd_rs <= this_op.rs;
            lcd_en <= '0';
            if cnt = 0 then
                next_op_state <= WAIT_DELAY_L;
                next_cnt      <= this_op.delay_l;
            else
                next_op_state <= WAIT_HOLD_L;
                next_cnt      <= cnt - 1;
            end if;

        when WAIT_DELAY_L =>
            lcd_db <= (others => '0');
            lcd_rs <= '0';
            lcd_en <= '0';
            if cnt = 0 then
                next_op_state <= DONE;
                next_cnt      <= 0;
            else
                next_op_state <= WAIT_DELAY_L;
                next_cnt      <= cnt - 1;
            end if;

        when DONE =>
            lcd_db <= (others => '0');
            lcd_rs <= '0';
            lcd_en <= '0';
            next_op_state <= IDLE;
            next_cnt      <= 0;
    end case;
end process proc_op_state;

reg_op_state : process (clk_i) is
begin
    if rising_edge(clk_i) then
        if state = RESET then
            op_state <= IDLE;
        else
            op_state <= next_op_state;
            cnt      <= next_cnt;
        end if;
    end if;
end process reg_op_state;
end architecture rtl;
```