Universidade de São Paulo

Escola de Engenharia de São Carlos
Departamento de Engenharia Elétrica

Guilherme Henrique Renó Jorge

Arquitetura para extração de características invariantes em imagens binárias utilizando dispositivos de lógica programável complexa

São Carlos
2006

Universidade de São Paulo

Escola de Engenharia de São Carlos
Departamento de Engenharia Elétrica

Arquitetura para extração de características invariantes em imagens binárias utilizando dispositivos de lógica programável complexa

Guilherme Henrique Renó Jorge

Dissertação apresentada à Escola de Engenharia de São Carlos da Universidade de São Paulo, para a obtenção do Título de Mestre em Engenharia Elétrica

ORIENTADOR : Prof. Dr. Valentin O. Roda

São Carlos
2006

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

J82a

Jorge, Guilherme Henrique Renó
Arquitetura para extração de características invariantes em imagens binárias utilizando dispositivos de lógica programável complexa / Guilherme Henrique Renó Jorge ; orientador Valentin O. Roda. -- São Carlos, 2006.

Dissertação (Mestrado-Programa de Pós-Graduação em Engenharia Elétrica. Área de Concentração: Processamento de Sinais e Instrumentação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2006.

1. FPGAs. 2. *Soft core*. 3. Momentos invariantes. 4. Padrão Wishbone. 5. PCI. I. Título.

FOLHA DE JULGAMENTO

Candidato: Bacharel **GUILHERME HENRIQUE RENO JORGE**

Dissertação defendida e julgada em 17-08-2006 perante a Comissão Julgadora:

aprovado

Prof. Associado **VALENTIN OBAC RODA (Orientador)**
(Escola de Engenharia de São Carlos/USP)

Aprovado

Prof. Dr. **CARLOS DIAS MACIEL**
(Escola de Engenharia de São Carlos/USP)

Aprovado

Prof. Associado **EDUARDO MARQUES**
(Instituto de Ciências Matemáticas e de Computação/USP)

Prof. Associado **HOMERO SCHIABEL**
Coordenador do Programa de Pós-Graduação em
em Engenharia Elétrica

Profa. Titular **MARIA DO CARMO CALIJURI**
Presidente da Comissão de Pós-Graduação da EESC

DEDICATÓRIA

Aos meus pais, Ailton e Sônia, pelo apoio incondicional durante todos esses anos.

A Deus, pela força nos momentos de fraqueza.

AGRADECIMENTOS

Ao professor Dr. Valentin Obac Roda, pelo apoio e paciência incondicionais.

Ao meu amigo Alex Ayres Stavarengo, pelos anos de companheirismo e amizade nessa caminhada conjunta.

A minha noiva Priscila de Almeida Silva, pelo amor, carinho e compreensão nessa jornada em que à distância e a saudade tanto nos castigaram.

Aos companheiros do Laboratório de Instrumentação Virtual e Microprocessada da USP de São Carlos.

A todos que direta ou indiretamente colaboraram para elaboração desse trabalho.

RESUMO

Os projetistas de sistemas digitais enfrentam sempre o desafio de encontrar o balanço correto entre velocidade e generalidade de processamento de seu *hardware*. Originalmente dispositivos de lógica programável de alta densidade como FPGAs (*Field Programable Gate Arrays*) e CPLDs (*Complex Logic Programmable Devices*) vinham sendo utilizados como dispositivos de "lógica acoplada"(*glue logic*), reduzindo significantemente o numero de componentes em um sistema. Seu uso como forma de substituir arquiteturas já existentes de microcontroladores e microprocessadores já é uma realidade. A representação e reconhecimento de objetos em imagens de duas dimensões em é um tópico importante. Uma forma comum de se fazer à representação de um objeto ou uma imagem é a utilização de momentos da função de intensidade de um grupo de *pixels*. Devido ao alto custo computacional para o calculo desses momentos tem sido importante a busca por arquiteturas que de alguma forma agilizem o calculo dos mesmos. Um problema enfrentado por arquiteturas desenvolvidas atualmente para trabalhar em forma de periférico com um computador pessoal (PC) ou uma estação de trabalho é a velocidade do barramento de transferência de dados. Interfaces de uso mais simples, como USB (*Universal serial bus*) ou *Ethernet*, têm sua taxa de transferência na casa dos Megabytes por segundo. Uma solução para esse problema é o uso do barramento PCI, as transferências feitas nesse barramento podem chegar à casa dos Gigabytes por segundo. Esse trabalho vem apresentar uma arquitetura, em forma de *soft core* totalmente compatível com o padrão *Wishbone*, para a extração de características invariantes em imagens binárias utilizando-se de dispositivos de lógica programável complexa. Desse modo torna-se possível o uso do barramento PCI para a transmissão de dados para um microcomputador ou uma estação de trabalho.

Palavras-chave: FPGAs, soft core, Momentos Invariantes, Padrão Wishbone, PCI.

ABSTRACT

A challenge for digital systems designers is to meet the balance between speed and flexibility was always. FPGAs and CPLDs where used as “glue logic”, reducing the number of components in a system. The use of programmable logic (CPLDs and FPGAs) as an alternative to microcontrollers and microprocessors is a real issue. Moments of the intensity function of a group of pixels have been used for the representation and recognition of objects in two dimensional images. Due to the high cost of computing the moments, the search for faster computing architectures is very important. A problem faced by nowadays developed architectures is the speed of computer communication buses. Simpler interfaces, as USB (Universal Serial Bus) and Ethernet, have their transfer rate in Megabytes per second. A solution for this problem is the use the PCI bus, where the transfer rate can achieve Gigabytes per second. This work presents a *soft core* architecture, fully compatible with the Wishbone standard, for the extraction of invariant characteristics from binary images using logic programmable devices.

Keywords: FPGAs, soft core, Moment invariants, Wishbone Standard, PCI

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASICS – *Application Specific Integrated Circuits*

BAR – *Base Address Register*

BIST – *Built In Self-Test*

BIOS – *Basic Input Output System*

CAD – *Computer Aided Design*

CCD – *Charge-Coupled Device*

CLB – *Configurable Logic Block*

CMOS – *Complementary Metal Oxide Semiconductor*

CPLD – *Complex Programmable Logic Device*

EEPROM – *Electrically Erasable Programmable Read Only Memory*

EPROM – *Erasable Programmable Read Only Memory*

FIFO – *First In First Out*

FPGA – *Field Programmable Gate Array*

HDL – *Hardware Description Language*

JTAG – *Joint Test Action Group*

LAB – *Logic Array Block*

LUT – *Look-up Table*

PAL – *Programmable Array Logic*

PCI – *Peripheral Component Interconnect*

PIA – *Programmable Interconnect Array*

PLA – *Programmable Logic Array*

PLD – *Programmable Logic Device*

PLL – *Phase-Locked Loop*

PROM – *Programmable Read-Only Memory*

ROM – *Read Only Memory*

SDRAM – *Synchronous Dynamic Random Access Memory*

SMD – *Superficial monting device*

SOC – *System on Chip*

SPLD – *Simple PLD*

SRAM – *Static RAM*

VHDL – *VHSIC Hardware Description Languange*

VLSI – *Very Large Scale Integration*

VHSIC – *Very High Speed Integreted Circuit*

Sumário

1. Introdução

Os projetistas de sistemas digitais enfrentam sempre o desafio de encontrar o balanço correto entre velocidade e generalidade de processamento de seu *hardware.* É possível desenvolver um *chip* genérico que realiza muitas funções diferentes, porém com sacrifício de desempenho (por exemplo: microprocessadores), ou *chips* dedicados a aplicações específicas, estes com uma velocidade muitas vezes superior aos *chips* genéricos. Circuitos Integrados de Aplicação Específica (*ASICs*), têm como características a ocupação mínima de área de silício, alto custo em relação aos *chips* genéricos, rapidez e um menor consumo de potência comparados com processadores programáveis. Um fator importante na escolha entre versatilidade e velocidade é o custo. Um *ASIC* executa a função para qual foi concebido de uma forma otimizada, porém uma vez "queimado" o *chip*, alterações na funcionalidade do circuito integrado não são possíveis. Logo, todo esforço despendido no seu projeto e implementação deve ser amortizado manufaturando um número elevado de unidades (1).

Originalmente dispositivos de lógica programável de alta densidade como FPGAs (*Field Programable Gate Arrays*) e CPLDs (*Complex Logic Programmable Devices*) vinham sendo utilizados como dispositivo de “lógica acoplada”(*glue logic*), reduzindo significantemente o numero de componentes em um sistema, e diminuindo os riscos de desenvolvimento de projetos, ao mesmo tempo em que proporcionava a flexibilidade para correções e atualizações de forma rápida e mais segura. Agora estes dispositivos são utilizados no projeto e controle de inúmeros sistemas com aplicações específicas, significativamente complexas, e dedicadas (2).

Durante os últimos anos os dispositivos de lógica programável de alta densidade vem sendo usados nas mais diversas áreas (3). Sua evolução e possíveis aplicações para essa tecnologia poderosa, que vem ganhando cada vez mais força no mundo científico e comercial, são um assunto cada vez mais explorado e discutido.(4)(5).

Por outro lado, o uso de tal tecnologia vem constantemente sendo associado à procura de problemas específicos, com o objetivo de otimizar o *hardware* (6). Seu uso como forma de substituir arquiteturas já existentes de microcontroladores e microprocessadores já é algo que ganha força devido ao grande ganho de desempenho que o hardware dedicado vem mostrando frente ao de propósito geral e o baixo custo frente aos *ASICs*.

Já na área de Visão Computacional, a representação e reconhecimento de objetos em imagens de duas dimensões em é um tópico importante. Uma forma comum de se fazer a representação de um objeto ou uma imagem é a utilização de momentos da função de intensidade de um grupo de *pixels*. Devido ao alto custo computacional para o cálculo desses momentos tem sido importante a busca por arquiteturas que de alguma forma agilizem o cálculo dos mesmos. (7)(8)(9)(10). Isso foi de certa forma beneficiado com o advento dos dispositivos de lógica programável complexa. A flexibilidade dos mesmos permite o desenvolvimento e simulação de novas arquiteturas a um custo acessível nos dias de hoje.

Um problema enfrentado por arquiteturas desenvolvidas atualmente para trabalhar em forma de periférico com um computador pessoal (PC) ou uma estação de trabalho é a velocidade do barramento de transferência de dados. Grande parte das arquiteturas busca extrair o máximo de eficiência dos dispositivos de lógica programável, e acabam caindo em limitações do próprio PC. Interfaces seriais de uso mais simples, como USB (Universal serial bus) ou *Ethernet*, têm sua taxa de transferência na casa dos *Megabytes* por segundo. Uma solução conseguir altas taxas de transferência de dados é o uso de barramentos paralelos de alta velocidade, entre esses barramentos se destaca o barramento PCI(11) para uso em microcomputadores.

As transferências feitas no barramento PCI podem chegar à casa dos *Gigabytes* por segundo, tornando o gargalo de dados muito menor e permitindo assim uma exploração maior dos limites dos FPGAs.

Entretanto, o uso do barramento PCI ainda é algo proibitivo no meio acadêmico, devido ao alto custo das plataformas de desenvolvimento disponíveis no mercado. Uma alternativa a essas plataformas é a plataforma para desenvolvimento PCI desenvolvida e distribuída de forma gratuita pela Universidad de la Republica (Montevidéu, Uruguai). (12)

Este trabalho implementa uma arquitetura, em forma de *soft core* totalmente compatível com o padrão *Wishbone*, para a extração de características invariantes em imagens binárias utilizando-se de dispositivos de lógica programável complexa. Desse modo torna-se possível o uso do barramento PCI para a transmissão de dados para um microcomputador ou uma estação de trabalho. Há também um ganho de desempenho no cálculo dos momentos frente ao microcomputador e uma rápida e portável solução de transmissão possível, utilizando-se o barramento PCI.

No capítulo 2 é dada uma visão geral sobre os dispositivos de lógica programável, incluindo suas principais características, as principais áreas de aplicação, a evolução e as perspectivas para o futuro. Tudo isso juntamente com a família escolhida para o trabalho de mestrado e as principais vantagens que a mesma oferece e que serão utilizadas no trabalho.

No capítulo 3 é feita uma revisão sobre reconhecimento de padrões e características invariantes. São apresentadas as etapas do processo de

reconhecimento de padrões, a importância das mesmas e qual delas é proposta para ser implementada pelo trabalho. Em seguida são apresentados e situados as características invariantes em imagens digitais, suas principais características e motivação para implementação dos mesmos.

No capítulo 4 é apresentada a arquitetura de aquisição, armazenamento e exibição de imagens binárias utilizada no projeto, juntamente com a arquitetura para extração de características invariantes em imagens binárias desenvolvida em linguagem VHDL, suas características testes realizados e resultados obtidos.

No capitulo 5 são apresentadas as conclusões tiradas do trabalho desenvolvido e apresentadas sugestões para trabalhos futuros.

2. Lógica Programável de Alta Densidade

2.1 Considerações Iniciais

O objetivo desse capítulo é dar uma noção básica de como são os dispositivos de lógica programável complexa de alta densidade (FPGAs – *Field Programable Gate Arrays*), suas principais características, sua evolução e as principais áreas de aplicação.

Um FPGA é formado basicamente por um arranjo de blocos lógicos, circundados por blocos de entrada e saída (*I/Os*), programáveis, que são conectados via interconexões programáveis (13), como apresentado na figura 2.1.

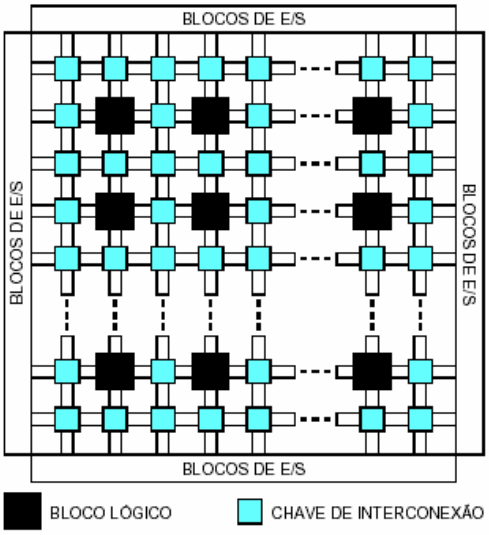


Figura 2.1 Estrutura Básica de um FPGA

Um FPGA é programado com o uso de chaves eletrônicas programáveis. As propriedades destas chaves, tais como, tamanho, resistência de contato e capacitâncias definem os compromissos de desempenho da arquitetura interna do FPGA. Existem também diferentes tecnologias de programação de chaves eletrônicas que são: SRAM (Memória estática de acesso aleatório), anti-fusível, porta flutuante e memória *flash* (Memória permanente programável). No caso dos FPGAs Xilinx (família 4000 e Virtex) (14), Altera (família Flex 10K) (15), Plessey (16), Algotronix (17), Concurrent Logic (18) e Toshiba (19), a tecnologia de programação utiliza memórias de acesso aleatório estáticas – (SRAM - *static random access memories*). Sendo estas memórias voláteis, o FPGA deve receber seu arquivo de configuração toda vez que o circuito for ligado. Isto requer memória externa e permanente para armazenar os *bits* de configuração, podendo ser empregadas Memórias Programáveis de somente leitura – PROM (*Programable Read-Only Memory)*, Memórias PROM apagáveis – EPROM (*Erasable PROM)*, Memórias PROM eletricamente apagáveis

e programáveis – EEPROM (*Electrically EPROM*) ou discos magnéticos. A maior desvantagem das SRAM é a área ocupada. São necessários pelo menos cinco transistores para implementar uma célula SRAM e pelo menos mais um transistor para servir de chave programável.

Apesar da desvantagem de maior área ocupada e necessidade de memória externa para configuração, esta é a tecnologia que domina hoje o mercado de FPGAs, pois permite prototipação de sistemas e reconfiguração de arquiteturas, tanto estática quanto dinamicamente.

A seguir é apresentada a família e o modelo de FPGAs escolhidas para serem usadas no trabalho de mestrado, as famílias Cyclone e Stratix da empresa Altera.

2.2 A Família Cyclone da Altera

2.2.1 Motivação da escolha

O modelo de FPGA proposto inicialmente para uso no projeto, embora não tenha sido utilizado, foi o EP1C6Q240C8 da família Cyclone da Altera. Optou-se por esse modelo de FPGA basicamente por dois fatores principais: configuração de pinos e densidade.

A configuração de pinos foi decisiva pelo fato do sistema ser implementado em hardware utilizando-se o equipamento para produção de circuito impresso e solda de componentes eletrônicos presentes no laboratório. O FPGA usado deveria ter configuração de pinos compatível com a capacidade do equipamento de solda, e o modelo a ser apresentado foi o que melhor se adequou às características necessárias.

Uma densidade alta e um número grande de pinos de entrada e sai são necessários para a implementação do sistema, devido ao esperado tamanho que a arquitetura alcançou, mesmo esse modelo extremamente mais denso que as CPLDs disponíveis no laboratório mostrou-se não suficiente para comportar o sistema completo. Apenas a arquitetura de extração de imagens já preenche completamente o modelo de FPGA pré-existente no laboratório, sendo que no modelo escolhido a arquitetura ocupa em torno de 1% do FPGA, deixando o restante para a implementação da arquitetura desenvolvida nesse trabalho. Como já foi dito anteriormente, esse tamanho acabou mostrando-se insuficiente para comportar o sistema, mas deixou uma nova ferramenta de testes de sistemas em hardware através da placa apresentada no anexo D.

2.2.2 Características Principais

A família Cyclone faz parte de uma linha de FPGAs de baixo custo da Altera. Segundo o website da empresa (ALTERA) os dispositivos da linha Cyclone são os

que possuem uma das melhores relação custo/elementos lógicos dentre todas as linhas de FPGAs. Pode-se notar na figura 2.5 que o modelo a ser usado no projeto possui cerca de seis mil elementos lógicos.

Dispositivo	EP1C3	EP1C4	EP1C6	EP1C12	EP1C20
Elementos lógicos	2,910	4,000	5,980	12,060	20,060
M4K RAM blocks (128 × 36 bits)	13	17	20	52	64
Total RAM bits	59,904	78,336	92,160	239,616	294,912
PLLs	1	2	2	2	2
Pinos de entrada e saída (max.)	104	301	185	249	301

Figura 2.2 - Atributos da Família Cyclone

Uma característica interessante dos FPGAs da família Cyclone é a existência de blocos de memória presentes internamente nas mesmas. A estrutura de um FPGA da família Cyclone com as linhas de memória é apresentada na figura 2.6.

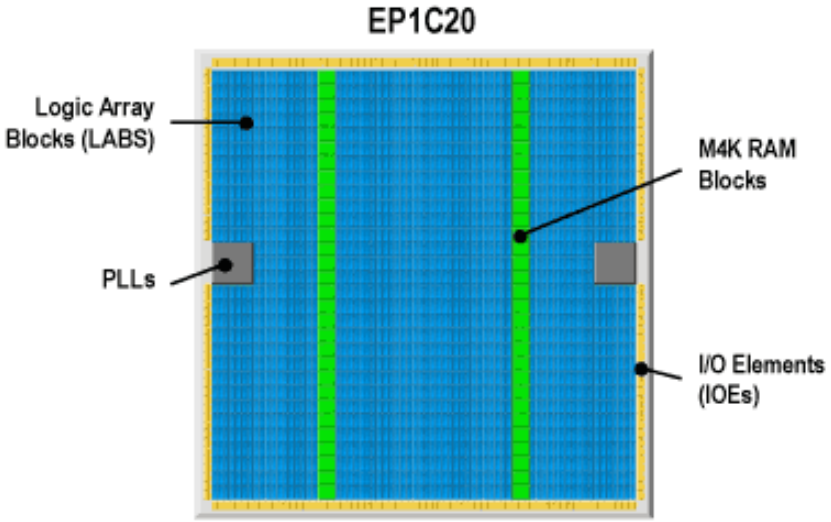


Figura 2.3 – Plano base de um FPGA da família Cyclone

Uma característica interessante dos modelos da família Cyclone é o fato dos mesmos possuírem blocos configuráveis exclusivos para memória, o que é chamado de memória embutida. Esses blocos possuem um alto desempenho e tem a vantagem de não gastarem elementos lógicos que podem ser utilizados para outras funções. Armazenar os resultados dos cálculos nesses blocos de memória, configurando-os de acordo com as necessidades seria interessante para o projeto, podendo ser implementado com FPGAs da família Cyclone II que possuem densidade maior e mesma característica.

2.2.3 Memória Embutida

A memória embutida nos FPGAs da família Cyclone consiste de colunas de blocos de memória do tipo M4K (bloco de memória de 4 Kbits). O modelo EP1C6, que será utilizado no projeto, tem somente uma coluna de blocos M4K, enquanto os modelos EP1C12 e EP1C20 possuem duas colunas. Como pode ser observado na figura 2.5, o modelo usado no projeto possui 90Kbits de memória total. Cada bloco M4K pode implementar vários tipos de memória com ou sem paridade, incluindo duplo barramento composto, duplo barramento simples, e barramento único de RAM, ROM, e *buffers* FIFO (áreas temporárias de memória do tipo “primeiro que entra é o primeiro que sai”; “*First in, First out*”). Os blocos M4K suportam as seguintes especificações:

- 4,608 *bits* de RAM
- Velocidade de até 200Mhz
- Memória de duplo barramento composto
- Memória de duplo barramento simples
- Memória de barramento único
- *Byte enable*
- *Bits* de paridade
- Registradores de deslocamento
- *Buffers* do tipo FIFO
- ROM
- Modos de *clock* misto

2.2.3.1 Modos de memória

Os blocos de memória M4K incluem registradores de entrada que sincronizam os registros de leitura e escrita para sistemas *pipeline* e aumentam o desempenho do sistema. Os blocos M4K oferecem o modo de duplo barramento composto para suportar qualquer combinação de operações em duplo barramento: duas escritas, duas leituras, ou uma leitura e uma escrita com duas freqüências de *clock* (relógio) diferentes. A figura 2.6 mostra uma memória de duplo barramento composto.

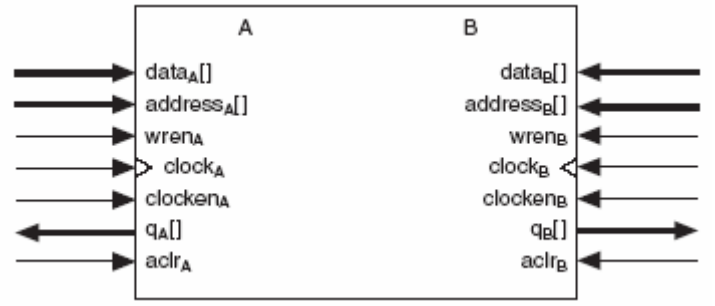


Figura 2.4 – Memória de barramento duplo composto

Além de duplo barramento composto, os blocos de memória M4K suportam RAM de duplo barramento simples e barramento único. O modo de duplo barramento simples suporta escrita e leitura simultâneas. O modo de barramento simples suporta escrita e leitura não simultaneamente. A figura 2.7 mostra essas diferentes configurações de barramento de memória RAM que os blocos M4K suportam.

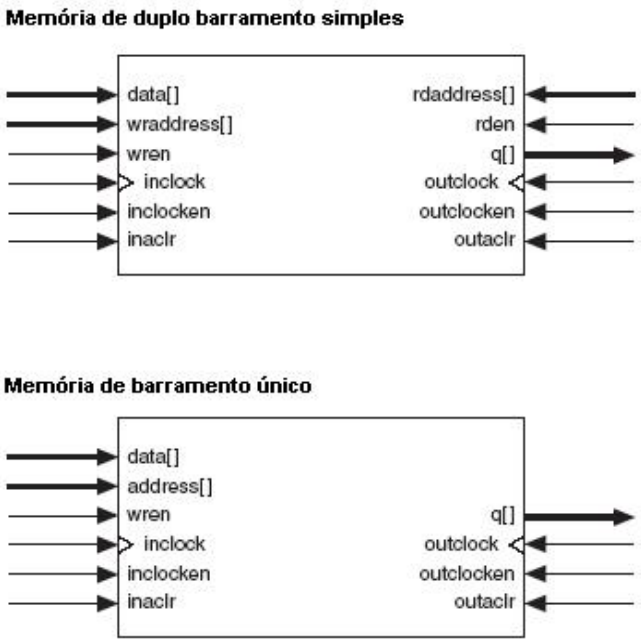


Figura 2.5 – Memória de duplo barramento simples e de barramento único

Os blocos de memória também permitem leitura e escrita com tamanho variável de dados nas portas de RAM na configuração de RAM de duplo barramento. Por exemplo, o bloco de memória pode ser escrito no modo 1x na porta A e lido no modo 16x na porta B.

A arquitetura de memória da família Cyclone permite que se implementem RAMs totalmente síncronas através do uso de registradores nos sinais de entrada e saída do bloco M4K. Todos as entradas do bloco são registradas, provendo assim ciclos de escrita síncronos.

Quando configurados como RAM ou ROM, o projetista do sistema pode usar um arquivo de pré-configuração para definir o conteúdo inicial das memórias.

Dois blocos de memória de barramento único podem ser implementados em um mesmo bloco M4K, desde que ambos tenham o tamanho menor ou igual a até metade do tamanho total do bloco M4K.

O software Quartus II da Altera, que é uma ferramenta CAD utilizada para o desenvolvimento de arquiteturas e programação de dispositivos de lógica programável complexa, implementa automaticamente blocos grandes de memória através da combinação de múltiplos blocos M4K. Por exemplo, dois blocos de RAM de 256x16*bits* podem ser combinados para formar um único bloco de 256x32-*bits* de RAM. O desempenho da memória não é comprometido em blocos que usam o maior tamanho possível de palavra de dados. Blocos de memória lógicos que utilizam

menos do que o maior tamanho permitido são nada além do que blocos físicos em paralelo, o que elimina qualquer lógica de controle externo que poderia acarretar em atraso. Para criar blocos de memória de alta velocidade mais largos, o software Quartus II combina automaticamente blocos de memória com controle lógico feito por elementos lógicos do próprio FPGA.

2.2.3.2 Habilitação de bytes (*Byte Enable)*

Os blocos M4K suportam escrita de dados quando a porta de entrada tem os tamanhos de 16, 18, 32 ou 36 *bits*. O *byte enable* permite que se mascarem os dados de entrada permitindo a escrita em *bytes* específicos. Os *bytes* não escritos retêm os valores que foram previamente escritos. A tabela na figura 2.8 sumariza a escolha de *bytes*.

byteena[3..0]	datain × 18	datain × 36
[0] = 1	[8..0]	[8..0]
[1] = 1	[17..9]	[17..9]
[2] = 1	–	[26..18]
[3] = 1	–	[35..27]

Figura 2.6 – Escolha de bytes por *byte enable*

2.2.3.3 Suporte a *Bits* de paridade

Os blocos M4k suportam um bit de paridade para cada *byte*. O *bit* de paridade, juntamente com uma lógica interna em elemento lógico, pode implementar um teste de paridade para detecção de erros a fim de garantir a integridade dos dados. Podem-se também usar palavras de paridade, a fim de se guardar os *bits* de controle

especificados pelo usuário. Sinalizadores de *byte* (*Byte enables)* também são disponíveis para mascarar dados durante operações de escrita.

2.2.3.4 Suporte a Registradores de Deslocamento (*Shift Register*)

Os blocos de memória M4K podem ser configurados como registradores de deslocamento. O tamanho *W x M x N* de um registrador de deslocamento é determinado pela largura dos dados de entrada (W), o comprimento das ligações (M), e o número de ligações (N). O tamanho de um registrador de deslocamento *W x M x N* deve ser igual ou menor ao tamanho máximo de *bits* de memória do bloco M4K (4608 *bits*). O número total de saídas do registro de deslocamento (número de ligações N x extensão W) deve ser menor que a extensão máxima do bloco de memória RAM M4K (x36). Para criar registradores mais largos, múltiplos blocos são ligados em modo de cascata.

Os dados são escrito em cada endereço na borda de descida do sinal de *clock* e lidos do endereço na borda de subida do sinal de *clock*. A lógica do modo de registrador de deslocamento controla automaticamente as bordas positivas e negativas do sinal de *clock*, a fim de deslocar os dados em um ciclo de *clock*. A figura 2.9 mostra o bloco M4K no modo registrador de deslocamento.

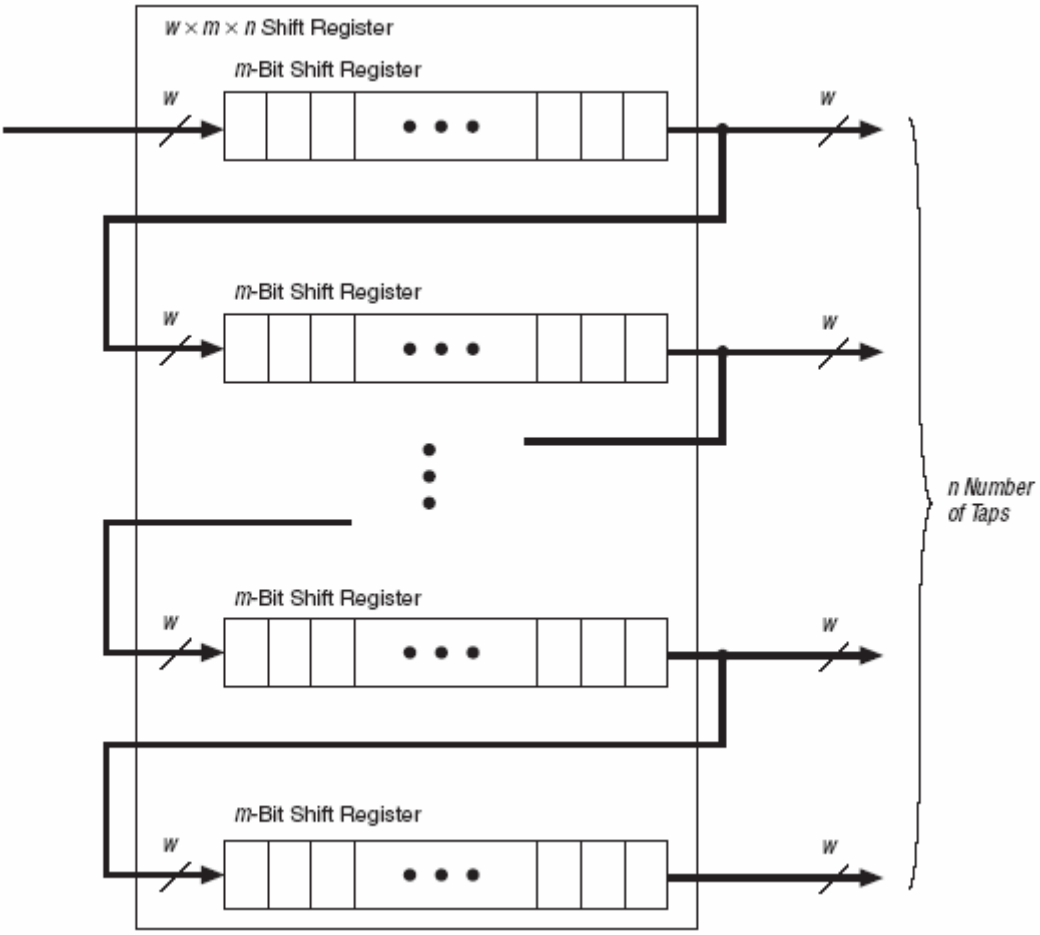


Figura 2.7 – Modo de configuração de Registrador de Deslocamento

2.2.3.5 Tamanhos de configuração

A profundidade de endereços e a largura das saídas de dados podem ser configuradas como 4096 x 1, 2048 x 2, 1024 x 4, 512 x 8 (ou 512 x 9 *bits*), 256 x 16 (ou 256 x 18 *bits*), e 128 x 32 (ou 128 x 36 *bits*). As configurações 128 x 32 ou 36 *bits* não são disponíveis no modo duplo barramento composto. Larguras de saída variáveis são possíveis, permitindo profundidades diferentes de leitura e escrita. As tabelas presentes nas figuras 2.10 e 2.11 ilustram as possíveis configurações do bloco M4K como RAM.

Read Port	Write Port								
	4K × 1	2K × 2	1K × 4	512 × 8	256 × 16	128 × 32	512 × 9	256 × 18	128 × 36
4K × 1	✓	✓	✓	✓	✓	✓			
2K × 2	✓	✓	✓	✓	✓	✓			
1K × 4	✓	✓	✓	✓	✓	✓			
512 × 8	✓	✓	✓	✓	✓	✓			
256 × 16	✓	✓	✓	✓	✓	✓			
128 × 32	✓	✓	✓	✓	✓	✓			
512 × 9							✓	✓	✓
256 × 18							✓	✓	✓
128 × 36							✓	✓	✓

Figura 2.8 – Configurações possíveis em duplo barramento simples

Port A	Port B						
	4K × 1	2K × 2	1K × 4	512 × 8	256 × 16	512 × 9	256 × 18
4K × 1	✓	✓	✓	✓	✓		
2K × 2	✓	✓	✓	✓	✓		
1K × 4	✓	✓	✓	✓	✓		
512 × 8	✓	✓	✓	✓	✓		
256 × 16	✓	✓	✓	✓	✓		
512 × 9						✓	✓
256 × 18						✓	✓

Figura 2.9 – Configurações possíveis em duplo barramento composto

2.2.3.6 Modos de *clock*

- Modo de *clock* independente: Os blocos M4K implementam modos de *clock* independentes para memórias de duplo barramento composto. Nesse modo, um *clock* separado é disponível para cada uma das portas (porta A e porta B). O *clock* A controla todos os registros na porta A, enquanto *clock* B controla

todos os registros no lado da porta B. Cada porta, A e B, também suporta sinais de *clock enable* e *asynchronous clear* independentes para seus registradores. A figura 2.12 mostra um bloco M4K em modo de *clock* independente.

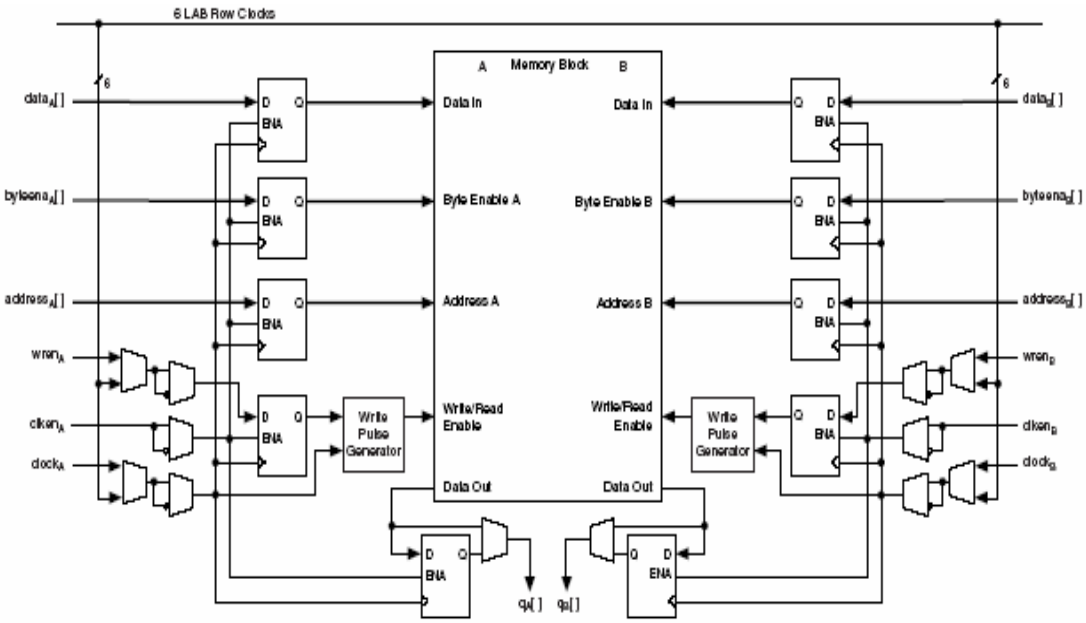


Figura 2.10 – Modo de *clock* independente

- Modo de entrada e saída: O modo de entrada e saída pode ser implementado por ambos os modos de barramento, duplo simples e composto. Em cada uma das duas portas, A ou B, um sinal de *clock* controla todos os registradores de entrada no bloco de memória: entrada de dados, *wren* e endereços. O outro *clock* controla os registradores de saída do bloco. Cada porta do bloco de memória, A ou B, também suporta *clock enables* e sinais *clear* assíncronos para os registradores de entrada e saída independentes. As figuras 2.13 e 2.14 ilustram os blocos de memória no modo de *clock* de entrada e saída.

Clock em modo Entrada e Saída em duplo barramento composto

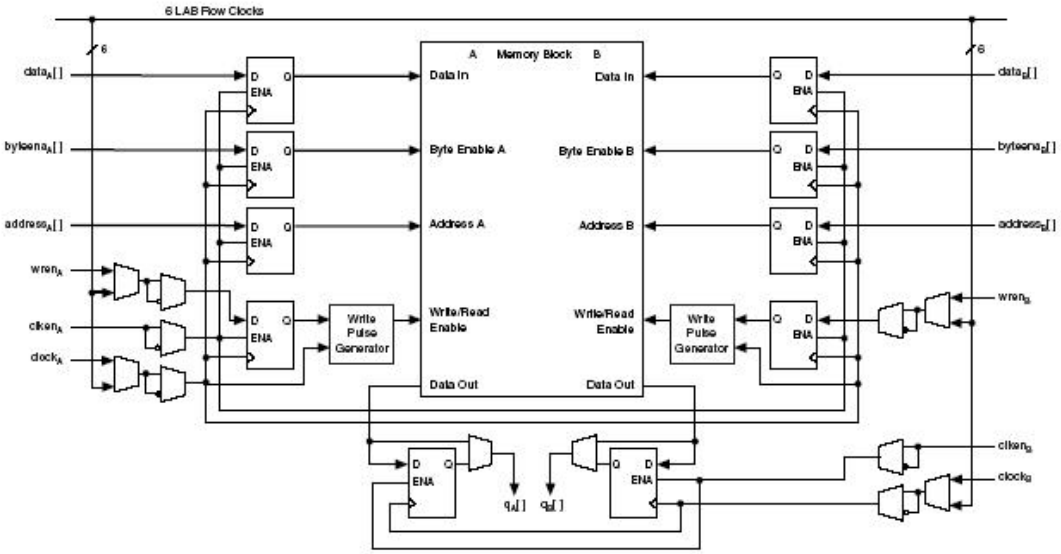


Figura 2.11 – *Clock* em modo Entrada e Saída em duplo barramento composto

Clock em modo Entrada e Saída em duplo barramento simples

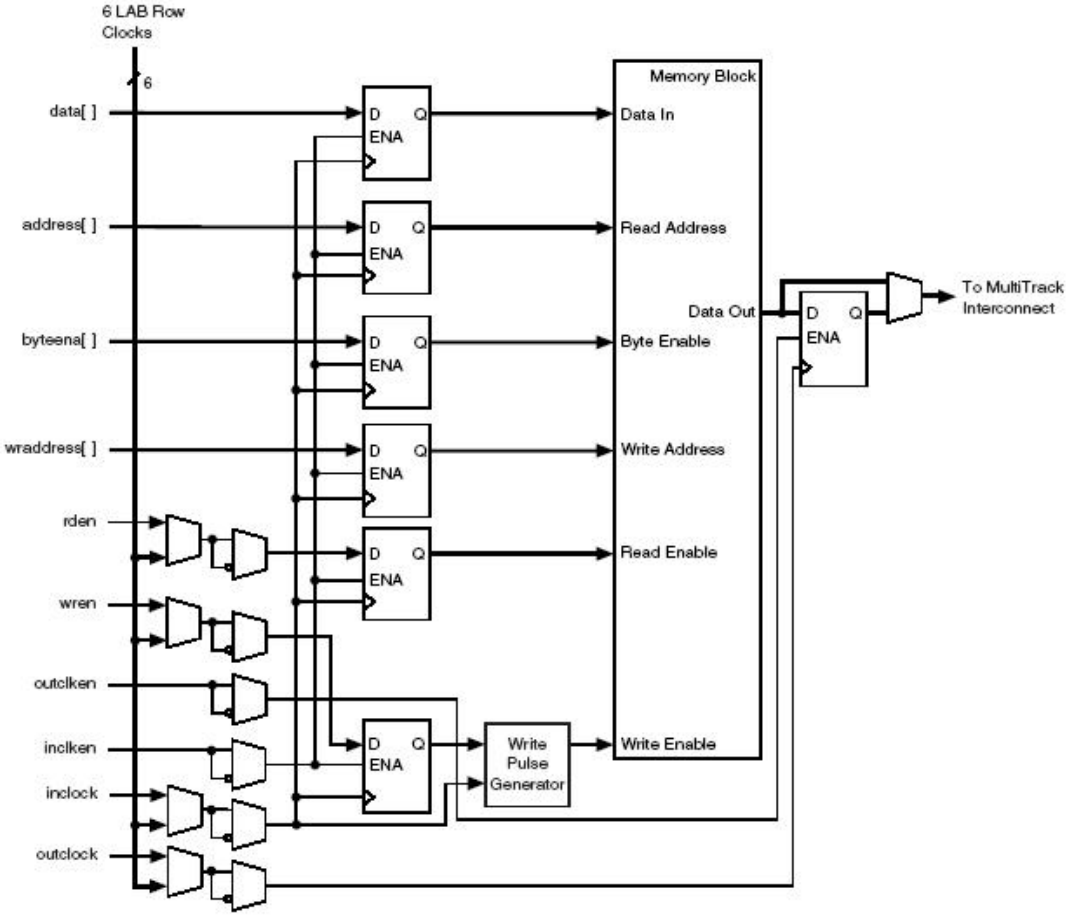


Figura 2.12 – *Clock* em modo Entrada e Saída em duplo barramento simples

- Modo de Escrita e Leitura: Os blocos de memória M4K implementam o modo de *clock* Escrita/Leitura para duplo barramento simples. Podem ser usados até dois sinais de *clock* nesse modo. O *clock* de escrita controla os sinais de entrada do bloco, *wraddress*, e *wren*. O *clock* de leitura controla os sinais de saída de dados, *rdaddress*, e *rden*. O bloco de memória suporta sinais de *clock enable* independentes para cada *clock* e sinais de *clear* assíncronos para os registros leitura e escrita. A figura 2.15 mostra um bloco de memória no modo Escrita/Leitura.

2.3 A Família Stratix da Altera

2.3.1 Descrição

Um modelo EP1S10F780C6 da família Stratix foi utilizado para a realização de testes práticos do sistema. Esse modelo possui características semelhantes a da família cyclone, porém com as seguintes capacidades:

- Elementos lógicos: 10.000
- Blocos de memória M4K: 60
- Total de bits de RAM: 920.448
- Pinos de Entrada/Saída: 426
- PLLs : 6

2.4 Computação Configurável

A computação com FPGAs é denominada configurável devido ao fato de ser definida pela configuração de *bits* no FPGA, a qual define a função dos blocos lógicos e a interconexão entre estes. Como nos processadores, FPGAs são programados após a fabricação para solucionar virtualmente qualquer tarefa computacional, isto é, qualquer tarefa que caiba nos recursos finitos do dispositivo. Esta padronização pós-fabricação distingue processadores e FPGAs de blocos funcionais padronizados, os quais têm suas funções definidas durante a fabricação e implementam somente uma função ou um pequeno número de funções.

Diferentemente dos processadores, os FPGAs implementam o equivalente a uma única, e poderosa, instrução (22). Nos dispositivos configuráveis as operações são implementadas de forma espacial, explorando-se o paralelismo inerente das aplicações alvo. Já nos processadores, as instruções são organizadas seqüencialmente para executarem a mesma função. A Figura 2.16 (23) compara a implementação de um filtro digital, utilizando-se dispositivos programáveis e microprocessadores.

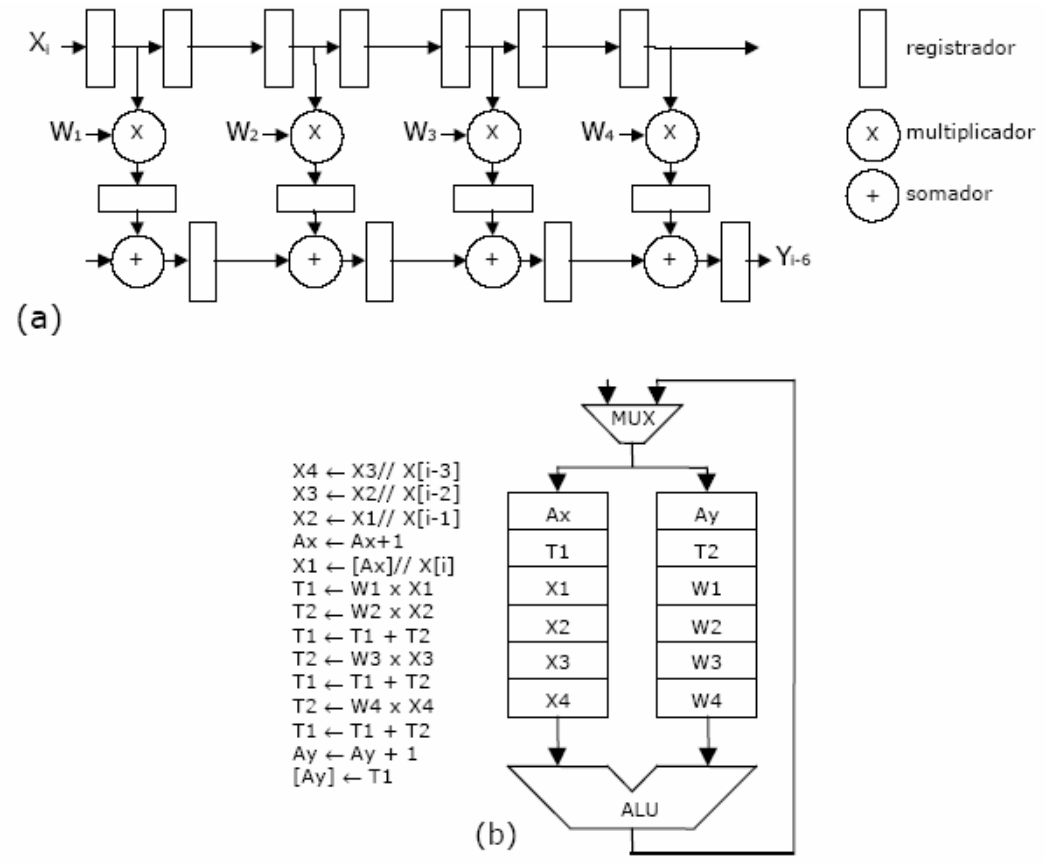


FIGURA 2.13 – Comparação de implementação entre FPGA e microprocessador padrão x86. (a) computação paralela – FPGA; (b) computação temporal – microprocessador. Ambas implementações computam a expressão y[i]=w1.x[i] + w2.x[i-1] + w3.x[i-2] + w4.x[i-3], de um filtro com resposta a um impulso finito (FIR).

Para a implementação FPGA, observa-se que operadores aritméticos e registradores são replicados, resultando em solução com uma alta taxa de saída (um *bit* processado por ciclo de *clock*). Já na implementação com microprocessador, temos um conjunto seqüencial de operações, o que resulta em uma taxa de processamento inferior aos dispositivos programáveis.

O advento de FPGAs complexos (por exemplo, com capacidade para implementar circuitos com dois milhões de portas lógicas equivalentes em famílias

VIRTEX-XILINX e APEX-ALTERA), possibilitou retomar o trabalho proposto no início da década de 60 por Geral Estrin (22) (24). Estrin propôs um “computador com uma estrutura fixa e uma estrutura variável”, no qual o *hardware* era dedicado tanto a abstração de um processador programável (inflexível) quanto a um componente que implementava lógica digital (flexível) . Esta arquitetura básica, que dá suporte a *hardware* programável e *software*, é o núcleo de muitos sistemas computacionais configuráveis subseqüentes (25). Muitos dos conceitos aplicados atualmente em computação reconfigurável têm como base o trabalho de Estrin. Ao final dos anos 80 e início dos anos 90, várias arquiteturas reconfiguráveis (26) (27) foram propostas e desenvolvidas. Atualmente elas continuam a serem desenvolvidas com resultados cada vez mais animadores frente às tecnologias já existentes, mostrando o poder e a evolução dos FPGAs (28) (29) (30).

Os FPGAs têm sido utilizados em um crescente número de sistemas digitais. Equipamentos computacionais baseados em FPGAs, também denominados Máquinas Computacionais Especializadas – CCMs, são eficientes e apresentam alto desempenho na solução de problemas computacionais complexos. Em muitos casos, um simples arranjo de FPGAs supera o desempenho de uma estação de trabalho ou até mesmo um supercomputador (31).

Um bom desempenho e eficiência são alcançados por arquiteturas computacionais especializadas desenvolvidas para solucionar problemas específicos. As técnicas de especialização de arquiteturas computacionais incluem a otimização dos elementos processadores e armazenamento, conduzindo à solução adequada para

um domínio limitado de problemas. Limitando-se uma arquitetura a uma certa aplicação específica permite-se aos recursos de *hardware* serem utilizados mais eficientemente se comparado a arquiteturas de propósito geral.

Muitos exemplos de CCMs apresentam incremento significativo de desempenho devido à personalização da arquitetura a uma área específica de aplicação. Pode-se citar como exemplo, a pesquisa em um banco de dados de genética realizada pela CCM SPLASH-2. Para esta aplicação (na data de publicação da referência), o SPLASH-2 superava, em velocidade, um supercomputador em duas ordens de grandeza (32). O SPLASH-2 alcança este alto nível de desempenho replicando o processamento de casamento (*matching*) de caracteres através de sua arquitetura reconfigurável. Como outro exemplo, pode-se citar a implementação do circuito de criptografia RSA na CCM DECPeRLe-1 que apresenta, na decodificação criptográfica, desempenho superior ao estado-da-arte na data de sua publicação (35). Este sistema alcança altas taxas de decodificação utilizando multiplicadores de inteiros longos dedicados e módulos de exponenciação, ambos implementados em FPGAs.

Em sistemas computacionais tradicionais a especialização arquitetural é alcançada ao custo da flexibilidade. Arquiteturas computacionais projetadas para uma área de aplicação são geralmente muito ineficientes ou impróprias para outras áreas de aplicação. A necessidade de flexibilidade nos sistemas de propósito específico inibe seu emprego mais amplo. Em sistemas baseados em FPGAs, a especialização da arquitetura não sacrifica sua flexibilidade. Devido à possibilidade

de reconfiguração dos circuitos, um FPGA pode operar como uma variedade de arquiteturas computacionais específicas. Esta flexibilidade torna um sistema CCM uma alternativa atraente para muitas aplicações específicas. A ampla variedade de problemas computacionais solucionados pelo DECPeRLe-1 demonstra esta flexibilidade. Exemplos de aplicação que demonstram altos níveis de desempenho alcançados pelo DECPeRLe-1 incluem (35): multiplicação de números longos, criptografia RSA, compressão de dados, casamento de cadeias de caracteres, equações de Laplace, mecânica de Newton, convolução em duas dimensões, máquina de Boltzman, geometria em três dimensões e a transformada discreta do coseno (DCT). Com uma arquitetura fixa não há como atingir um bom desempenho em tal variedade de problemas computacionais. Um ASIC projetado para realizar todas as tarefas do DECPeRLe-1, por exemplo, apresentaria níveis de desempenho muito mais baixos, pois teria que ser implementado praticamente como um processador genérico.

2.4.1 Módulos de hardware pré-projetados

O mercado atual de sistemas digitais é caracterizado pela necessidade de um reduzido tempo para colocação de novos produtos em comercialização (*time-to-market)*, conter um elevado número de portas lógicas, além de terem que apresentar alto desempenho e baixo consumo de potência. Apesar da necessidade de um reduzido *time-to-market*, a qualidade dos produtos não pode ser comprometida, correndo-se o risco de se chegar ao mercado com um produto não competitivo. Este

ambiente competitivo induz mudanças fundamentais nos métodos de projeto de sistemas digitais VLSI. A utilização de módulos pré-projetados e pré-validados de *hardware*, denominados *cores* permite o desenvolvimento de circuitos muito mais complexos em um tempo de projeto muito mais reduzido. As empresas especializam-se em uma determinada área, por exemplo telecomunicações, e os módulos em que a empresa não é especialista são comprados de terceiros, por exemplo, núcleos de processadores. Estes *cores* são protegidos por leis de Propriedade Intelectual (*Intelectual Proprierty* – IP) (34) (35).

Outra tendência observada no projeto de circuitos VLSI é a integração completa de sistemas digitais em um único *chip* (*Sistem-on-Chip* – SoC) (35). Estes sistemas integrados em um único *chip* são implementados utilizando-se *cores*, apresentando a vantagem da redução do tempo entre a comunicação *hardware/software*, uma vez que o núcleo processador e o *hardware* estão integrados no mesmo dispositivo.

Uma integração desse tipo pode ser feita em trabalhos futuros com a utilização do *core* para extração de características invariantes juntamente com uma rede neural para fazer todo o processo de reconhecimento de padrões no próprio hardware, criando assim um SoC para reconhecimento de padrões.

2.5 Considerações Finais

Segundo CHOW (35), conforme a tecnologia se escala, a área da lógica decresce, entretanto, o efeito relativo dos atrasos devido às conexões aumenta. Portanto, faz-se necessário o surgimento de novas estruturas de roteamento. Também, através do desenvolvimento de arquiteturas com blocos lógicos não homogêneos é possível obter um melhor compromisso entre área e desempenho. É importante também ao projetista conhecer a arquitetura do dispositivo a ser utilizado para que se possa otimizar o desempenho do sistema. Além disso, uma boa descrição de projeto pode otimizar a sintetização lógica.

Grande parte desses avanços já é visível. Os circuitos de lógica programável vêm crescendo em densidade e velocidade de *clock*. O reconhecimento do poder desse tipo de ferramenta se reflete no constante aumento do volume de pesquisas na área, despertando o interesse de cada vez mais setores acadêmicos e industriais e fazendo com que os fabricantes se empenhem em oferecer cada vez mais inovações. Com o advento dos SoC, observa-se uma tendência de integração cada vez maior entre os desenvolvedores de *soft cores* (36). Isso foi e é de fundamental importância para a disseminação da tecnologia de FPGAs, e é exatamente o que é feito nesse trabalho de mestrado: um *soft core* com interface para integração dele com outros de forma funcional.

Apesar de ainda não terem chegado ao nível de velocidade de *clock* obtidos pelos grandes processadores do mercado, as FPGAs atuais já conseguem demonstrar superioridade frente aos mesmos e uma gama de aplicações, inclusive no processamento digital de imagens(28) (15) (37), fazendo com que a expectativa de ganho de desempenho pelo sistema desenvolvido seja algo não ilusório.

Isso tudo vem decretar não a futura falência dos processadores como conhecemos, mas que a integração entre as duas tecnologias está próxima e estará fazendo parte de nosso cotidiano mais breve do que imaginamos.

3. Reconhecimento de padrões e características invariantes.

3.1 Reconhecimento de padrões

O reconhecimento de padrões é o estudo sobre as formas como as quais as máquinas observam seu entorno, aprendem a distinguir padrões de interesse e tomam decisões razoáveis sobre as categorias dos padrões (38). Um *padrão* é uma descrição de um objeto, que pode ser classificado como concreto (espaciais: caracteres, imagens; e temporais: formas de onda, séries) ou abstrato (raciocínio, soluções de problemas, etc) (39). Um computador consegue reconhecer padrões, convertendo-os em sinais digitais e comparando-os com outros sinais já armazenados na memória.

A tarefa de Reconhecimento pode ser dividida, conforme a figura, em quatro etapas:

- Aquisição de dados;
- Pré-Processamento;
- Extração de Atributos;
- Classificação.

Na etapa de aquisição de dados, os objetos a serem classificados devem ser captados por sensores de imagens, os quais irão produzir modelos de imagens com algumas simplificações em relação às imagens reais: são discretos, bidimensionais, limitados em extensão e em número de cores ou níveis de cinza possíveis. Estes modelos, que são uma representação numérica da imagem, tornam possíveis o armazenamento e processamento das imagens através de computadores digitais. Tamanhos típicos de imagens são 128x128, 512x512, e para imagens monocromáticas 256 é um número freqüentemente utilizado de tonalidades de cinza, o qual normalmente é utilizado em função de requisitos de armazenamento.

A etapa de pré-processamento tem como objetivo filtrar ou minimizar os ruídos e distorções que possam resultar de processo de aquisição dos dados de entrada. Também podem ser realizadas, em alguns casos, operações de segmentação e normalização dos dados de entrada. A segmentação faz a separação de padrões que estejam de alguma forma ligados a outros, enquanto que a normalização modifica os dados de entrada reduzindo-os à escala normal das classes dos padrões, limitando o espaço de entrada.

A etapa de extração de atributos ou características consiste na obtenção de medidas relevantes (atributos) que possam ser usadas na caracterização de padrões. Tais características podem ser numéricas (área, volume), simbólicas (cor), ou uma combinação das duas. A extração de atributos é também uma forma de compressão de dados, já que reduz a dimensionalidade da informação, elimina a redundância e aumenta a discriminação entre classes.

A escolha de um conjunto de atributos deve levar em conta algumas propriedades dos mesmos (40):

- Velocidade de Processamento;
- Grande Discriminação de Classes;
- Pouca variância em cada Classe;
- Completude da Descrição.

A utilização de atributos invariantes a transformações é um método bastante utilizado no reconhecimento de padrões, sendo que a escolha destes na maioria das vezes influencia as taxas de reconhecimento alcançadas. Fatores como, tempo de cálculo dos atributos (na fase de pré-processamento) e redução de dados (compressão) devem também, ser levados em conta (41).

A etapa de classificação realiza um mapeamento entre os atributos e as classes dos padrões, ou seja, através dos atributos calculados, o classificador define uma determinada classe baseado em uma função de decisão.

Os *n* atributos podem ser visualizados como coordenadas de um espaço n-dimensional, e os objetos a serem identificados podem ser representados por pontos neste espaço. As diversas classes de objetos particionam o espaço de atributos, formando subconjuntos (durante a fase de treinamento do sistema). Tais subconjuntos são regiões de decisão. Assim, um objeto será atribuído a uma determinada classe se seus pontos de atributos representativos estiverem em apenas uma região de decisão. Caso contrário, o objeto deve ser rejeitado (não classificado).

Visto isso, através da escolha do uso dos atributos invariantes de Hu (42), espera-se que a arquitetura proposta apresente um ganho de desempenho frente às arquiteturas tradicionais de microprocessadores. Com isso o resultado será uma aceleração na fase de pré-processamento em um possível sistema de reconhecimento de padrões, dando assim aplicabilidade ao trabalho.

3.2 Atributos invariantes

3.2.1 Características

Atributos são propriedades mensuráveis dos objetos presentes na imagem e a representação de padrões através de atributos invariantes a transformações geométricas é um método de implementação do reconhecimento de padrões.

Os atributos podem ser globais (perímetro, área, momentos), locais (segmentos de reta, vértices) ou relacionais (regiões do objeto, distâncias) e sua escolha, na maioria das vezes, determina o desempenho do sistema de reconhecimento de padrões usando espaço de atributos invariantes (43).

Normalmente, a escolha dos tipos de atributos utilizados pelo sistema depende dos tipos de objetos ou padrões a serem reconhecidos. Idealmente um atributo invariante teria valor constante, independente do grau de transformação sofrido pela imagem do objeto. Porém, na prática, diversas fontes de incertezas são acrescentadas ao modelo ideal, resultando, em uma variabilidade de valores medidos.

Momentos descrevem quantidades numéricas em alguma distância de um ponto de referência ou eixo e podem ser utilizados para caracterizar uma imagem, binária ou em níveis de cinza, considerando esta como uma função de distribuição bidimensional de densidade (44) (10).

A representação de toda informação presente em uma imagem utilizaria um número infinito de valores de momento. Assim, para a implementação prática de sistemas de reconhecimento que utilizem momentos, torna-se necessário determinar

a ordem para a qual os momentos possam trazer informações que caracterizem a imagem.

A definição de momentos geométricos regulares tem a forma de uma projeção da função f(x,y) que representa a imagem, em uma função polinomial do tipo $x^p y^q$. O momento (p+q) é definido de acordo com a equação 3.1, a seguir.

$$m_{p,q} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} x^p y^q . f(x,y).dxdy \qquad 3.1$$

Para (p,q = 0,1,2,....)

Onde $m_{0,0}$ é a área da região e $m_{0,1}$ e $m_{1,0}$ são as coordenadas do centro de massa da região.

Momentos centrais são momentos centralizados em regiões, e podem ser expressos como na equação 3.2, a seguir.

$$\mu_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x-\bar{x})^p (y-\bar{y})^q f(x,y)dxdy \qquad 3.2$$

onde $\bar{x} = \frac{m_{10}}{m_{00}}$ e $\bar{y} = \frac{m_{01}}{m_{00}}$, que são as coordenadas do centro de massa normalizadas pela área.

Para uma imagem digital,

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \qquad 3.3$$

Para valores de p,q entre 0 e 3, são obtidos à partir da equação 3.3 os momentos centrais de 1ª até 3ª ordem, invariantes a translação e escalonamento.

1ª ordem

$$\mu_{10} = \mu_{01} = 0$$

2ª ordem

$$\mu_{20} = m_{20} - \bar{x} m_{10}$$
$$\mu_{02} = m_{02} - \bar{y} m_{01}$$
$$\mu_{11} = m_{11} - \bar{y} m_{10}$$

3ª ordem

$$\mu_{12} = m_{12} - 2\bar{y}m_{11} - \bar{x}m_{02} + 2\bar{y}^2 m_{10}$$
$$\mu_{21} = m_{21} - 2\bar{x}m_{11} - \bar{y}m_{20} + 2\bar{x}^2 m_{01}$$
$$\mu_{30} = m_{30} - 3\bar{x}m_{20} + 2\bar{x}^2 m_{10}$$
$$\mu_{03} = m_{03} - 3\bar{y}m_{02} + 2\bar{y}^2 m_{01}$$

A combinação de momentos de segunda e terceira ordem resulta em atributos que são invariantes também à rotação dos padrões da imagem.

A partir dos momentos centrais normalizados pela áera:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}} \quad \text{onde: } \gamma = \frac{p+q}{2} + 1 \quad \text{Para (p+q = 2,3,4,...)}$$

(HU (1961)) calculou um conjunto de sete momentos que são relativamente invariantes à translação, rotação e escala.

$$\phi_1 = \eta_{20} + \eta_{02}$$
$$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$
$$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$
$$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$
$$\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})$$
$$(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$
$$\phi_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$
$$\phi_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{12} - \eta_{30})$$
$$(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

Vários tipos de momentos têm sido utilizados para reconhecer padrões invariantes de imagens bi dimensionais nas mais diversas formas de aplicação (45) (46). Momentos ortogonais, momentos de Legendre, momentos de Zernike, momentos rotacionais, momentos complexos, momentos regulares ou geométricos são alguns dos mais freqüentes. Cada um destes tem características e habilidades próprias no que se refere à capacidade de representação de imagens, sensibilidade ao ruído e redundância de informações que carrega (45).

Os métodos para extração de características invariantes existentes possuem certas limitações, sendo a principal delas o alto custo computacional. Com isso surge a necessidade de desenvolvimento de um sistema de extração de características invariantes.

O principal motivo de ter-se escolhido utilizar momentos invariantes no processo de extração de características é o fato desta técnica ser capaz de representar propriedades, de imagens, que são invariantes em relação à rotação, translação e escala. Isto significa que uma imagem, após sofrer alguma das operações citadas (translação, rotação e escala), será representada, por momentos invariantes, da mesma forma que a imagem original. A lógica a ser implementado tem por objetivo principal a extração dos Momentos invariantes de Hu, juntamente com uma interface compatível com o padrão *Wishbone* (que será detalhado nos capítulos seguintes) para

comunicação com um *CORE PCI* que também possui uma interface padronizada *Wishbone*. Conforme descritos anteriormente.

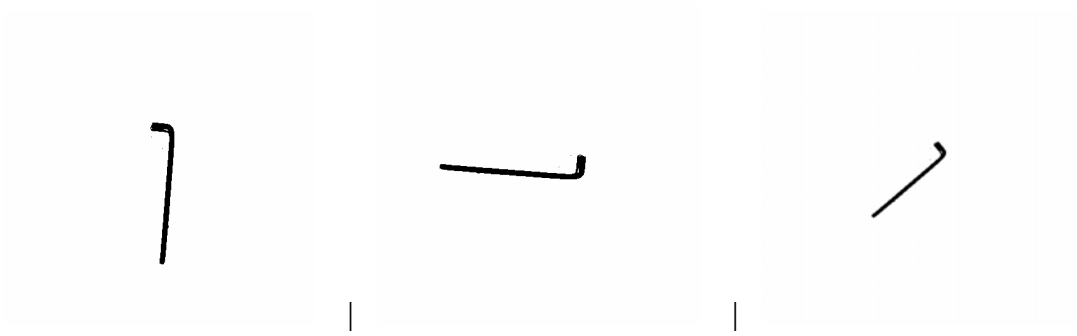
3.2.2 Exemplos

Para exemplificar o cálculo e o uso dos momentos invariantes, que serão implementados pela arquitetura proposta nesse trabalho, apresentamos à seguir um exemplo de cálculo e os valores obtidos para os momentos em diferentes imagens. Isso para demonstrar a usabilidade dos momentos invariantes.

Temos como exemplo de cálculo de momentos em imagens binárias o cálculo dos mesmos, feito através de um programa em *Matlab*. Foram calculados os sete momentos invariantes de Hu.

Os dados obtidos mostram que com apenas quatro casas após a virgula de precisão já é possível obter valores invariantes. São apresentados primeiramente os valores absolutos obtidos, em seguida os valores normalizados da forma como são utilizados em sistemas de reconhecimento de padrões. O código MATLAB dessa função encontra-se no ANEXO C.

Chave Alen:

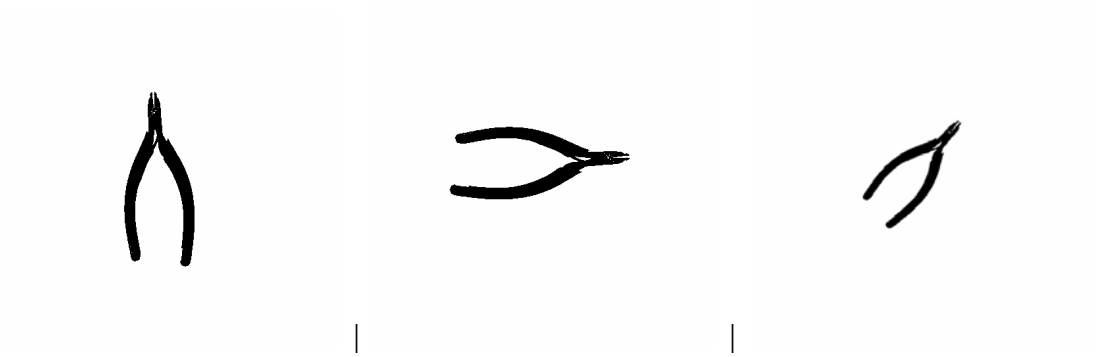


1º momento: 0.00066469	1º momento: 0.00066469	1º momento: 0.00065929
2º momento: 6.0435e-013	2º momento: 6.0435e-013	2º momento: 3.6235e-014
3º momento: 1.0342e-015	3º momento: 5.8156e-015	3º momento: 2.8511e-015
4º momento: 4.5354e-015	4º momento: 2.9416e-015	4º momento: 3.4390e-015
5º momento: -9.5655e-030	5º momento: 1.2031e-029	5º momento: -1.0768e-029
6º momento: 5.0555e-008	6º momento: 4.4493e-008	6º momento: 4.3255e-008
7º momento: -2.2097e-030	7º momento: 1.7962e-030	7º momento: -1.4575e-031

Valores normais (ou normalizados):

1º momento: 0.0007	1º momento: 0.0007	1º momento: 0.0007
2º momento: 0.0000	2º momento: 0.0000	2º momento: 0.0000
3º momento: 34.3106	3º momento: 32.7408	3º momento: 33.4161
4º momento: 32.9791	4º momento: 33.3870	4º momento: 33.2410
5º momento: 36.0437	5º momento: 36.0437	5º momento: 36.0437
6º momento: 16.8002	6º momento: 16.9279	6º momento: 16.9562
7º momento: 36.0437	7º momento: 36.0437	7º momento: 36.0437

Alicate:



1º momento: 0.0141	1º momento: 0.0141	1º momento: 0.0141
2º momento: 1.9943e-004	2º momento: 1.9854e-004	2º momento: 1.9913e-004
3º momento: 1.1835e-007	3º momento: 1.1762e-007	3º momento: 1.1811e-007
4º momento: 1.1835e-007	4º momento: 1.1762e-007	4º momento: 1.1811e-007
5º momento: 1.4008e-014	5º momento: 1.3834e-014	5º momento: 1.3949e-014
6º momento: 1.6714e-009	6º momento: 1.6573e-009	6º momento: 1.6667e-009
7º momento: -1.4008e-014	7º momento: -1.3834e-014	7º momento: -1.3949e-014

Valores normais:

1º momento: 4.2600	1º momento: 4.2623	1º momento: 4.2608
2º momento: 8.5201	2º momento: 8.5245	2º momento: 8.5216
3º momento: 15.9496	3º momento: 15.9558	3º momento: 15.9517
4º momento: 15.9496	4º momento: 15.9558	4º momento: 15.9517
5º momento: 31.8834	5º momento: 31.8957	5º momento: 31.8875
6º momento: 20.2096	6º momento: 20.2181	6º momento: 20.2124
7º momento: 32.0694	7º momento: 32.0820	7º momento: 32.0736

Inglesa:

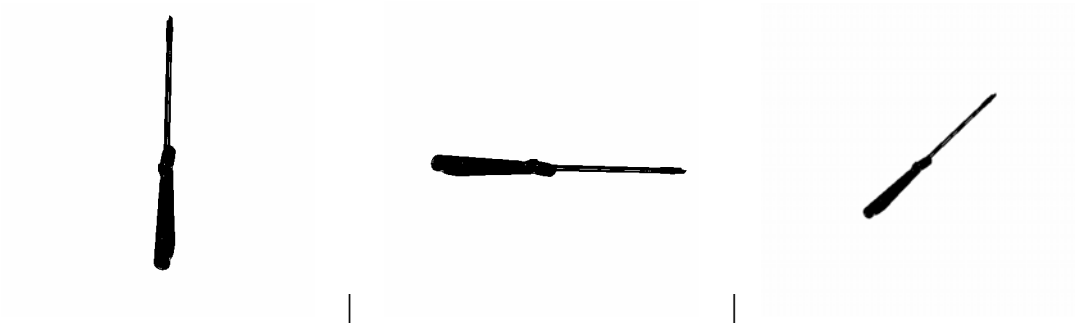
| | |

1º momento: 0.0095	1º momento: 0.0095	1º momento: 0.0095
2º momento: 9.0735e-005	2º momento: 9.0356e-005	2º momento: 8.9979e-005
3º momento: 9.6836e-008	3º momento: 9.6328e-008	3º momento: 9.5824e-008
4º momento: 9.6836e-008	4º momento: 9.6328e-008	4º momento: 9.5824e-008
5º momento: 9.3772e-015	5º momento: 9.2791e-015	5º momento: 9.1822e-015
6º momento: 9.2241e-010	6º momento: 9.1566e-010	6º momento: 9.0896e-010
7º momento: -9.3772e-015	7º momento: -9.2791e-015	7º momento: -9.1822e-015

Valores normais:

1º momento: 4.6538	1º momento: 4.6559	1º momento: 4.6580
2º momento: 9.3076	2º momento: 9.3118	2º momento: 9.3159
3º momento: 16.1502	3º momento: 16.1555	3º momento: 16.1608
4º momento: 16.1502	4º momento: 16.1555	4º momento: 16.1608
5º momento: 32.2771	5º momento: 32.2874	5º momento: 32.2976
6º momento: 20.8040	6º momento: 20.8114	6º momento: 20.8187
7º momento: 32.4768	7º momento: 32.4875	7º momento: 32.4982

Fenda:



| | |

1º momento: 0.0098	1º momento: 0.0098	1º momento: 0.0097
2º momento: 9.5726e-005	2º momento: 9.5322e-005	2º momento: 9.4920e-005
3º momento: 1.0315e-007	3º momento: 1.0261e-007	3º momento: 1.0206e-007
4º momento: 1.0315e-007	4º momento: 1.0261e-007	4º momento: 1.0206e-007
5º momento: 1.0640e-014	5º momento: 1.0528e-014	5º momento: 1.0417e-014
6º momento: 1.0092e-009	6º momento: 1.0018e-009	6º momento: 9.9438e-010
7º momento: -1.0640e-014	7º momento: -1.0528e-014	7º momento: -1.0417e-014

Valores normais:

1º momento: 4.6270	1º momento: 4.6291	1º momento: 4.6312
2º momento: 9.2540	2º momento: 9.2583	2º momento: 9.2625
3º momento: 16.0871	3º momento: 16.0924	3º momento: 16.0977
4º momento: 16.0871	4º momento: 16.0924	4º momento: 16.0977
5º momento: 32.1535	5º momento: 32.1639	5º momento: 32.1742
6º momento: 20.7141	6º momento: 20.7215	6º momento: 20.7289
7º momento: 32.3481	7º momento: 32.3589	7º momento: 32.3697

A tabela 3.1 abaixo traz o intervalo dos valores dos momentos obtidos para cada uma das imagens, mostrando que é possível distinguir as imagens através deles e utilizá-los para reconhecer padrões.

TABELA 3.1 – Intervalo dos Momentos Invariantes.

Momento\ Objeto	Alen	Alicate	Inglesa	Fenda
1º momento	0.0007 à 0.0007	4.2600 à 4.2623	4.6538 à 4.6580	4.6270 à 4.6312
2º momento	0.0000	8.5201 à 8.5245	9.3076 à 9.3159	9.2540 à 9.2625
3º momento	32.7408 à 34.3106	15.9496 à 15.9558	6.1502 à 6.1608	16.0871 à 6.0977
4º momento	32.9791 à 33.3870	15.9496 à 15.9558	16.1502 à 16.1608	16.0871 à 16.0977
5º momento	36.0437 à 36.0437	31.8834 à 31.8957	32.2771 à 32.2976	32.1535 à 32.1742
6º momento	16.8002 à 16.9562	20.2096 à 20.2181	20.8040 à 20.8187	20.7141 à 20.7289
7º momento	36.0437 à 36.0437	32.0694 à 32.0820	32.4768 à 32.4982	32.3481 à 32.3697

A partir dos dados apresentados na tabela 3.1, um sistema de classificação poderia ser treinado para reconhecer os objetos de acordo com a faixa numérica apresentada. Cada combinação obtida através da analise conjunta das sete faixas é única para cada objeto. Um objeto que apresentasse a primeira entrada na faixa de 4.2600 à 4.2623, a segunda na faixa de 8.5201 à 8.5245 e assim por diante, seria classificado como sendo um alicate. O mesmo poderia ser feito com os valores obtidos para os outros objetos para que o classificador os reconhecesse da mesma forma.

3.3 Metodologia para o cálculo dos momentos invariantes

O método mais simples de extração de características consiste em utilizar a própria imagem como vetor de características, conhecido como mapa de *bits*. O mapa de *bits* tem o problema de não apresentar nenhum tipo de invariância embutida no processo de extração, além de ser muito sensível à variabilidade na forma dos objetos. Isso pode ser corrigido através de métodos de segmentação ou classificação que levem esses problemas em consideração.

Outro problema é que um bom mapa de *bits* possui uma dimensão muito grande. Outros atributos facilmente computados são as projeções horizontal e vertical, que são utilizados para imagens binárias. A projeção horizontal y(xi) é o número de *pixels* de objeto (*pixels* iguais a 1, por exemplo) que possuem a coordenada x igual a xi. A projeção vertical é similar, porém para as coordenadas y. Esses atributos porém não são capazes de capturar muitas informações sobre alguns aspectos importantes da imagem, como a forma do objeto, e são normalmente utilizados com outros objetivos, como correção de ângulo e segmentação de imagens binárias. Mesmo assim, projeções e mapas de *bits* são algumas das características muito utilizadas em sistemas de reconhecimento de padrões ainda hoje (47).

No exemplo dado acima implementado em Matlab é utilizado o método de projeção horizontal e vertical. Na arquitetura implementada é utilizado o mesmo

método, uma vez que a arquitetura para extração e binarização da imagem já fornece como resultado uma imagem binária.

3.4 Considerações finais

Os momentos invariantes de Hu têm a poderosa vantagem frente aos outros métodos para reconhecimento de padrões de serem invariantes a escala, rotação e translação. Seu uso em sistemas de reconhecimento de padrões é amplo até os dias atuais (48) (49). Como todos os métodos com esse mesmo propósito, os momentos de Hu têm um custo computacional muito grande, estando aí a grande importância em se procurar propor novas arquiteturas para o cálculo de extração dos momentos em imagens.

4. Arquitetura para extração de momentos invariantes em imagens binárias

4.1 Considerações Iniciais

Neste capítulo será apresentada a arquitetura para aquisição e armazenamento de imagens binárias utilizada no projeto e também a arquitetura para extração de momentos invariantes desenvolvida, os testes e implementações realizados para a validação do trabalho e sugestões para a utilização e trabalhos futuros.

4.2 Arquitetura para aquisicão e armazenamento de imagens binárias

4.2.1 Introdução

Nesta seção será apresentada a arquitetura desenvolvida que é utilizada no projeto para a extração e armazenamento de imagens na forma binária.

A arquitetura para aquisição e armazenamento das imagens binárias a ser utilizada nesse projeto tem como base o sistema de aquisição de imagens de vídeo desenvolvido por PEDRINO (13). Uma nova placa de circuito impresso foi desenvolvida, utilizando componentes mais modernos e também o FPGA escolhido, assim como conectores para conexão com uma placa PCI. Ou seja, a arquitetura responsável por fazer a aquisição e armazenamento de imagens que é utilizada pela arquitetura final é essa mesma, apenas com a introdução de um limiar para a binarização da imagem, tendo sido desenvolvido no projeto de mestrado uma nova parte, ou um novo módulo, para a extração de características invariantes das imagens armazenadas, formando assim um sistema completo para aquisição, armazenamento e cálculo de momentos invariantes.

4.2.2 Características Principais

O sistema original é composto basicamente por: um separador de sincronismo, um conversor A/D, memórias, Unidade de controle, conversor D/A e uma saída de vídeo. A maior parte desses componentes foi reavaliado e novos componentes, mais modernos, foram utilizados no projeto. O diagrama de blocos do sistema de aquisição de vídeo é mostrado na figura 4.1.

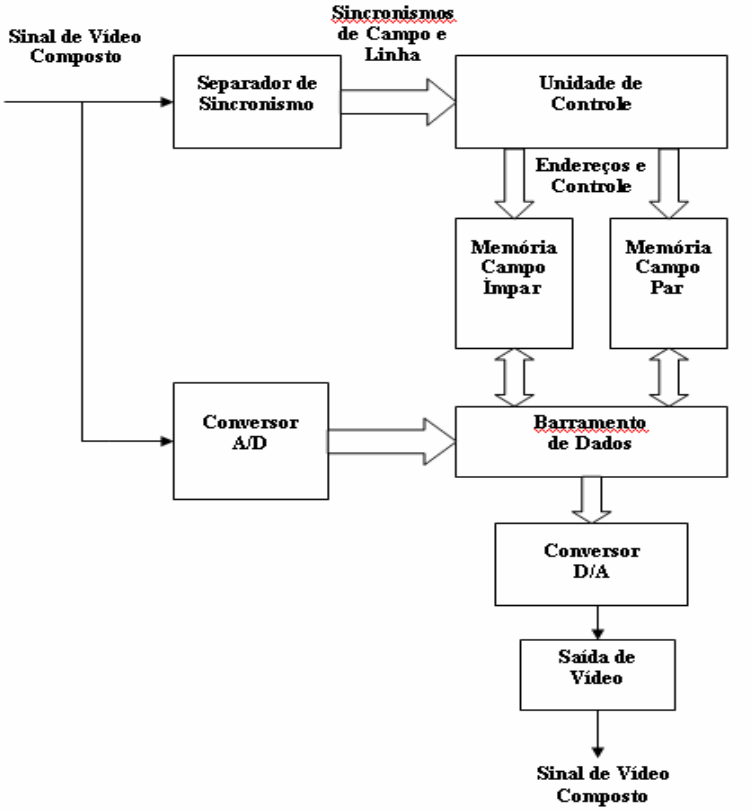


FIGURA 4.1 – Diagrama em blocos do circuito de aquisição, armazenamento e exibição de imagens monocromáticas.

O separador de sincronismo originalmente utilizado foi o EL4581C da empresa Elantec, cuja função é extrair as informações de sincronismo vertical, horizontal e de tipo de campo (par ou ímpar) de um sinal de vídeo composto padrão. Foi mantido o uso desse mesmo separador de sincronismo.

Uma vez que a informação de uma linha de vídeo é de aproximadamente 53 μs, para se obter 512 amostras nesse tempo é necessário utilizar um conversor A/D capaz de realizar a conversão em aproximadamente 100 ns, o que equivale a 10 milhões de amostras por segundo. Assim, devemos utilizar conversores analógico/digitais do tipo *flash*, que são os mais indicados para a digitalização de sinais de vídeo por permitirem altas taxas de conversão. Dessa forma, originalmente, foi utilizado o conversor A/D Bt218KP20 de 8 *bits* do tipo *flash* da *BrookTree*. O modelo utilizado neste projeto, entretanto, foi o TCL5540 da *Texas Instruments*, que possui as mesmas características que o conversor originalmente utilizado, mas possui encapsulamento SMD. Na figura 4.2 temos o diagrama de tempos do conversor A/D TLC5540.

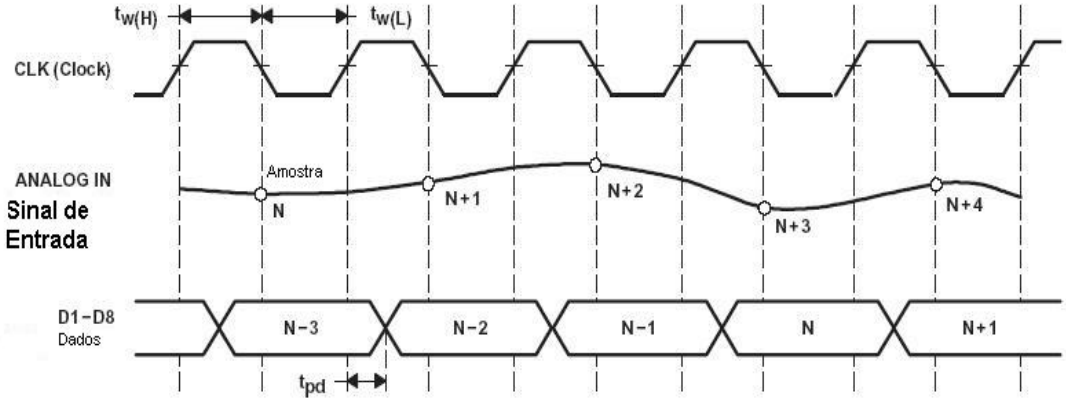


FIGURA 4.2 – Diagrama de tempos do conversor A/D TLC5540. O sinal é amostrado na borda de descida do *clock* e os dados ficam disponíveis na terceira borda de descida.

Considerando-se que o sinal de vídeo é entrelaçado, originalmente utilizou-se duas memórias RAMs estáticas para armazenar cada um dos campos. A memória RAM originalmente adotada foi a TC551001 de 70ns e 131072 palavras de 8 *bits* da empresa Toshiba. O modelo a ser utilizado no projeto é o KM681000E também de 70ns e 131072 palavras de 8 *bits* da empresa Samsung. Assim como o novo conversor A/D adotado, esse modelo de memória também possui encapsulamento SMD.

Para implementar a unidade de controle originalmente foi utilizado o CPLD EPM7128SLC84-7 da família MAX 7000S da Altera e para programar o CPLD foi utilizado o software MAX+PLUS II, também da Altera. No presente projeto foi utilizado o mesmo dispositivo e para programá-lo foi utilizado o *software* Quartus II, também da Altera.

O conversor D/A utilizado originalmente foi o CA3338 de 8*bits* da empresa Intersil, que utiliza uma rede R2R e é específico para aplicações que envolvem sinais de vídeo. Este conversor pode operar numa taxa de até 50 MSPS. O conversor adotado para ser usado nesse projeto será o mesmo modelo utilizado originalmente. O diagrama de tempos do conversor CA3338 pode ser visto na figura 4.3.

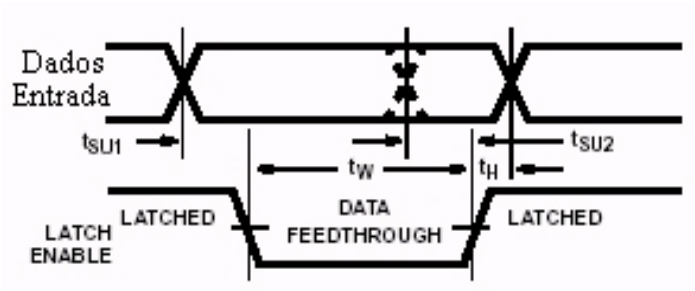


FIGURA 4.3 – Diagrama de tempos do conversor D/A CA3338.

A saída de vídeo implementada tem por objetivo reconstituir o sinal de vídeo composto na saída de dados, o sinal de saída do conversor D/A e o sinal de sincronismo composto proveniente do separador de sincronismo. Este circuito pode ser visto no diagrama esquemático do anexo B.

4.2.3 Considerações Finais

Todos os níveis de tensão utilizados no projeto original, e também neste projeto são compatíveis com o nível TTL (5V). As informações técnicas dos diversos dispositivos utilizados neste projeto podem ser encontradas nas folhas de especificações específicos de cada fabricante sobredito. As impedâncias de entrada e de saída do sinal de vídeo são de 75 *ohms*. Também, não abordamos sobre as informações de cor do sinal de vídeo composto porque foi utilizado um sinal de vídeo fornecido através de uma câmera monocromática compatível com o formato RS170. O *clock* de 10 MHz do circuito foi obtido pela divisão por 2 da frequência

fornecida por um módulo oscilador de cristal de 20 MHz. Através do *bit* menos significativo de um circuito contador pudemos realizar esta tarefa. Para eliminar possíveis distorções na imagem na direção vertical devido à falta de sincronismo entre o *clock* e o início de cada linha de vídeo, tornou-se necessário reiniciar o circuito contador a cada início de linha.

Uma foto das placas finais desenvolvidas a fim de implementar a nova arquitetura e a arquitetura para aquisição e armazenamento de imagens pode ser vista no anexo D.

4.3 Arquitetura Desenvolvida

4.3.1 Considerações Iniciais

Nesta seção será apresentada a arquitetura para extração de momentos invariantes implementada em VHDL.

O cálculo dos momentos invariantes consiste em basicamente três etapas. O cálculo da área; O cálculo dos momentos centrais; O cálculo dos momentos invariantes em si. Uma vez que a arquitetura de extração e armazenamento prove a imagem já na forma binária e essa informação pode ser recuperada das memórias em

forma de uma matriz de *pixels*, a metodologia de cálculo através de projeções horizontal e vertical é perfeitamente aplicável. O sistema implementado está descrito na figura 4.4, a seguir.

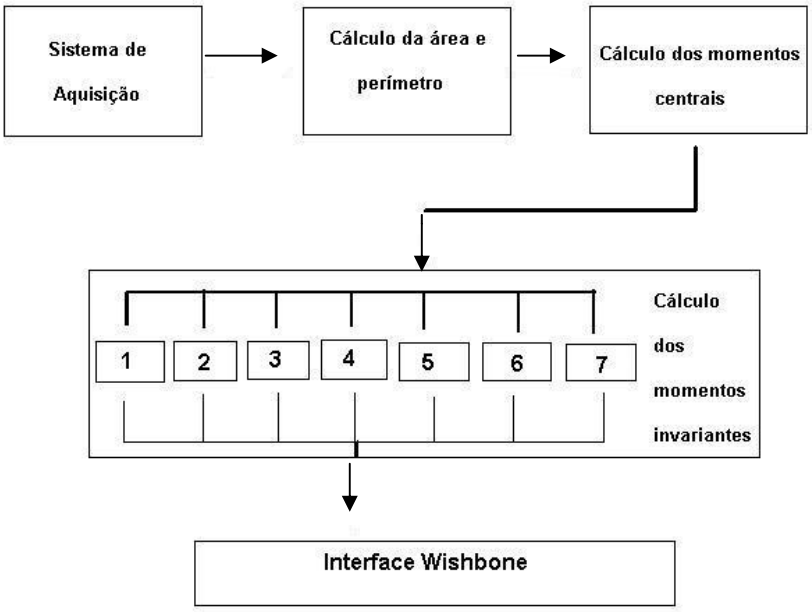


Figura 4.4 Diagrama de blocos da Arquitetura implementada

O sistema de aquisição é o anteriormente apresentado que foi desenvolvido por PEDRINO (13) e foi para uso no projeto. A tarefa de descrição de *hardware* foi desenvolvida no ambiente Quartus II da Altera, fazendo o uso de todos os recursos disponíveis até o momento. A interface *Wishbone* foi criada de acordo com as normas do padrão *Wishbone* estabelecidas segundo OPENCORES (50).

Uma vez calculados os momentos centrais, que servem como base para o cálculo dos momentos invariantes, é possível se fazer o cálculo dos momentos

invariantes de forma paralela, gerando um ganho de desempenho esperado (51) frente às arquiteturas tradicionais, como já apresentado no Capítulo 2.

O sistema se encontra distribuído em nove módulos, entre eles quatro pacotes adicionais para o uso de variáveis com ponto flutuante, três desenvolvidos para efetuar o cálculo dos momentos e dois para a interface *wishbone*. Cada um desses módulos é apresentado separadamente nos anexos de E a I, sendo que os pacotes adicionais encontram-se em VHDL-200x (52).

A seguir vamos descrever o funcionamento de cada um dos três módulos responsáveis pelas etapas de cálculo dos momentos invariantes e em seguida os responsáveis pela interface *wishbone*. O funcionamento dos módulos para cálculo dos momentos invariantes segue a lógica da função Matlab apresentada no anexo C.

4.3.2 Cálculo da área e médias em x e y

O cálculo da área da imagem consiste na soma dos valores de todos os *pixels* válidos, gerando assim um valor em quantidade de *pixels*. O cálculo das médias em X e em Y são valores iniciais utilizados para o posterior cálculo dos momentos centrais. O procedimento que realiza o cálculo dos mesmos encontra-se no ANEXO E, juntamente com todo o código do arquivo “calcumomentos.vhd” que é o topo hierárquico do sistema desenvolvido.

O procedimento que realiza o cálculo da área e médias em *x* e *y* denomina-se *are*. A variável *ar* recebe um novo valor a cada ciclo de *clock*, atualizado com a soma dos pixels anteriores ao do pixel atual que é dado pelo valor da entrada *mein* vinda das memórias. Ao mesmo tempo, os valores de *mtpx* e *mtpy* também são calculados e atualizados a cada ciclo de *clock*, sendo que *mtpx* guarda a soma dos valores dos pixels multiplicados pelo índice da coluna da imagem, considerando a imagem como uma matrix de 512x512 pixels, enquanto *mtpy* guarda a soma do valor dos pixels multiplicados pelo índice da linha da imagem.

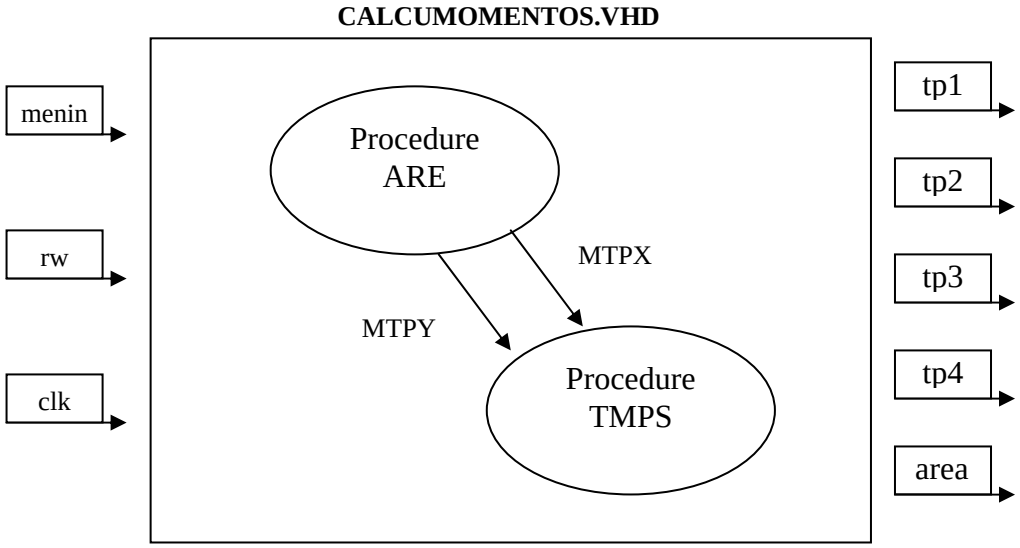


Figura 4.5 Diagrama de blocos do módulo de cálculo da área e médias em X e Y

Uma vez que o procedimento encontra o final de um quadro, através da variável *contlin*, o mesmo é encerrado e os valores finais de *ar*, *mtpx e mtpy* são retornados ao programa. O valor final de *mtpx* e *mtpy* é correspondente as variáveis *meanx* e *meany* encontradas na função Matlab do anexo C.

4.3.3 Cálculo dos momentos centrais

Uma vez terminado o cálculo da área e das médias em *x* e *y*, uma nova varredura é feita nas memórias para leitura dos valores dos pixels da imagem no procedimento chamado *tmps*, que também encontra-se no módulo do anexo E. Os valores gerados por esse procedimento são valores intermediários que serão posteriormente normalizados pela área nas funções que realizam o cálculo dos momentos centrais. Esses valores são gerados através da multiplicação e soma dos pixels da imagem por valores obtidos através de operações realizadas entre o valor das médias *x* e *y* e os índices dos pixels tanto em linha quanto em coluna.

Uma vez que o procedimento encontra novamente o final do quadro da imagem, através do valor da variável contadora de linhas *contlin*, os valores de *tp1, tp2, tp3 e tp4* são retornados ao programa para servirem como entradas para o cálculo dos momentos centrais que é realizado pelas funções *n20, n02, n11, n30, n12, n21 e n03*, que estão presentes no módulo denominado *intermed* que está apresentado no anexo F.

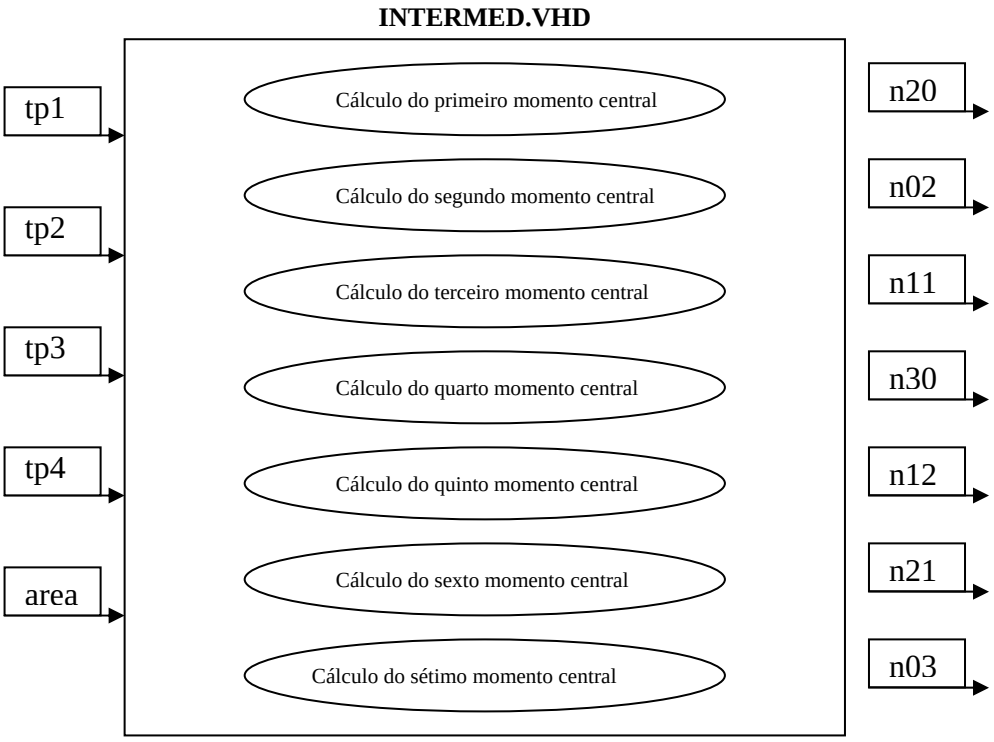


Figura 4.6 – Diagrama de blocos do módulo de cálculo dos momentos centrais

As funções de cálculo dos momentos centrais nada mais fazem do que a normalização pela área elevada ao quadrado, nos casos de *n20, n02 e n11*, e da raiz quadrada da área elevada a quinta potência nos casos restantes, dos valores intermediários calculados pelo procedimento *tmps*. O cálculo desses valores corresponde também ao modo como é apresentado na função matlab do anexo C.

4.3.4 Cálculo dos momentos invariantes

Uma vez gerados os momentos intermediários, o cálculo dos momentos invariantes é feito através da instanciação de funções implementadas no módulo presente no anexo G. Através de operações de soma e multiplicação aplicadas aos valores dos momentos centrais *n20, n02, n11, n30, n12, n21 e n03* as funções calculam o valor de cada um dos momentos invariantes de forma paralela.

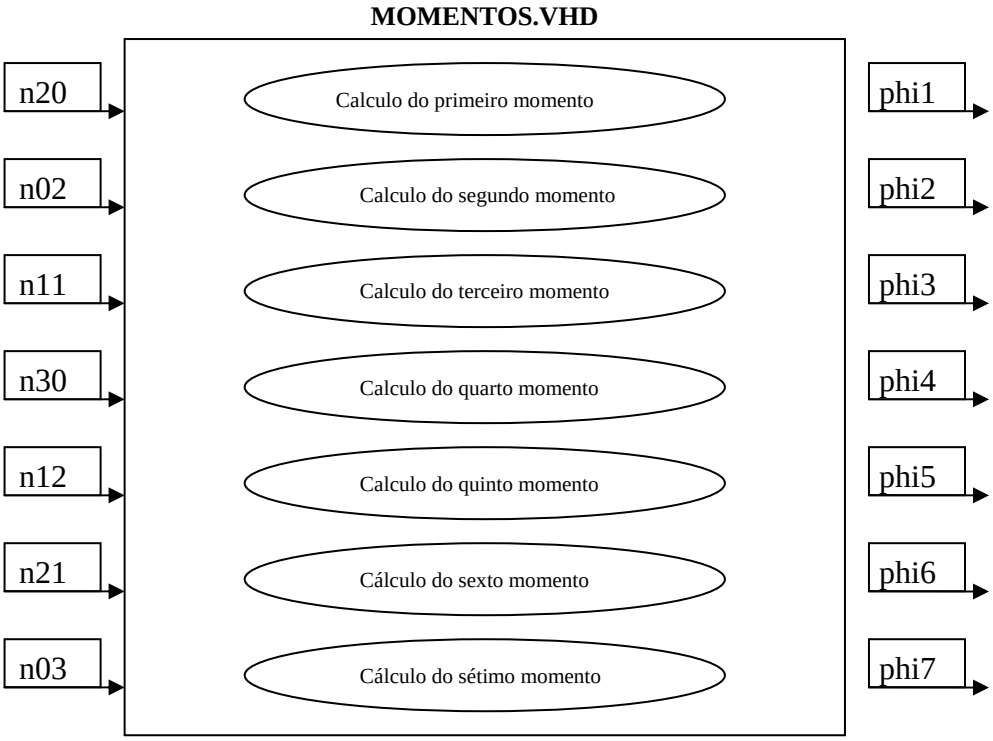


Figura 4.7 – Diagrama de blocos do módulo de cálculo dos momentos finais

Foram feitas simulações em *software* para a validação tanto das funções de cálculo dos momentos centrais quanto dos momentos finais. As mesmas estão apresentadas no anexo J. Os valores de *tstout1 e tstout2* correspondem ao valor final do cálculo dos momentos através da inserção manual dos valores intermediários de *tp1, tp2, tp3 e tp4* obtidos através da função Matlab para o cálculo dos momentos.

4.3.5 Interface Wishbone

A interface *wishbone* está implementada pelos módulos apresentados nos anexos H e I. No anexo H encontra-se a definição de um bloco de memória RAM que é implementado em dois blocos de memória MK4 na FPGA. No anexo I encontra-se a definição dos sinais da interface *wishbone* e a implementação do bloco de memória pré-definido atrelado à interface *wishbone*.

Os sinais implementados da interface *wishbone* e seu funcionamento seguem as definições do padrão de acordo como apresentado no anexo A. A interface implementada comportasse como escrava em um sistema *wishbone* e pode ser lida como se fosse uma memória, aonde são armazenados os valores dos momentos invariantes calculados.

4.4 Implementações e testes práticos

4.4.1 Considerações Iniciais

Originalmente os testes práticos do sistema deveriam ter sido feitos na placa de circuito impresso desenvolvida com a FPGA da família Cyclone. Entretanto, após a compilação no *software* Quartus II da Altera foi constatado que esse modelo de FPGA não comportaria o sistema completo, sendo que os mesmos acabaram por ser feitos em um kit de desenvolvimento NIOS II da ALTERA, como descrito nos tópicos a seguir.

Nos tópicos seguintes será abordada essa questão e a metodologia que foi adotada para testar o sistema de modo prático contornando as dificuldades encontradas.

4.4.2 Tamanho do sistema e alternativas encontradas

Como pode ser verificado na figura 4.8 o sistema completo para extração de características invariantes de imagens binárias com interface *wishbone* e com o

sistema de captura e armazenamento de imagens usaria um total de aproximadamente quarenta e nove mil e quinhentos blocos lógicos, ficando assim completamente fora da realidade para uma Cyclone de cinco mil blocos lógicos como o modelo que foi escolhido, apesar de cinco mil blocos lógicos poder ser considerado um tamanho grande para uma FPGA principalmente frente aos modelos anteriormente disponíveis.

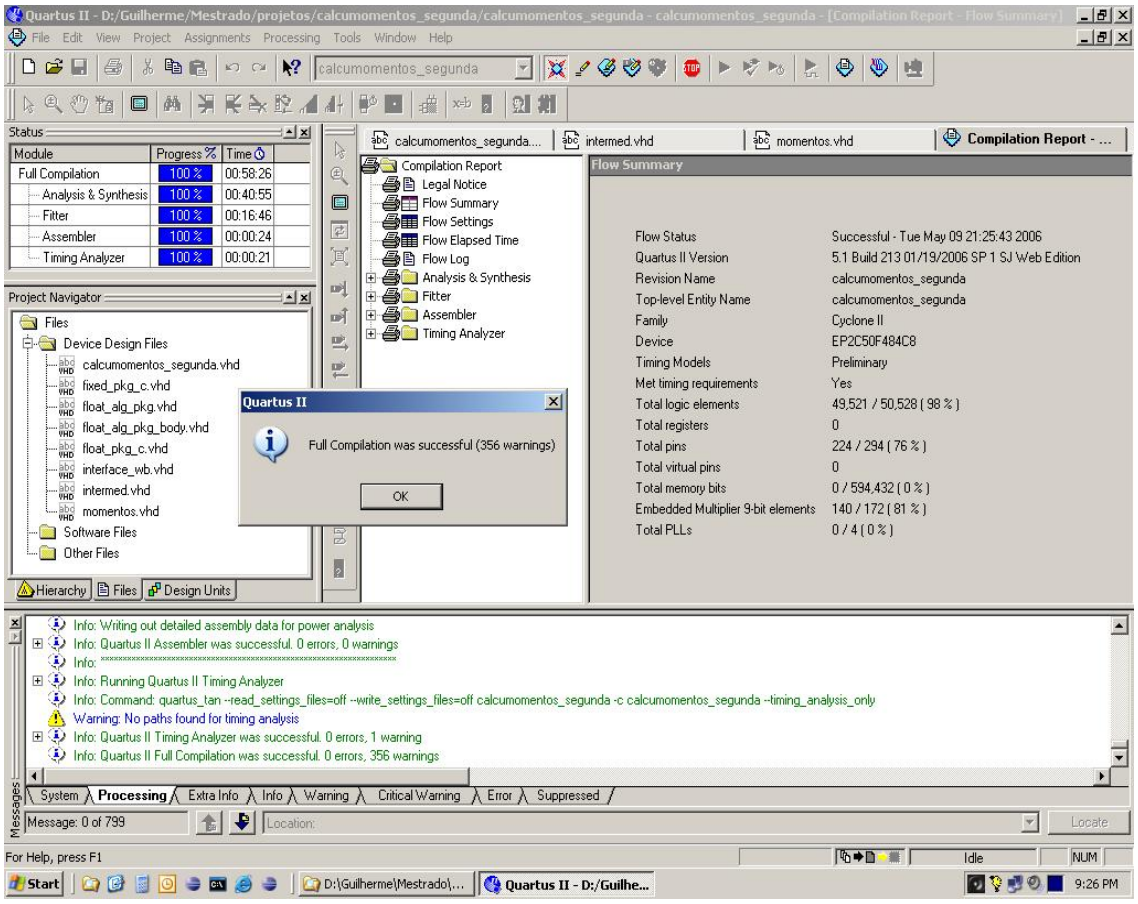


Figura 4.8 Compilação do sistema completo

Como alternativa para poder se fazer testes práticos foi proposta a “quebra” do sistema em dois módulos, o primeiro que faria a interface com o sistema de captura e armazenamento de imagens binárias e faria o cálculo dos momentos

intermediários, uma vez que a operação de leitura das memórias e cálculo dos momentos intermediários é feita simultaneamente. E um segundo módulo que faria o cálculo dos valores finais através dos valores intermediários inseridos diretamente no código.

Como pode ser verificado na figura 4.9, o primeiro módulo é exatamente o responsável pelo grande uso de blocos lógicos no sistema completo, não importando a quantidade de momentos intermediários que é calculada. Isso ocorre porque a grande capacidade lógica é requerida na operação de leitura das memórias quando ocorre o cálculo de valores como área e médias em X e Y da imagem. Esses processos fazem uso de matrizes de variáveis de ponto flutuante de grande dimensão (512 x 512). É necessário que as variáveis de entrada sejam tratadas como de ponto flutuante de 32*bits* devido a uma característica da biblioteca de ponto flutuante utilizada no projeto, que somente converte entradas do tipo "STD_LOGIC_VECTOR" de 32 *bits* em variáveis de ponto flutuante, não sendo possível assim utilizar entradas de oito *bits* assim como implementado no sistema de captura e armazenamento de imagens binárias. O resultado final é um módulo composto por trinta e dois mil e oitocentos blocos lógicos, algo que tornou impossível ser aplicado qualquer tipo de teste prático a essa parte do sistema.

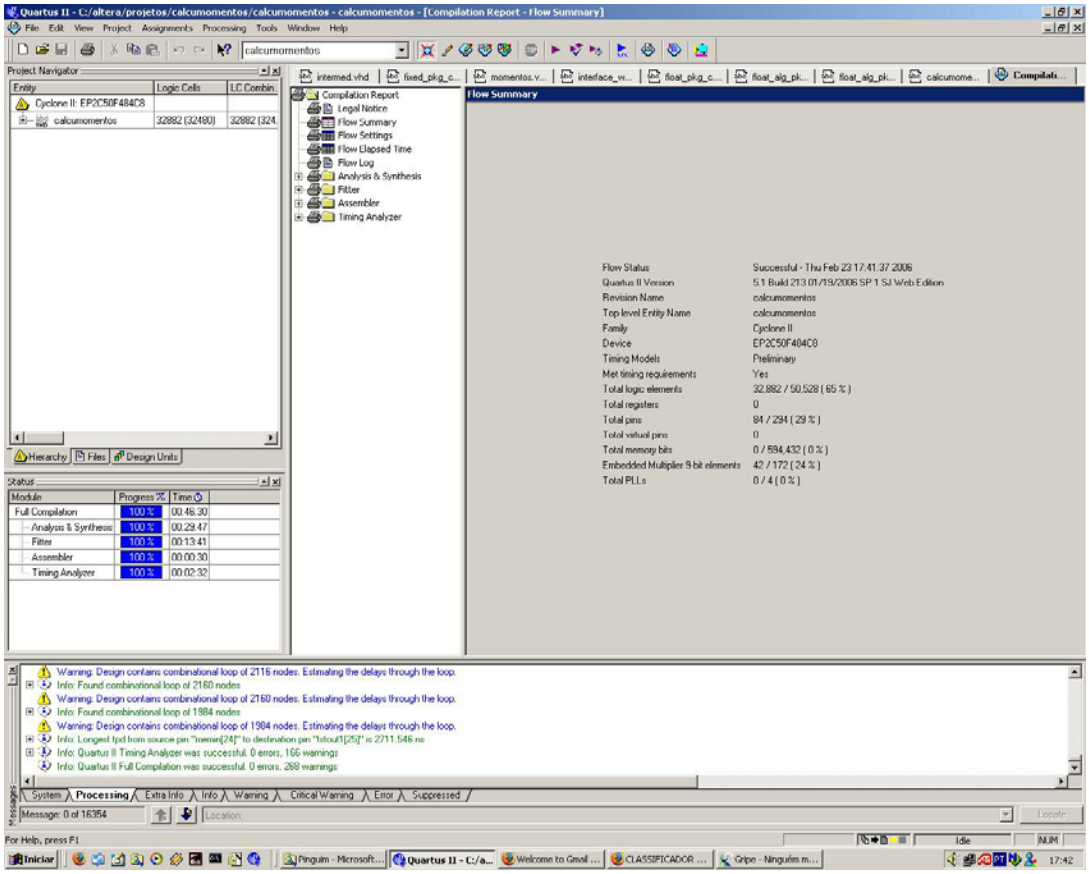


Figura 4.9 – Compilação do primeiro módulo do sistema

Já na figura 4.10 pode-se verificar o tamanho do sistema gerado na compilação do segundo módulo do sistema. O sistema gerado possui cerca de sete mil e duzentos blocos lógicos, o que, como dito anteriormente, tornou inviável a execução de testes na placa com a FPGA da família Cyclone. Visto isso, foi adotada como plataforma para testes práticos a placa de desenvolvimento do kit NIOS II da ALTERA, que possui uma FPGA da família STRATIX, com dez mil blocos lógicos programáveis disponíveis, como será detalhado no tópico a seguir.

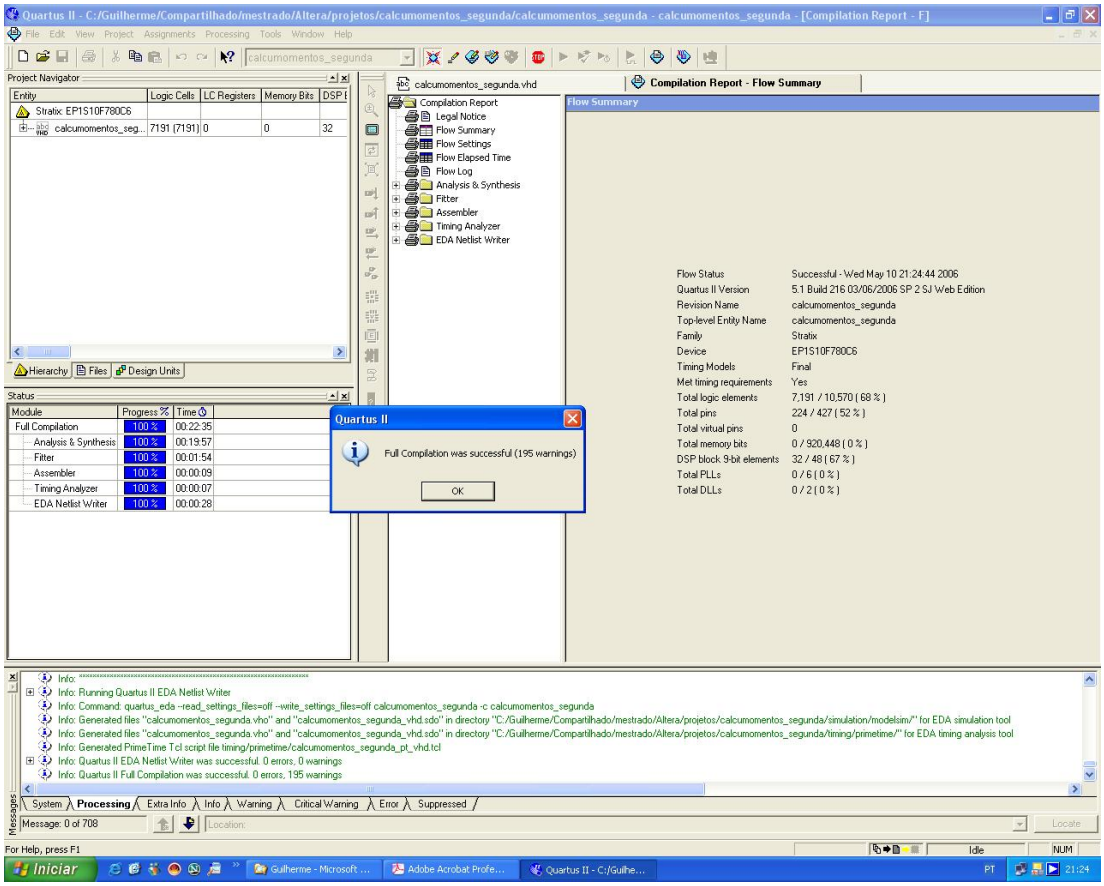


Figura 4.10 – Compilação do segundo módulo do sistema

Uma vez verificado isso, ficou claro que seria possível realizar testes práticos apenas com a segunda parte do sistema, inserindo os valores intermediários diretamente no código, a partir dos dados obtidos no Matlab.

4.4.3 Sistema de Hardware utilizado para teste

Originalmente o sistema de *hardware* utilizado para a implementação dos testes práticos do sistema deveria ser o que está exposto no anexo D. Apesar de

finalizada, a placa de circuito impresso com a FPGA da família Cyclone apresentou, além de uma densidade aquém da necessária, uma série de problemas que não conseguiriam ser contornados em tempo hábil para a conclusão do trabalho.

Como alternativa para a realização dos testes práticos do sistema foi então adotado um Kit NIOS da ALTERA que possui uma FPGA da família Stratix da ALTERA modelo EP1S10F780C6, disponível nos laboratórios de ensino do Departamento de Engenharia Elétrica. Esse modelo de FPGA e placa de teste mostram-se ser totalmente compatíveis com as nossas necessidades, uma vez que a FPGA possui dez mil blocos lógicos, e a placa de desenvolvimento apresenta conectores de expansão, através dos quais foi possível fazer leituras dos valores de saída do sistema com a utilização do analisador lógico presente no laboratório, resultando no sistema apresentado nas figuras 4.11 e 4.12.

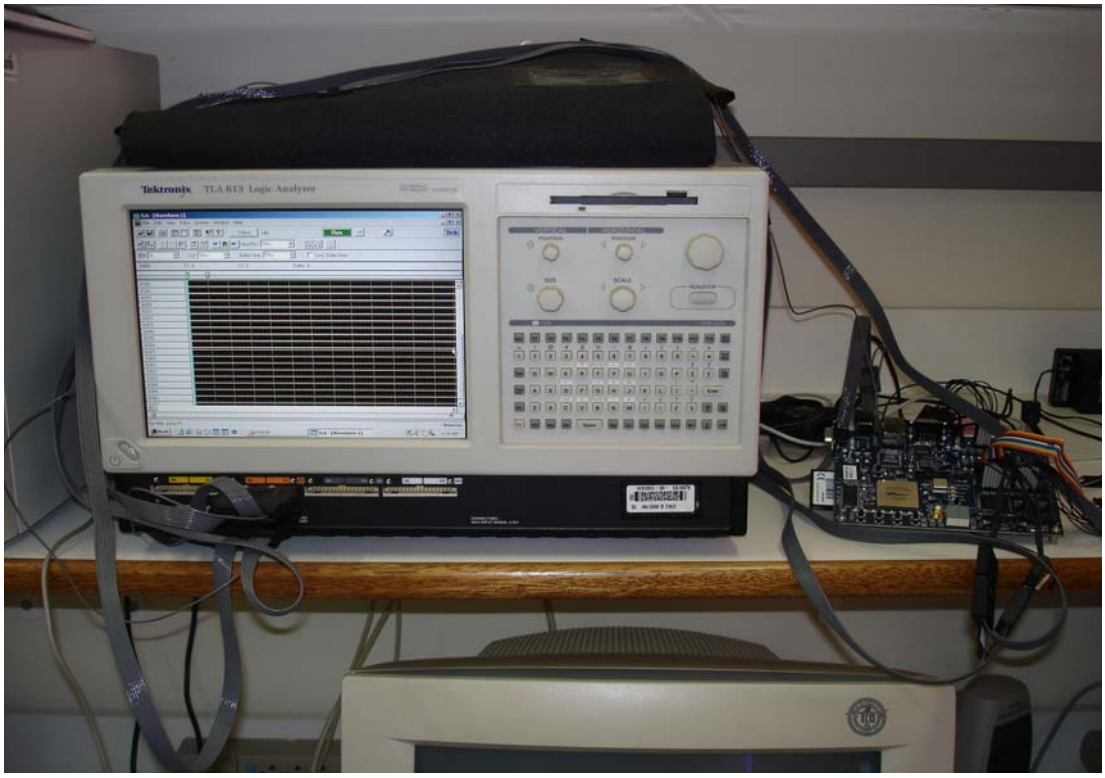


Figura 4.11 – Sistema utilizado para testes práticos I

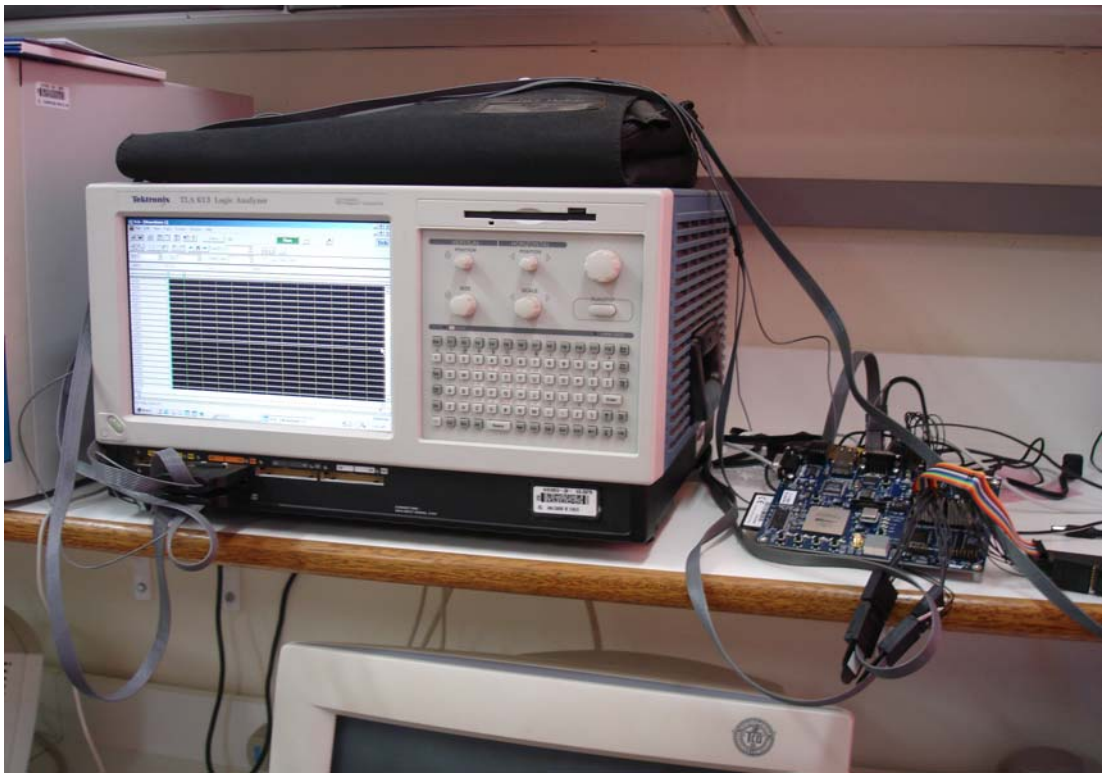


Figura 4.12 – Sistema utilizado para testes práticos II

4.4.4 Considerações finais

Apesar de todas as dificuldades encontradas, foi possível realizar testes práticos com o sistema, ou pelo menos parte dele, a fim de validar as simulações efetuadas anteriormente e apresentar novos desafios aos desenvolvedores da biblioteca de ponto flutuante utilizada.

Foram obtidos resultados através de simulações realizadas no *software* Quartus II da ALTERA que serão apresentados como forma de validação da arquitetura.

Um módulo que não foi citado nas divisões do sistema é o responsável pela interface *wishbone*. Tal parte do sistema só poderia ser testada na pratica mediante implementação em conjunto com um outro sistema que apresenta-se uma interface *wishbone* mestre. Isso não foi possível de ser feito nesse sistema devido a limitação de capacidade dos FPGAs disponíveis em comportar o sistema como um todo, o que é imprescindível para o teste da interface.

4.5 Resultados Obtidos

Os primeiros resultados obtidos afim de validar o funcionamento da arquitetura foram obtidos através do simulador do *software* Quartus II da ALTERA. Foram inseridos os valores dos momentos intermediários no segundo módulo do sistema e obtidos resultados de acordo com a figura 4.13.

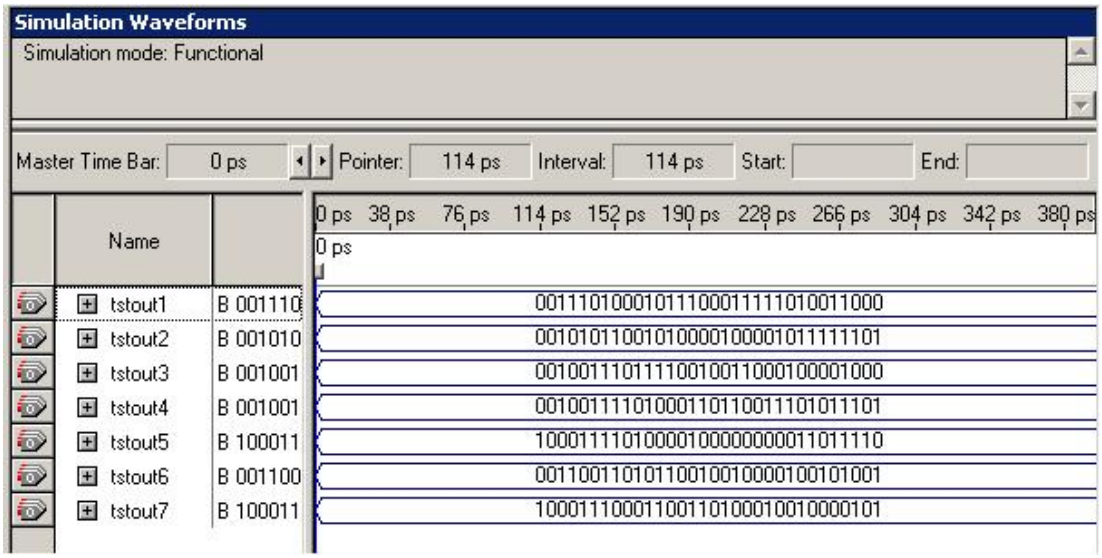


Figura 4.13 – Simulação no *software* Quartus II

Os valores obtidos para os momentos são comparados com os resultados obtidos no MATLAB de acordo com a tabela 4.1 a seguir:

TABELA 4.1 –Momentos Invariantes obtidos através de simulação.

Momentos	Valor Binário obtido	Valor Decimal correspondente	Valor obtido no Matlab
1º momento	0011101000101110001111010011000	0,00066469	0,00066469
2º momento	0010101100101000010000101111101	6,0435e-013	6,0435e-013
3º momento	0010011101111001001100010000 1000	1,0342e-015	1,0342e-015

4º momento	00100111101000110110011101011101	4,5354e-015	4,5354e-015
5º momento	10001111010000100000000011011110	-9,5655e-030	-9,5655e-030
6º momento	00110011010110010010000100101001	5,0555e-008	5,0555e-008
7º momento	10001110001100110100010010000101	-2,2097e-030	-2,2097e-030

Como pode ser observado, os resultados obtidos nas simulações conferem com os obtidos no programa MATLAB. As conversões foram feitas seguindo o padrão IEEE-754 para ponto flutuante.

O tempo necessário para a compilação do módulo e posteriormente da simulação é algo relevante, sendo que o tempo total ficou em 58 minutos. Desses, cinqüenta para a compilação, mais dois minutos para se gerar a *netlist* utilizada na simulação e posteriormente mais seis minutos para a simulação em si. Esses tempos foram obtidos utilizando-se uma maquina com um processador Pentium 4 HT de 3Ghz com 1Gbyte de memória RAM e rodando o sistema operacional *Windows XP*.

Foram obtidos resultados nas implementações feitas em *hardware* que puderam ser observados graças a utilização do analisador lógico. A seguir serão apresentadas as evidencias colhidas através de telas do *software* do analisador lógico que nos fornecem informações sobre cada um dos canais analisados.

Foram coletados os sinais de cada *bit* referente a cada um dos sete momentos invariantes nas imagens com as quais foram feitos os testes. Na figura 4.14 e 4.15

são apresentadas as telas coletadas para o primeiro momento da primeira imagem da chave alen.

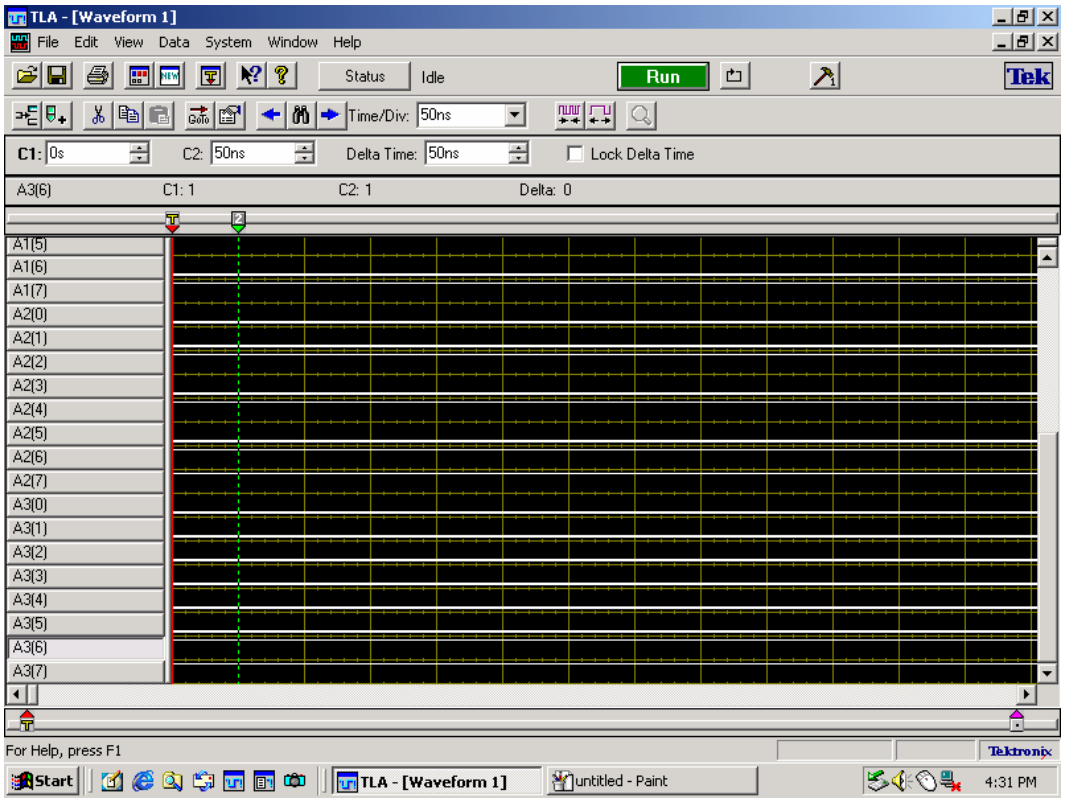


Figura 4.14 – Oito bits mais significativos do primeiro momento

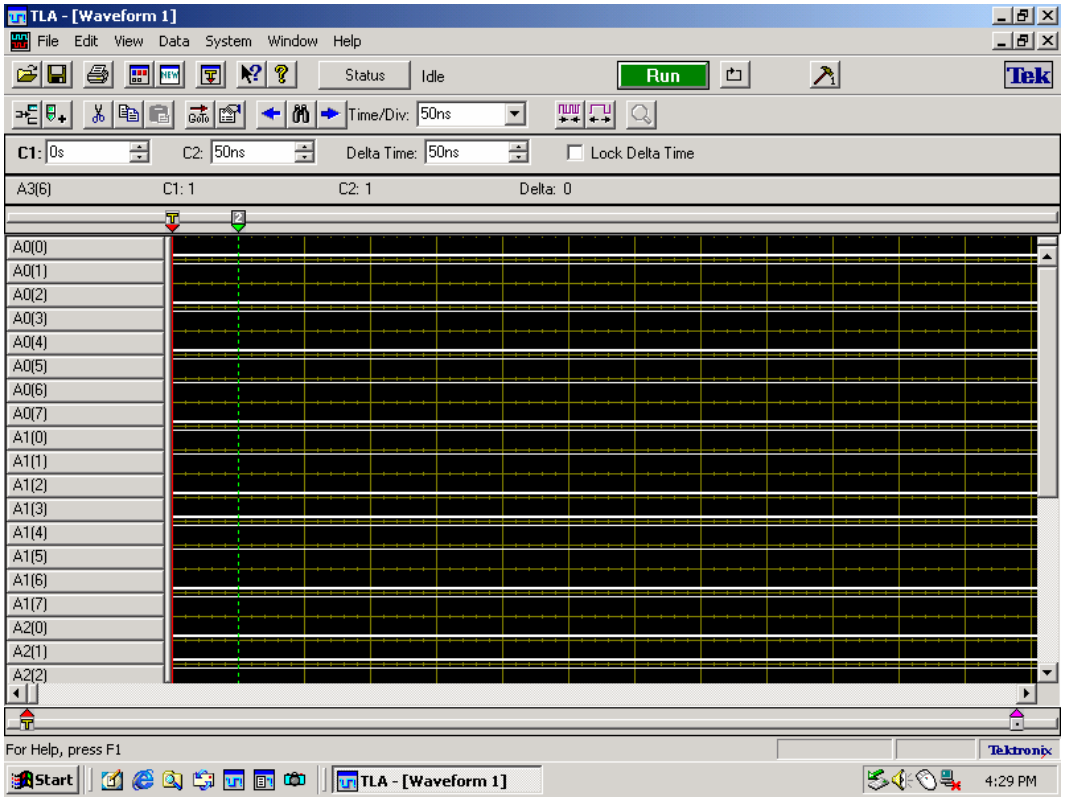


Figura 4.15 – Oito bits menos significativos do primeiro momento

Não serão apresentadas as figuras referentes aos demais momentos colhidas no analisador lógico, uma vez que as mesmas apresentam valores obtidos para os momentos que não coincidem com os obtidos nas simulações. Isso ocorre devido a uma falha gerada, de acordo com os desenvolvedores da biblioteca de ponto flutuante, pelo software da ALTERA, fazendo com que o comportamento de certas variáveis durante as operações de multiplicação, expoente e divisão se comportem de forma errada.

Esse problema foi relatado ao grupo de desenvolvimento. A resposta obtida foi que uma solução tentaria ser implementada, mas que tal tipo de problema é uma dificuldade que vem sendo apresentada exclusivamente pelo *software* da ALTERA, o

que gerou um certo grau de frustração, uma vez que não houve tempo hábil para chegar-se a uma solução.

4.6 Desempenho obtido

Outro dado interessante obtido através do software Quartus II foi o mapeamento lógico da FPGA escolhida para uma possível implementação do sistema completo e uma estimativa de tempo para o cálculo, levando em consideração o *clock* máximo suportado pela FPGA que é de 250Mhz, de acordo com a figura 4.16.

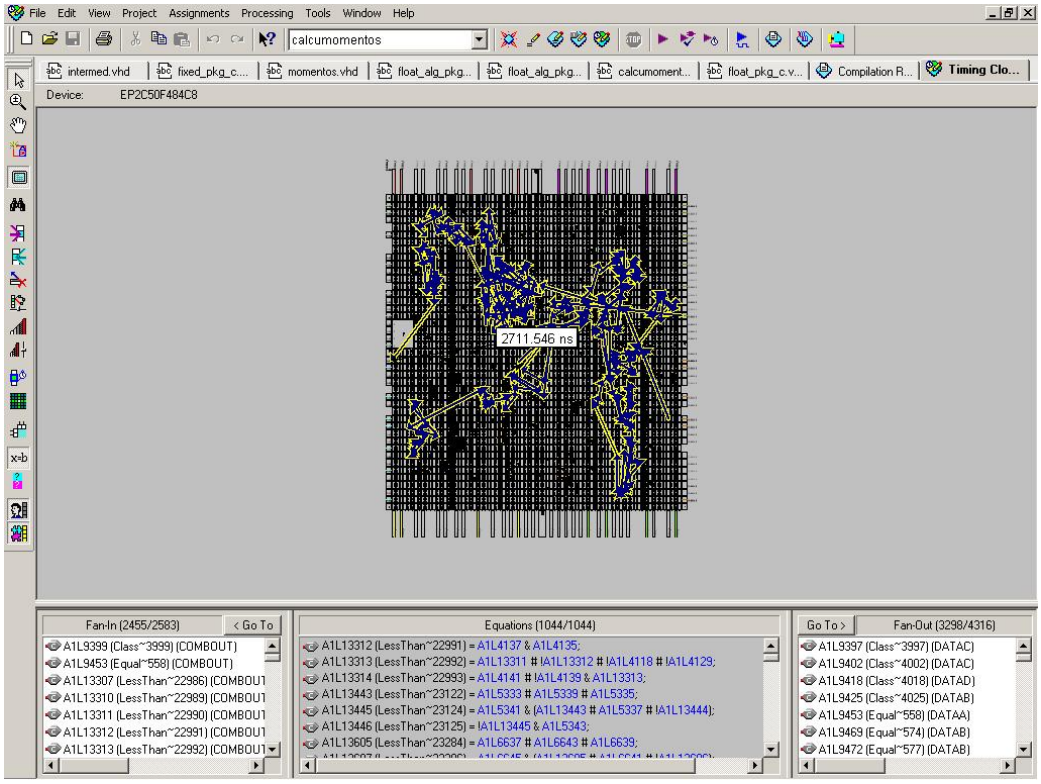


Figura 4.16 – Mapeamento lógico do sistema

É possível observar que o tempo máximo estimado entre a entrada de um dos *bits* de um dos *pixels* da imagem, até a saída do resultado final é de 2771.546ns.

Através de analise de tempo fornecida pelo próprio MATLAB, foi constatado que um microcomputador com as mesmas configurações apresentadas para a simulação do sistema leva em torno de 94000000.00ns para executar o programa implementado em MATLAB.

Isso vem comprovar o real ganho de desempenho que era esperado do sistema implementado em hardware frente a arquitetura PC. O ganho de desempenho obtido foi na ordem de 33.000% (trinta e três mil por cento).

5. Conclusões e Considerações Finais

O tamanho final do sistema foi algo que surpreendeu bastante nossas expectativas. A surpresa maior ficou por conta não do uso de números com ponto flutuante, mas pelo fato do uso de matrizes para cálculos com os mesmos ter se mostrado algo tão exigente do *hardware*.

O cálculo de momentos invariantes em dispositivos de lógica programável é um desafio. Trabalhar com números em ponto flutuante em sistemas desenvolvidos em VHDL é ainda algo pouco explorado, uma vez que oficialmente a linguagem VHDL não tem suporte a esse tipo de dados. Com o desenvolvimento de bibliotecas para o uso de tal tipo de dados, um grande leque de aplicações se abre para o desenvolvimento de futuros sistemas.

Apesar do uso de microcomputadores de ultima geração e das mais recentes versões dos softwares de desenvolvimento, a compilação do sistema mostrou-se demorada e muitas vezes inviável devido ao grande nível de complexidade do mesmo. Apesar disso, os resultados obtidos foram considerados satisfatórios frente às dificuldades encontradas e mostram-se de grande valor para o desenvolvedor do trabalho e para a comunidade cientifica.

O ganho de desempenho da arquitetura frente ao microcomputador tradicional surpreendeu. Não pelo fato em si, mas pela proporção medida. Mesmo com uma diferença muito grande de velocidade de relógio dos sistemas, a arquitetura mostrou-se extremamente mais rápida que o microcomputador com processador de ultima geração. Isso vem validar a proposta inicial e uma das motivações para o desenvolvimento do trabalho.

Por fim o trabalho apresentou resultados satisfatórios frente aos desafios que foram encontrados explorando esse novo horizonte que se mostra a nossa frente: o uso de matemática complexa em hardware programável de forma pratica e funcional. Esperamos que os benefícios do advento dessa nova tecnologia venham trazer: novos desafios, novas idéias e, com isso, muito mais benefícios para a humanidade.

5.1 Bibliotecas de Ponto Flutuante

A necessidade da utilização de uma biblioteca para os cálculos com números de ponto flutuantes é evidente, uma vez que a especificação e as bibliotecas oficiais da linguagem VHDL atual não possuem suporte a tal tipo de dados. Devido a complexidade dos cálculos a serem realizados no sistema implementado, o mesmo se tornaria completamente inviável não fosse o advento de tal recurso.

A biblioteca utilizada é uma versão preliminar da biblioteca oficial de que deve ser lançada juntamente com a mais nova especificação da linguagem VHDL que deve ser lançada nos próximos anos. O trabalho de mestrado fez uso e assim também contribuiu para o desenvolvimento da biblioteca, uma vez que todos os problemas relatados encontrados durante o desenvolvimento do trabalho foram compartilhados com o grupo desenvolvedor da mesma.

Esperamos ansiosamente que todos os problemas sejam resolvidos o mais rápido possível e que a nova especificação da linguagem VHDL com suporta a tipos ponto-flutuante seja lançada o quanto antes. Com certeza será um marco que abrirá uma infinidade de portas para soluções de problemas complexos como o apresentado nesse trabalho.

As bibliotecas utilizadas no sistema foram extraídas do site (VHDL-200X), que é um grupo oficial de desenvolvimento da IEEE. As bibliotecas desenvolvidas

por eles seguem a norma IEEE 754, que especifica um padrão para ponto flutuante em álgebra booleana. Mais uma vez, o desenvolvimento do trabalho ajudou no desenvolvimento dos pacotes, contribuindo assim para o crescimento científico oficial da linguagem VHDL, através da correção de problemas encontrados e de problemas de incompatibilidade resolvidos frente aos softwares de desenvolvimento da ALTERA.

5.2 Sugestões para trabalhos futuros

A implementação do sistema em dispositivos de outra empresa, como os da XILINX, por exemplo, poderia acabar com os problemas encontrados nas implementações práticas encontrados, de acordo com os desenvolvedores da biblioteca de ponto flutuante.

Seria interessante a construção, então, de uma placa com barramento PCI semelhante à desenvolvida no Uruguai, utilizando-se de uma FPGA de maior densidade afim de que a arquitetura completa pudesse ser implementada e testada juntamente com a interface wishbone.

Outro tópico interessante seria uma maior analise e estudo refinado das bibliotecas de ponto flutuante que estão sendo desenvolvidas pela IEEE. O trabalho

em conjunto com o time de desenvolvedores é possível e até estimulado pelos próprios.

Uma outra arquitetura, que fizesse a normalização dos momentos obtidos seria interessante afim de fornecer dados consistentes para um sistema classificador com o intuito de reconhecer padrões de objetos.

Finalmente um sistema que implementasse uma rede neural em conjunto com a arquitetura de cálculo de momentos poderia ser desenvolvido para compor um sistema completo de reconhecimento de padrões através da captura de imagens de objetos a partir de câmera CCD e placa PCI, tudo feito de forma integrada em um computador pessoal.

ANEXO A

INTERFACE WISHBONE E BARRAMENTO PCI

INTERFACE WISHBONE

1. Introdução

O presente documento resumo os aspectos mais importantes da revisão B.3 da especificação WISHBONE.

O objetivo da especificação WISHBONE é criar uma interface comum entre IP cores.

Ela define um padrão para a troca de dados entre módulos IP core. Não querendo especificar o funcionamento do core, mas apenas a forma de se comunicar com o mesmo ou entre os mesmos.

A arquitetura WISHBONE é análoga ao barramento de um microprocessador.

- Oferece uma solução flexível de integração
- Oferece vários tipos de ciclos e largura de barramento para situações distintas.
- Permite que uma mesma aplicação seja desenvolvida por vários fabricantes.

Características

As características mais importantes da especificação são:

- Baseado em protocolos padrão de transferência de dados
  - Ciclos Read/Write
  - Ciclos de transferência em bloco
  - Ciclos RMW (Read-Modify_Write)
- Suporta vários tipos de interconexão
  - Ponto a ponto
  - Barramento compartilhado
  - Switch de interconexões
- Protocolo de “aperto de mão” (Handshake) para regular a velocidade de transferência de dados.
- Suporta vários términos de ciclo
  - Normal
  - Com “retry”
  - Por erro
- Baseado na arquitetura mestre/escravo

Terminologia

A especificação WISHBONE define os seguintes termos:

- REGRA: Todas **devem** ser seguidas afim de assegurar a compatibilidade entre interfaces.

- RECOMENDAÇÕES: Quando aparece uma recomendação, recomenda-se segui-la. Não adotá-la pode resultar em uma perda de desempenho. Muitas recomendações estão baseadas na experiência acumulada dos pesquisadores que escreveram a especificação.

- SUGESTÃO: É um conselho que pode ser levado em conta pelo desenvolvedor. Pode ser útil, mas não é vital para o funcionamento da interface.

- PERMISSÃO: Indica quando algo pode ser feito ou não, mas a decisão de implementar ou não fica a cargo do desenvolvimento.

- OBSERVAÇÃO: Geralmente são textos que explicam alguma situação, não tendo nenhum significado além disso.

Convenção para o nome dos sinais

Todos os sinais possuem "_I" ou "_O", para indicar se são entradas ou saídas, respectivamente.

Todos os barramentos são indicados por nomes terminados em "()". Por exemplo: DAT_I().

Podem-se utilizar sinais especiais definidos pelo usuário, por exemplo: PAR_O. Estes sinais são chamados de *tags*. Eles possuem um tipo associado a eles, que indica em que momento do ciclo os mesmos possuem um valor válido.

2. Especificação da Interface

Documentação da interface

Ao se especificar a interface, deve-se incluir as seguintes informações.

1. Número da revisão WISHBONE com o qual o core é compatível. Neste caso B.3.
2. Tipo de interface: Mestre ou Escravo
3. O nome dos sinais definidos na interface WB.
4. No caso de suporte a sinais de erro.
   - Master (ERR_I) deve se indicar como é acionado
   - Slave (ERR_O) deve indicar quais são as situações que ativam o sinal
5. No caso de suporte a sinais de retry.
   - Master (RTY_I) deve se indicar como é acionado
   - Slave (RTY_O) deve indicar quais são as situações que ativam o sinal
6. Todas as interfaces que suportam TAGs devem indicar seu nome, tipo e funcionamento.
7. Largura do barramento de dados (8, 16, 32, 64).
8. Granularidade das portas.
9. Tamanho máximo de operação, não se sabendo fica sendo igual ao do barramento.

10. BIG ENDIAN ou LITTLE ENDIAN.
11. Se há restrições sobre o sinal CLK_I.

Níveis Ativos

Todos os sinais são ativos em nível alto.

Sinais na Interface

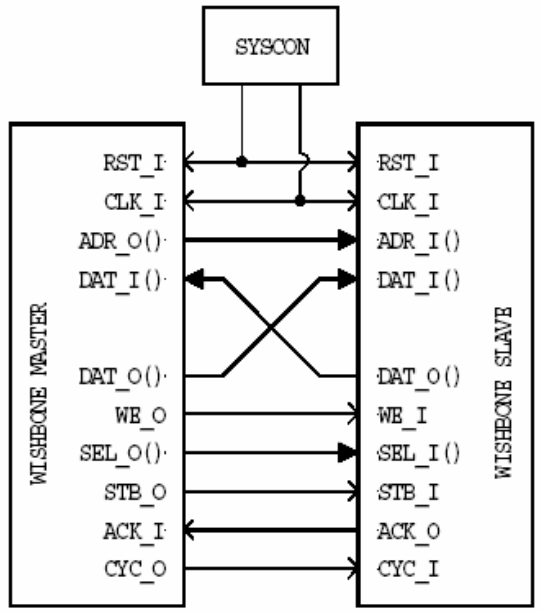
A idéia é que os sinais permitam conexões ponto-a-ponto, barramento compartilhado, etc.

São permitidos três ciclos básicos: Leitura, Escrita e RMW.

Permitem um aperto de mãos “handshake” para adequar a velocidade de transferência de dados e indicar erros e “retries”.

Os sinais não são bidirecionais, são sempre entradas ou saídas. Isso devido ao fato de alguns dispositivos de hardware ainda não suportarem sinais bidirecionais em suas lógicas internas ainda, por exemplo: os FPGAs da Altera.

Exemplo de interface e interconexões em uma arquitetura ponto-a-ponto:



Sinais comuns à Mestre e Escravo

- CLK_I: *Clock* de Entrada
- RST_I: Reset
- DAT_I() e DAT_O(): barramentos de entrada e saída de dados
- TGD_I() e TGD_O():

- Opcional
- Contém informação associada ao barramento de dados
- Válidos quando o sinal STB_O for ativo
- Ex: Paridade

Mestre

- ADR_O(N..n)
  - N: Limite superior dado pela largura do barramento de endereços
  - n: Limite inferior dado pela granularidade do barramento de dados
- CYC_O: Indica que está ocorrendo um ciclo válido no barramento. É ativado no começo do ciclo e permanece ativo até o final.
- WE_O: Indica se o ciclo é de escrita ou leitura
- STB_O: Indica que está ocorrendo um ciclo válido de transferência de dados
- ACK_I: Recebe a confirmação de uma transferência
- ERR_I: Recebe a indicação de um erro durante a transferência
- RTY_I: Recebe pedidos de re-transmissão de dados
- LOCK_O: Indica que o ciclo que está ocorrendo não pode ser interrompido
- SEL_O(): Associado a granularidade, indica aonde há ou aonde se esperam dados válidos no barramento
- TGA_O: Informação associada às linhas de endereço (Válido quando STB_O estiver ativo)
- TGC_O: Informação associada ao tipo do ciclo no barramento (Válido quando CYC_O ativo)

Escravo

- ADR_I(N..n)
  - N: Limite superior dado pela largura do barramento de endereços
  - n: Limite inferior dado pela granularidade do barramento de dados
- CYC_I: Indica o começo de um ciclo
- WE_I: Indica se a transferência é de leitura ou escrita
- STB_I: Indica que está ocorrendo um ciclo válido de transferência de dados
- ACK_O: Indica que a transferência foi realizada com êxito
- ERR_O: Utilizado para sinalizar um erro na transferência
- RTY_O: Utilizado para pedir uma nova tentativa de transferência
- LOCK_I: Indica que o ciclo que está ocorrendo não pode ser interrompido
- SEL_I(): Associado a granularidade, indica aonde há ou aonde se esperam dados válidos no barramento
- TGA_I: Informação associada às linhas de endereço (Válido quando STB_I estiver ativo)
- TGC_I: Informação associada ao tipo do ciclo no barramento (Válido quando CYC_I ativo)

3. WISHBONE clássico

A partir da revisão B.3, uma nova modalidade de transferência foi agregada e foram instituídas TAGs de sinalização.

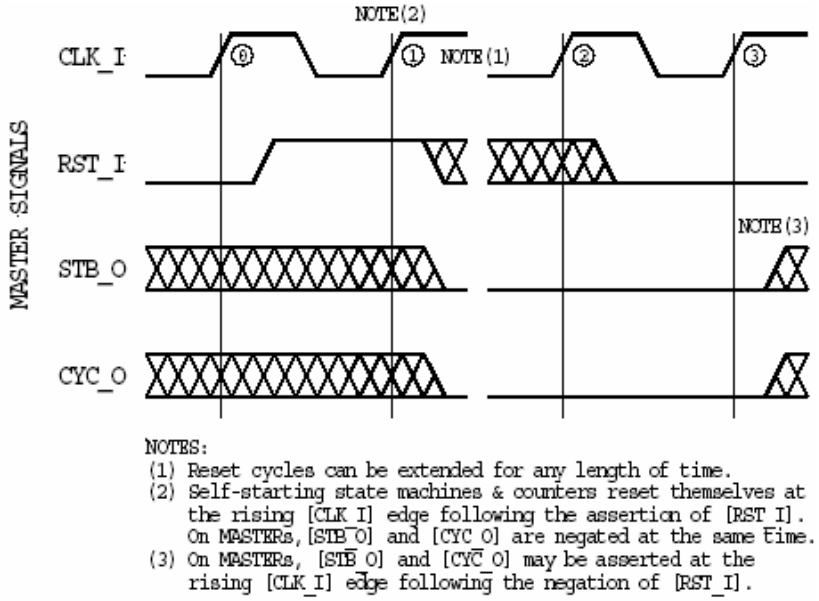
As mudanças permitem uma maior velocidade na transferência de dados, mas mantém a compatibilidade com as revisões anteriores do padrão Wishbone.

A maneira de se realizar transferências nas versões anteriores é chamada de "Wishbone clássico".

Funcionamento Geral

Reset

O Reset é feito de forma síncrona.



Todas as interfaces WISHBONE devem ser iniciadas no primeiro flanco de subida do sinal RST_I.

Inicio de um ciclo de transferência

O sinal CYC_O ativo indica que há um ciclo válido. Quando CYC_O está em nível lógico baixo, nenhum outro sinal do Mestre tem validade.

Protocolo de "aperto-de-mão" (Handshake)

O "handshake" para transferência de dados é extremamente simples. O Mestre ativa o sinal STB_O e o mantém assim até que o Escravo ative algum dos sinais ACK_I, ERR_I ou RTY_I. Quando isso ocorre o Mestre desativa o sinal.

Os sinais de resposta ACK_O, ERR_O e RTY_O só devem ser gerados se estirem ativos ambos CYC_I e STB_I.

As interface Escravo **devem** ser projetadas para que os sinais ACK_O, ERR_O e RTY_O sejam ativados e desativados em resposta a STB_O.

Uso de TAGS

TABELA: Uso de TAGS

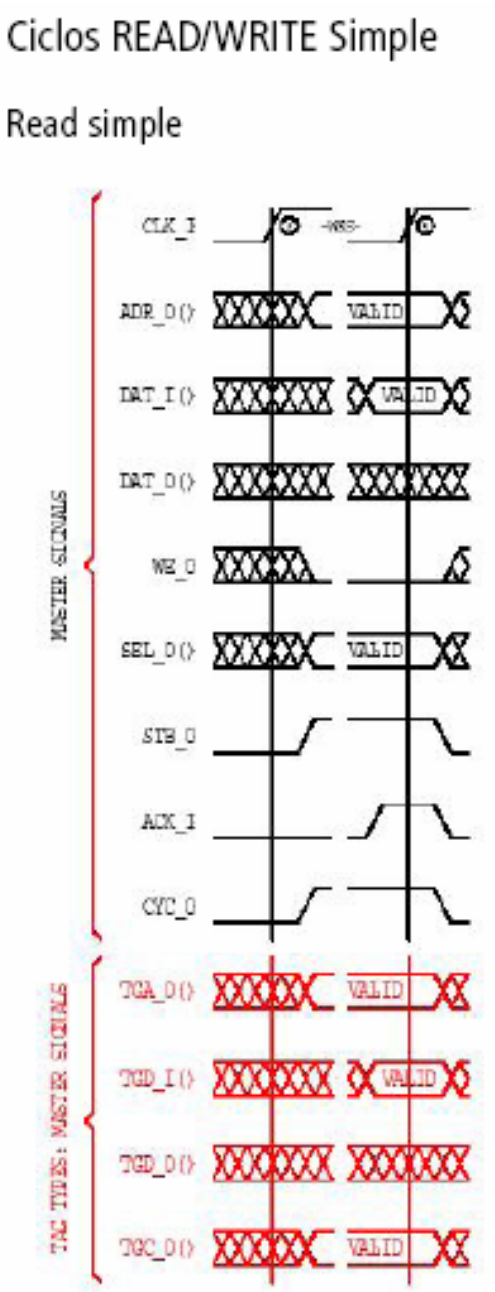
Descrição	TAG Type	Associado a
Address tag	TGA_I/O()	ADR_I/O()
Data tag, input	TGD_I()	DAT_I()
Data tag, output	TGD_O()	DAT_O()
Cycle tag	TGC_O()	Bus Cycle

Exemplo de definição do sinal PAR_O

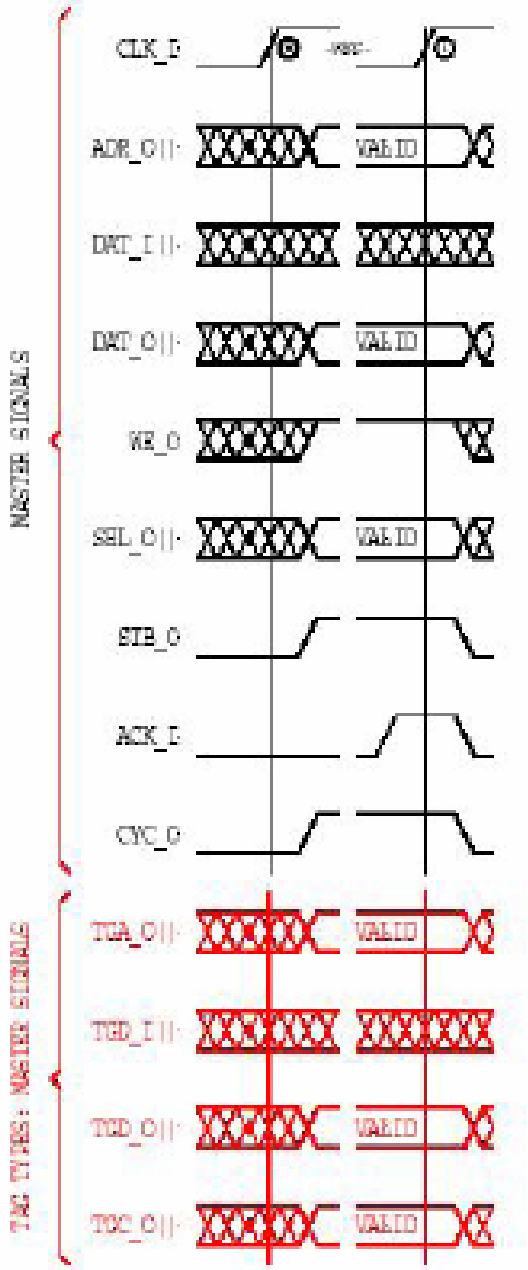
- NOME: PAR_O
- DESCRIÇÃO: Bit de paridade par
- MASTER TAG TYPE: TGD_O()

Ciclos READ/WRITE Simples

Read Simples



Write simple

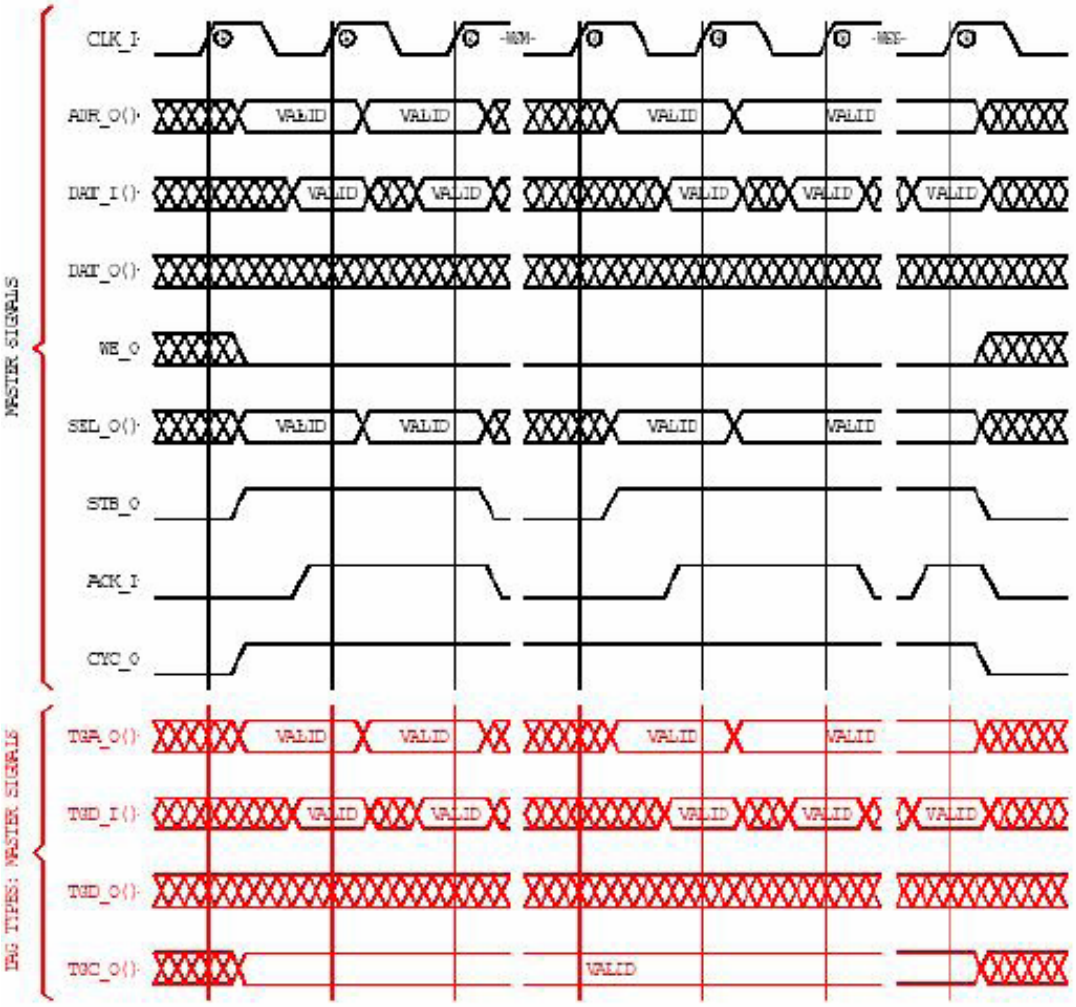


Ciclos READ/WRITE em bloco

Durante as transferências em bloco, a interface realiza basicamente transferências Read/Write simples.

Esse tipo de transferência é útil, sobretudo em sistemas com vários Másters.

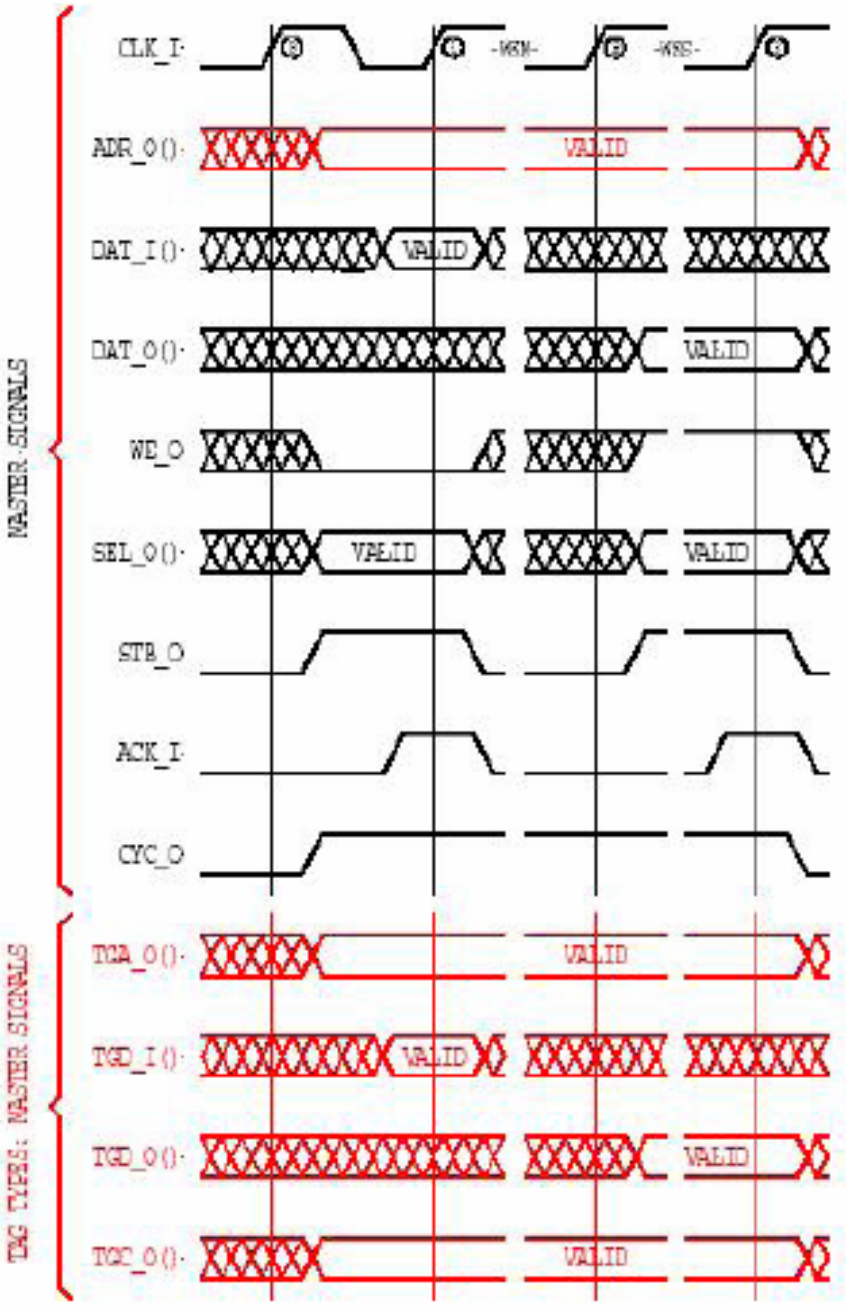
Read em bloco
Pode ser realizada uma transferência em cada ciclos de *clock*.



Write em bloco

Pode ser realizada uma transferência em cada ciclos de *clock*.

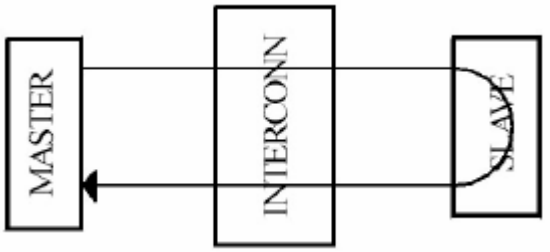
Ciclos RMW


MASTER SIGNALS
CLK_I
ADR_O()
DAT_I()
DAT_O()
WE_O
SEL_O()
STB_O
ACK_I
CYC_O
VALID
TAG TYPES: MASTER SIGNALS
TGA_O()
TGD_I()
TGD_O()
TGC_O()

4. WISHBONE com ciclos registrados

Para alcançar a máxima taxa de transferência, é necessário que os sinais de final de ciclo (ACK_I, ERR_I, RTY_I) sejam gerados de forma assíncrona. Por exemplo ACK_O como um AND de CYC_I e STB_I.

Mas, isto pode levar a atrasos muito grandes no chip, uma vez que um sinal passa por vários nós.



Uma solução para esse problema é registrar as terminações de ciclo.

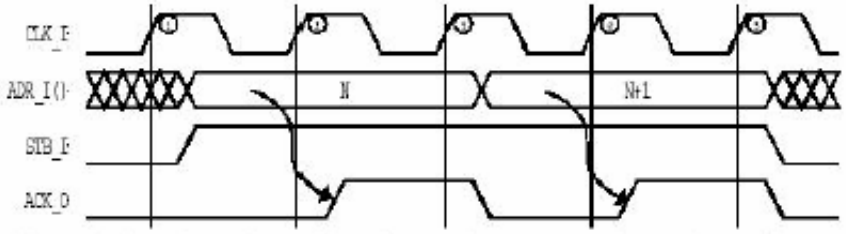


Figure 4-3 WISHBONE Classic synchronous cycle terminated burst

O problema dessa solução é que ela gera um WAIT STATE, reduzindo a metade à taxa de transferência.

Isto é solucionado com o uso de TAGs que identificam o tipo de ciclo que está sendo executado. Pode-se indicar se a transferência atual é a ultima o se vão seguir transferindo dados. Com isso é possível saber se deve-se desativar o sinal ACK_O ou mantê-lo ativo no próximo ciclo de *clock*.

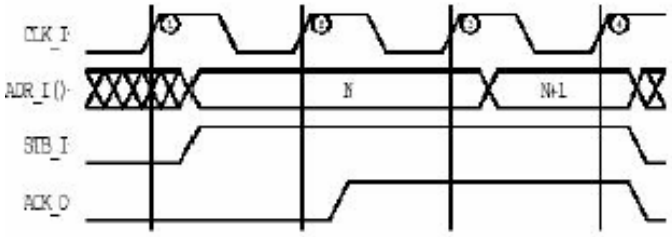


Figure 4-4 Advanced synchronous terminated burst

As interfaces que funcionam com esse tipo de função também suportam o Wishbone clássico.

4.1 TAGs utilizados

CTI_I()

- Cycle Type Identifier CTI_O() do tipo TGC_O() e CTI_I() do tipo TGC_I()
- O Mestre é quem envia a informação. O Escravo pode utilizá-la para preparar suas respostas.
- Para aproveitar ao máximo a largura da banda, tanto Mestre como Escravo devem suportar esse modo de trabalho.

CTI_O(2:0)	Descrição
000	Classic cycle
001	Constant address burst cycle
010	Incrementing burst cycle
011	Reserved
100	Reserved
101	Reserved
110	Reserved
111	End-of-Burst

As interfaces que suportam Ciclos Registrados devem suportar ao menos o modo clássico, para garantir compatibilidade.

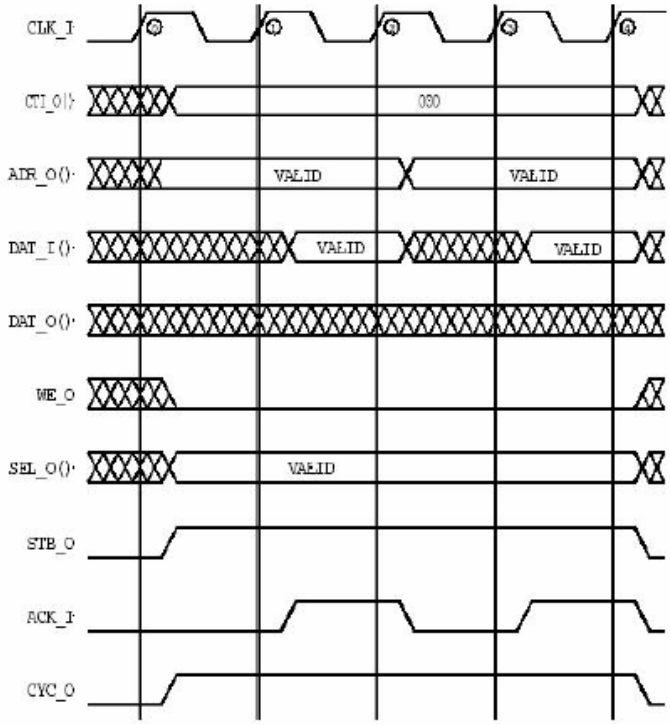
BTE_IO()

- Burst Type Extention BTE_O() e BTE_I do tipo TGA_N()
- Dão informações adicionais de como se incrementam os endereços.

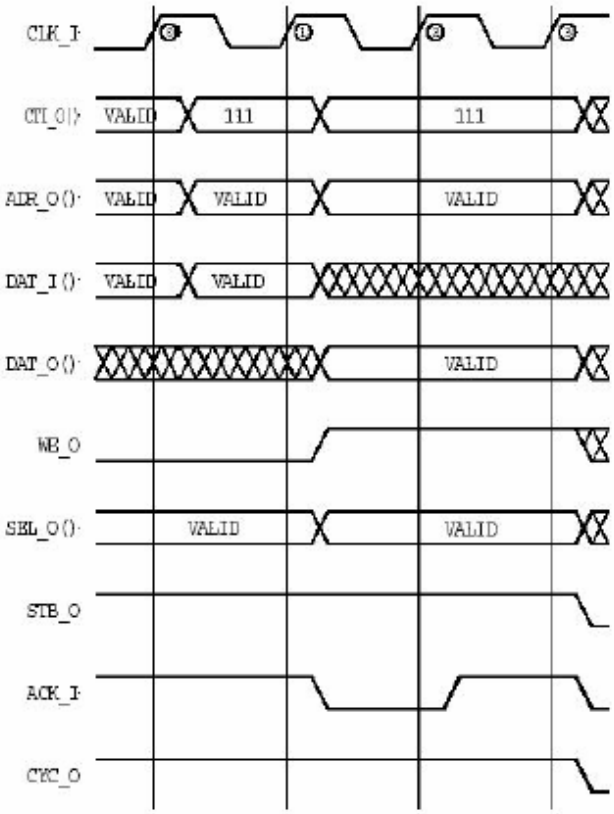
BTE_IO(1:0)	Descrição
00	Linear burst
01	4-beat wrap burst
10	8-beat wrap burst
11	16-beat wrap burst

As interfaces que suportam BURST incrementado devem suportar BTE_O() e BTE_I().

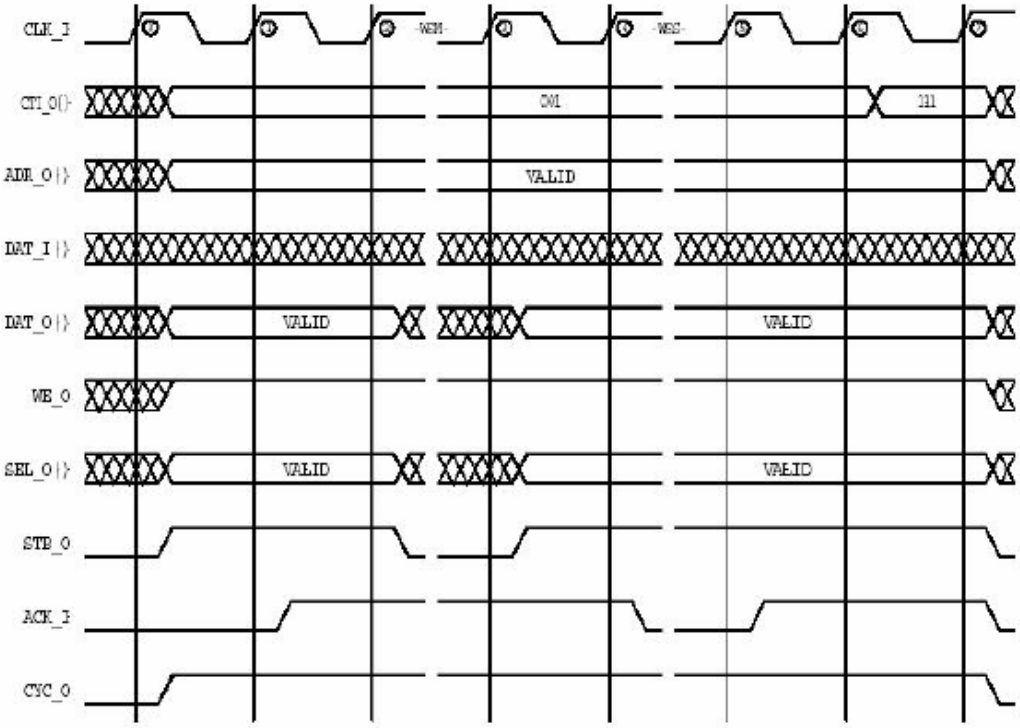
Classic Cycle



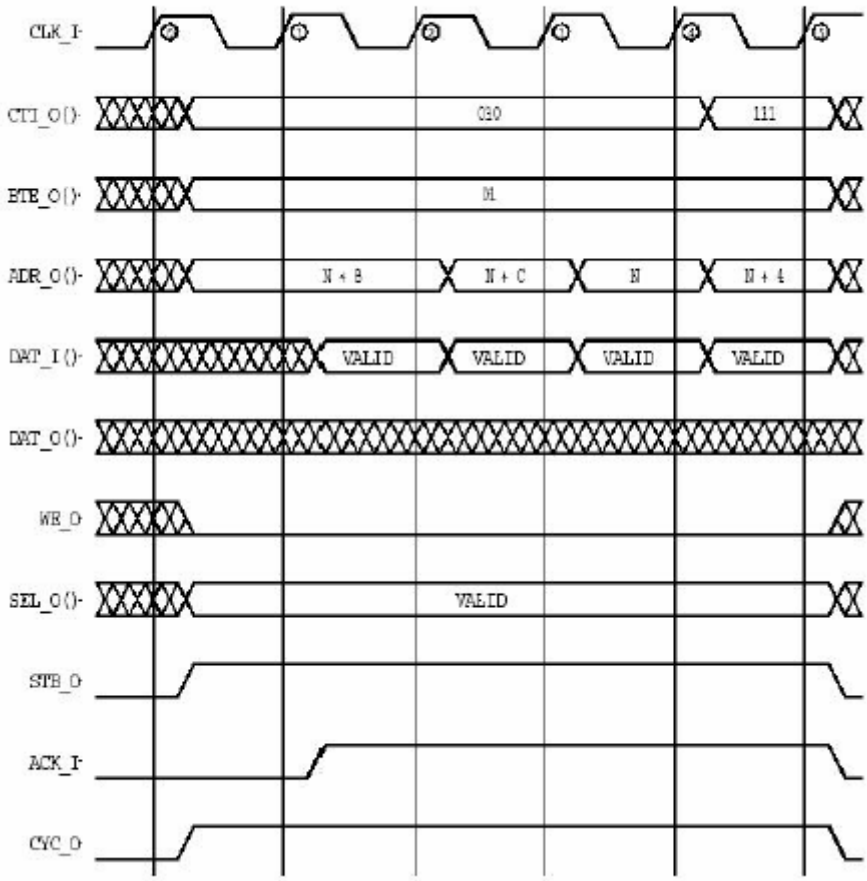
End Of Burst



Constant Address Cycle



Incremental Address Burst



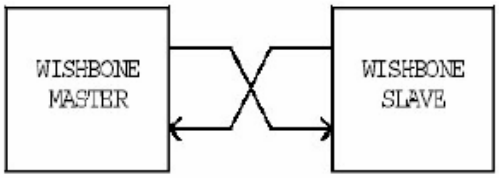
5. Conexões entre Módulos

Wishbone facilita o trabalho dos desenvolvedores, uma vez que padroniza a interface.

Está baseado em uma arquitetura Mestre/Escravo e os mesmo se comunicam através de uma interface usual chamada INTERCON.

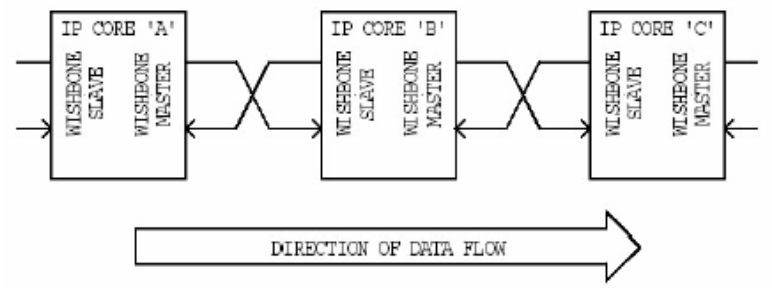
A INTERCON não é nada mágico, é também um arquivo escrito em linguagem de descrição de hardware.

Conexões ponto-a-ponto



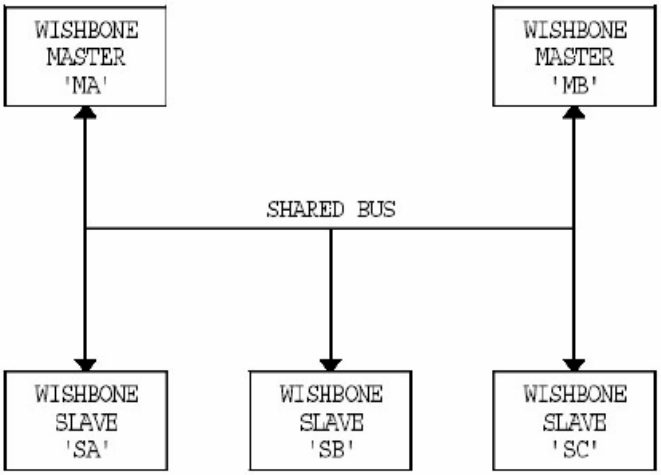
Fluxo de Dados

É uma arquitetura plausível para projetos que processem dados em pipeline.



Barramento Compartilhado

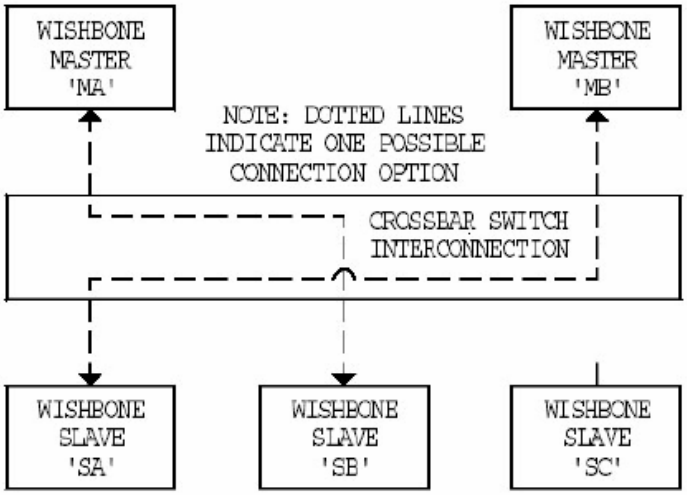
Todos os dispositivos estão conectados a um mesmo barramento e qualquer Mestre pode iniciar uma transação com qualquer Escravo.



Um árbitro determina quando há troca de turno entre cada um dos Mestres.

Switch de conexões

Consome mais recursos, mas permite vários pares de comunicação Mestre/Escravo de uma só vez.



INTERFACE PCI

1. Considerações Iniciais

O barramento PCI é atualmente padrão para a interconexão de dispositivos em um PC. Ele define uma interface para se conectar dispositivos que provém alguma função, por exemplo, som, captura de vídeo, acesso a redes, co-processadores matemáticos, etc.

Através do uso do barramento PCI cria-se uma placa portável a qualquer computador pessoal mediante unicamente ao desenvolvimento de um driver adequado a cada sistema operacional. Tornando assim o uso do barramento PCI bastante interessante para o desenvolvimento de qualquer aplicação, uma vez que a sua largura de banda para comunicação com o processador é maior do que as demais interfaces disponíveis atualmente (USB, *Ethernet*, etc).

2. Características do barramento PCI

2.1 Histórico e características principais

Em julho de 1992, a *Intel Corporation* apresentou a *Peripheral Component Interconnect*. Há muito esperada como uma especificação do barramento local, o anúncio inicial provou ser mais e menos do que a industria esperava. A primeira especificação PCI documentou completamente a concepção Intel sobre o que o barramento local deveria ser – mas não era um barramento local propriamente dito. Ao invés de apresentar um barramento pronto fechado, a Intel definiu regras de projeto mandatárias, incluindo linhas guias de Hardware para ajudar a certificar a operação apropriada de placas-mãe a altas velocidades com um mínimo de complexidade de projeto. Isso mostrava como ligar circuitos de PC – incluindo o barramento de expansão – para operações de alta velocidade. Mas o anúncio inicial do PCI falhou exatamente onde a indústria tinha maior interesse: os pinos de um conector de barramentos de expansão que permitisse o projeto de placas de expansão intercambiáveis. Na verdade o PCI surgiu não para ser um barramento local, mas um sistema de interconexão de alta velocidade que tiraria tal função do microprocessador e que trabalharia mais próximo da velocidade desse.

Com o passar dos anos novas versões da especificação foram surgindo sempre no intuito de melhorar o sistema. Em 1993, na especificação 2.0, o caminho de dados foi aumentado de 32 para 64 *bits* e foi dada uma completa descrição de conectores de expansão para ambas implementações de 32 e 64 *bits* num bus de

expansão PCI. Além disso, o PCI 2.0 foi desenhado para ser independente do microprocessador.

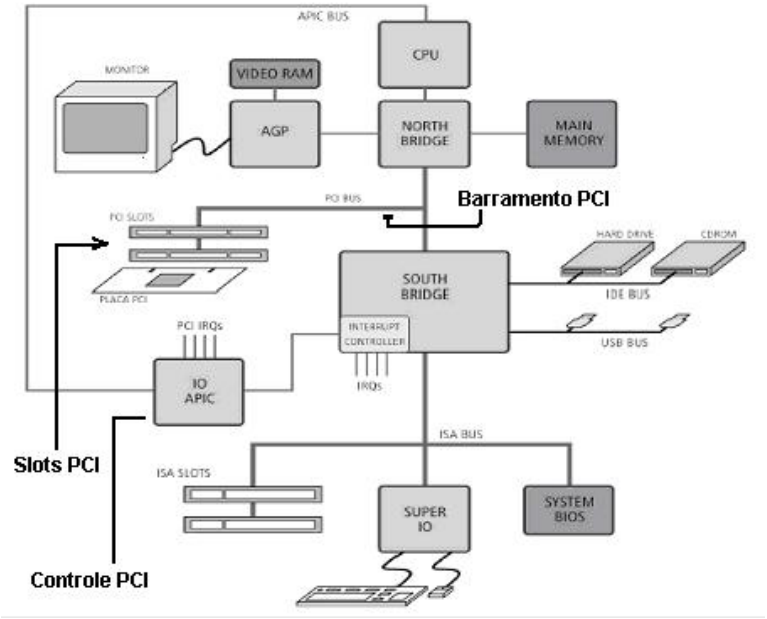
Outra inovação do barramento PCI é expandir a limitação de alguns carregamentos elétricos através de pontes PCI-PCI. Uma ponte deste tipo cria um barramento secundário isolado eletricamente do barramento raiz, permitindo-se a utilização de mais placas de expansão. Esta característica consagrou o PCI como barramento nas arquiteturas de servidores de grande desempenho.

Atualmente o barramento PCI está na especificação 3.0 (11). Essa especificação tem como principal evolução à migração completa do funcionamento do barramento PCI para 3.3V em vez dos originais 5V. Há ainda uma nova especificação chamada *PCI Express*, que vem com a proposta para o uso de um barramento PCI com velocidades de 133Mhz para novamente se equiparar com o barramento externo das CPUs. A tabela na figura a seguir mostra a evolução na capacidade de transferência de dados pelo barramento PCI.

Clock do sistema [MHz]	Largura do barramento [bits]	Taxa de transferência [MB/s]
33	32	132
33	64	264
66	64	528
133	64	1056

Evolução da banda de transferência do barramento PCI

A figura a seguir mostra a arquitetura básica de um sistema baseado no barramento PCI.



Arquitetura de um sistema baseado no barramento PCI

As principais características do barramento PCI segundo as ultimas especificações (não PCI Express) são:

- Independência da arquitetura do processador.
- Suporta um volumo de até 32 dispositivos físicos, para um total de 256 funções possíveis por barramento.
- Baixo consumo de energia.
- Uso de transferência de rajada(*burst*) para ambas as operações, de leitura e de escrita, com taxa de até 132MB/s em 32 *bits*.
- Velocidades de barramento de 33Mhz e 66Mhz
- Barramento de 32 ou 64*bits*.
- Tempo de acesso baixo (60ns a 33Mhz).
- Suporte a barramento Mestre que permite aos inicializadores acessos de igual para igual no barramento, bem como acesso à memória principal e expansão de mecanismos.
- Arbitragem oculta do barramento, que elimina a latência encontrada durante as arbitragens em outros barramentos.
- Baixo número de pinos, sendo necessários apenas 47 para a implementação de alvos PCI e 49 pinos para os inicializadores.
- Checagem da paridade de endereçamentos, comandos e dados.
- 3 espaços de endereçamento, sendo toda a definição de memória, I/O e configuração de espaço de endereços.
- Auto Configuração, através da completa especificação de *BIT-LEVEL* dos registradores necessários para o suporte, detecção e configuração de periféricos automaticamente.
- Transparência de software, já que os *drivers* utilizam os mesmos comandos e definições de *STATUS* quando se comunicam com todos os dispositivos PCI.

3 Tecnologias Existentes

3.1 Plataformas de desenvolvimento para o barramento PCI

É possível se comprar plataformas completas de desenvolvimento para o barramento PCI, em pacotes que incluem lógica reconfigurável, um core PCI e um *driver* de exemplo. Tais pacotes, no entanto, fazem uso de FPGAs de última geração, cores PCI e programas de desenvolvimento de *drivers* comerciais. Tudo isso torna o seu preço proibitivo para o uso em universidades nos dias de hoje. Não se encontram plataformas com dispositivos intermediários a um custo acessível.

A seguir alguns exemplos de preços e características de algumas plataformas disponíveis no mercado:

- APEX PCI Development Kit APEX 20KE (ALTERA)
  - Custo: US$3.495

- PCI Development Kit, Stratix Edition (ALTERA)
  - Custo: US$ 1.995

- DS-KIT-PCI-200
  - Custo US$1.995

3.2 Placas PCI

Uma alternativa a compra de uma plataforma de desenvolvimento completo seria a compra de uma placa PCI com lógica reconfigurável e desenvolver os *drivers* e o *core PCI*.

Também como no caso das plataformas, é difícil de se encontrar placas com preço acessível. A seguir alguns exemplos de placas disponíveis no mercado:

- GR-PCI-XC2V LEON
  - Custo $3.450 Euros

- ADS-XLX-V2-DEV1500
  - Custo US$ 1.000

- DS-KIT-2S200
  - Custo US$250

Pode-se notar que todos os kits possuem preços proibitivos a não ser o ultimo. Produzido pela *insight electronics*, possui um preço razoável. Mas ainda assim, seria necessário o desenvolvimento do *CORE PCI* e dos *drivers* para as aplicações desejadas.

4.3.3 Cores PCIs gratuitos

Atualmente existe um *CORE PCI* de livre distribuição no *site* www.opencores.org, que foi testado em uma placa com uma FPGA da XILINX. Esse *CORE* é também compatível com a interface *WISHBONE*.

Não foi encontrado nenhum outro *CORE PCI* com código aberto e gratuito que fosse compatível com a interface *WISHBONE* disponível para descarga na *Internet*.

4.3.4 Drivers PCI

A forma correta de se utilizar às funções fornecidas por uma placa conectada ao barramento PCI depende totalmente da aplicação. Então o *driver* deve ser

desenvolvido na medida em que a aplicação é desenvolvida, visando à aplicação final. Devido a isto, os *drivers* existentes servem somente de exemplo para o desenvolvimento de um *driver* específico para a nossa aplicação.

Existem ferramentas para o desenvolvimento de *drivers* para PCI para os mais diversos sistemas operacionais. Dentre elas destaca-se o *WinDriver*, fabricado pela Jungo (www.jungo.com/windriver).

Mas, o uso de tais ferramentas novamente seria proibitivo pelo alto custo em se adquirir os *softwares*. Uma licensa do windriver custa US$2.000.

Por outro lado, a versão 2.4 do sistema operacional Linux tornou o desenvolvimento de *drivers* sensivelmente mais simples do que nas versões anteriores, com ampla e detalhada documentação sobre o desenvolvimento de módulos de kernel. Tudo isso somado com a disponibilidade dos códigos fonte de todos os *drivers* existentes que já foram desenvolvidos, tudo isso com a livre licença do Linux.

Conseguiu-se acesso para uma possível implementação do projeto a plataforma PCI de distribuição totalmente gratuita desenvolvida na Universidad de la Republica (Montevidéu, Uruguai) do Uruguai, que é descrita a seguir.

4. Descrição da plataforma de desenvolvimento da Universidad de la Republica

O desenvolvimento de um dispositivo PCI seja para uso comercial ou para uso acadêmico, seria muito mais simples através do uso de uma já testada e desenvolvida placa de propósito geral que se adapte facilmente de acordo com as necessidades do projeto.

Como a placa PCI deve ser facilmente adaptável à aplicação designada, os componentes eletrônicos utilizados devem ser facilmente modificáveis. Hoje em dia isso é possível graças ao uso de dispositivos de lógica programável.

O design programando na FPGA deve implementar a nova função que está sendo adicionada ao PC e deve também ser capaz de se comunicar com o PC através do barramento PCI.

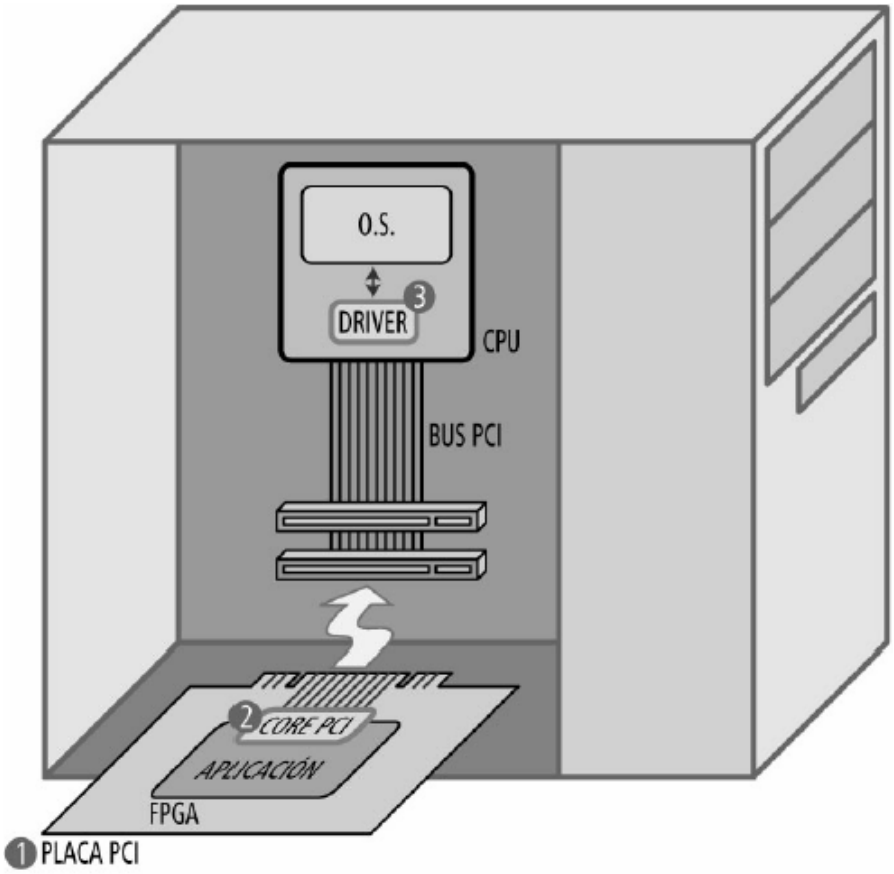
Dada a complexidade do barramento PCI, o design pode ser dividido em duas partes, uma responsável pela comunicação através do barramento PCI e outra que implementa a nova função. O módulo de comunicação é normalmente chamado de *PCI core*, ele simplifica o uso do barramento PCI “escondendo” ou “mascarando” os detalhes específicos de funcionamento do barramento, deixando assim seu uso transparente ao desenvolvimento da aplicação.

O resultado final ficou como sendo uma plataforma de desenvolvimento de hardware para funções utilizando o barramento PCI, e foi desenvolvida no Instituto de Engenharia Elétrica da Faculdade de Engenharia da Universidad de la Republica (Montevidéu, Uruguai) do Uruguai.

A plataforma contém:

1. Uma placa de propósito geral baseada em lógica programável, que pode ser conectada ao barramento PCI.
2. Um módulo core PCI, que será utilizado pela aplicação configurada conjuntamente no FPGA, que mascara o funcionamento complexo do barramento PCI ao usuário.

PLATAFORMA DE DESENVOLVIMENTO PCI.



Plataforma PCI

- Interface *Wishbone*

É necessário um conhecimento básico da interface *Wishbone* para se usar o *core PCITWBM*, que é o core desenvolvido para fazer a interface com o barramento PCI, mas não é necessário o conhecimento detalhado do barramento PCI.

- Considerações Finais

O uso do padrão PCI para o desenvolvimento de projetos tanto acadêmicos quanto comerciais era algo pesado devido ao alto custo das plataformas de desenvolvimento disponíveis no mercado. A falta de uma plataforma, mesmo que acadêmica, gratuita fazia com que muitos projetos partissem para outras formas de interface, devido à complexidade do barramento PCI, sendo que sua utilização, sem o auxilio de uma plataforma de desenvolvimento adequada não é algo trivial.

Com o advento da plataforma desenvolvida na Universidad de la Republica (Montevidéu, Uruguai) o desenvolvimento de aplicações PCI fica imensamente facilitado. Com a disponibilidade do core gratuitamente e de exemplos de aplicação, o desenvolvimento de novas aplicações ganha um grande estímulo. Vantagem ainda maior é a compatibilidade com o padrão *Wishbone*, que vêm ganhando força na comunidade de desenvolvimento de cores acadêmicos e gratuitos na *Internet* (50).

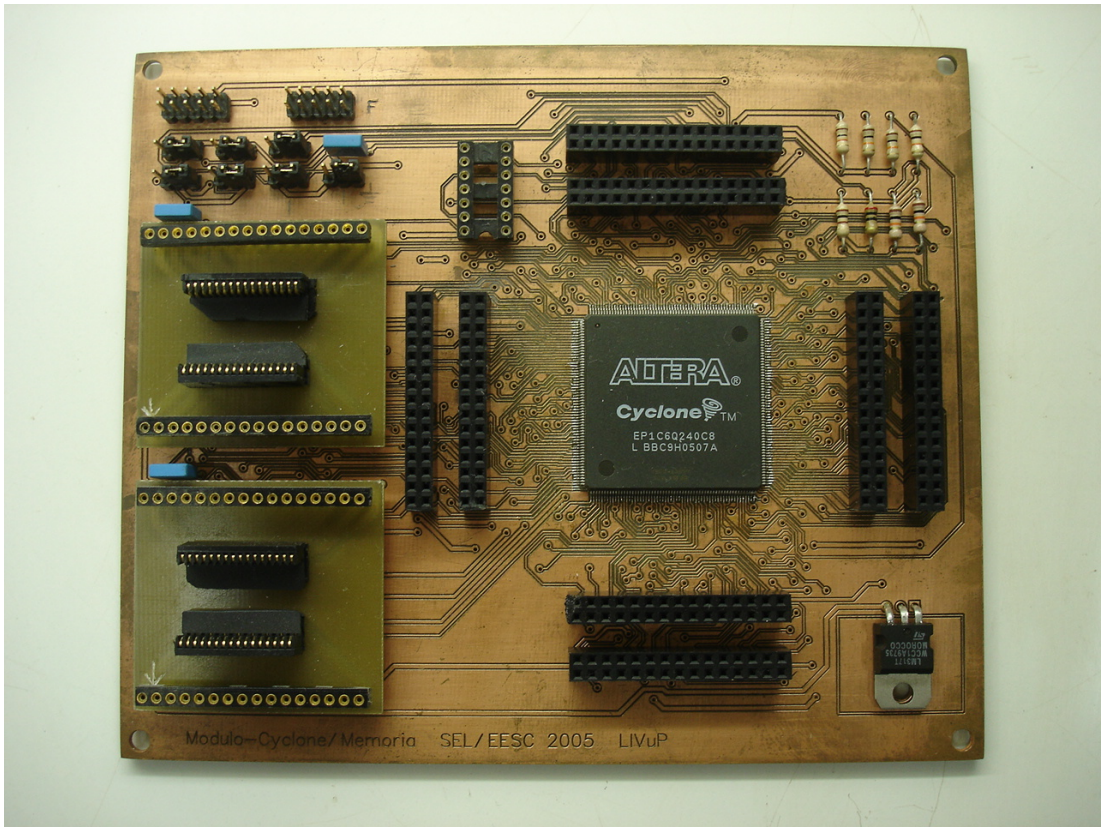
O objetivo do trabalho de mestrado ser compatível com a interface *Wishbone* é explorar exatamente essa potencialidade do barramento PCI e da disponibilidade de um core PCI gratuito, funcional e portável através do padrão *Wishbone*, características essas todas inerentes ao *core PCITWBM*. Através do uso do mesmo, pode-se esperar um ótimo desempenho no que diz respeito à transferência de dados para o computador pela arquitetura a ser desenvolvida.

ANEXO B

ANEXO C : Função Matlab para cálculo dos momentos invariantes

```
function [invmoments] = inmoments(im)
    [rows,cols] = size(im);
    x = ones(rows,1)*[1:cols];
    y = [1:rows]'*ones(1,cols);

    %Area e perímetro
    area = sum(sum(im));
    meanx = sum(sum(double(im).*x))/area;
    meany = sum(sum(double(im).*y))/area;


    % Momentos Centrais.
    n20 = sum(sum(double(im).*((x - meanx).^2)))/(area.^2);
    n02 = sum(sum(double(im).*((y - meany).^2)))/(area.^2);
    n11 = sum(sum(double(im).*(x - meanx).*(y - meany)))/(area.^2);
    n30 = sum(sum(double(im).*((x - meanx).^2)))/(area.^(5/2));
    n12 = sum(sum(double(im).*(x - meanx).*((y -
meany).^2)))/(area.^(5/2));
    n21 = sum(sum(double(im).*((x - meanx).^2).*(y -
meany)))/(area.^(5/2));
    n03 = sum(sum(double(im).*((y - meany).^2)))/(area.^(5/2));

    % Momentos Invariantes
    phi1 = n20 + n02
    pause
    phi2 = ((n20 - n02).^2) + 4.*(n11.^2)
    pause
    phi3 = ((n30 - 3.*n12).^2) + ((3.*n21 - n03).^2)
    pause
    phi4 = ((n30 + n12).^2) + (n21 + n03).^2
    pause
    phi5 = (n30 - 3.*n12).*(n30 + n12).*((n30 + n12).^2 - 3.*(n21 +
n03).^2) + (3.*n21 - n03).*(n21 + n03).*(3.*(n30 + n12).^2 - (n21 +
n03).^2)
    pause
    phi6 = (n20 - n02).*((n30 + n12).^2 - (n21 + n03).^2) +
4.*n11.*(n30 + n12) + (n21 + n03)
    pause
    phi7 = (3.*n21 - n30).*(n30 + n12).*((n30 + n12).^2 - 3.*(n21 +
n03).^2) + (3.*n12 - n30).*(n21 + n03).*(3.*(n30 + n12).^2 - (n21 +
n03).^2)
    pause


        invmoments = [abs(log(phi1 + eps)), abs(log(phi2 + eps)),
abs(log(phi3 + eps)), abs(log(phi4 + eps)), abs(log(phi5 + eps)),
abs(log(phi6 + eps)), abs(log(phi7 + eps))]
    end
```

ANEXO D: Foto do sistema Desenvolvido



ANEXO E: Módulo de cálculo da área e médias

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.fixed_pkg.all;
use WORK.float_pkg.all;
use WORK.float_alg_pkg.all;
use WORK.momentos.all;
use WORK.wb_interface.all;
use WORK.intermed.all;

entity calcumomentos is
   port (
         oddev: in STD_LOGIC;
                  clkcirc: in STD_LOGIC;
                  rw: in STD_LOGIC;
                  pos: in STD_LOGIC_VECTOR (16 downto 0);
                  memin: in STD_LOGIC_VECTOR (31 downto 0);
                  tstout1: out STD_LOGIC_VECTOR (31 downto 0)
            --momento2: out STD_LOGIC_VECTOR (63 downto 0)
                  --tstout2: out STD_LOGIC_VECTOR (127 downto 0)

   );
end calcumomentos;

architecture calcu_arch of calcumomentos is
--signal k,l,m,n,o,p,q : float64;
--signal a,b,c,d,e,f,g : float64;
signal tp1, tp2, tp3, tp4: float32;
signal area,meanx,meany: float32;
signal don1, don2: boolean := FALSE;
--type img is array (1 to 512, 1 to 512) of integer;
--signal im1: img;

procedure are (signal ck: in STD_LOGIC;
                              signal mein: in STD_LOGIC_VECTOR (31 downto
0);
                              signal done: out boolean;
                              signal ar, mtpx, mtpy: inout float32) is

variable contpix, contlin: integer := 0;
--variable art: float32;
--variable mtpxt: float32;
```

```
--variable mtpyt: float32;
begin

        if (ck = '1') then
        if (contlin <= 512) then
        contpix := contpix + 1;
        ar <= ar + to_float(mein,ar);
        mtpx <= mtpx + to_float(mein,ar) * contpix;
        mtpy <= mtpy + to_float(mein,ar) * contlin;
        else
        contlin := 0;
        done <= TRUE;
        return;
        end if;
        if (contpix = 512) then
        contpix := 0;
        contlin := contlin + 1;
        return;
        end if;
        end if;

end are;

procedure tmps (signal ck: in STD_LOGIC;
                                signal mein: in STD_LOGIC_VECTOR (31 downto
0);
                                signal mx, my: in float32;
                                signal done: out boolean;
                                signal tp1, tp2, tp3, tp4: inout float32) is

variable contpix, contlin: integer := 0;
begin

        if (ck = '1') then
        if (contlin <= 512) then
        contpix := contpix + 1;
        tp1 <= tp1 + (to_float(mein,tp1) * ((contpix - mx)*(contpix - mx)));
        tp2 <= tp2 + (to_float(mein,tp2) * ((contlin - my)*(contlin - my)));
        tp3 <= tp3 + (to_float(mein,tp3) * ((contpix - mx)*(contlin - my)));
        tp4 <= tp4 + (to_float(mein,tp4) * ((contpix - mx)*(contpix - mx)*(contlin
- my)));

        else
        contlin := 0;
        done <= TRUE;
        return;
        end if;
        if (contpix = 512) then
```

```
        contpix := 0;
        contlin := contlin + 1;
        return;
        end if;
        end if;

end tmps;

begin
--signal                tstout3: STD_LOGIC_VECTOR (63 downto 0);
--signal                tstout4: STD_LOGIC_VECTOR (63 downto 0);
--signal                tstout5: STD_LOGIC_VECTOR (63 downto 0);
--signal                tstout6: STD_LOGIC_VECTOR (63 downto 0);
--signal                tstout7: STD_LOGIC_VECTOR (63 downto 0);

--process (rw, clkcirc, don1)

begin

if (pos = "00000000000000000") then
while (not don1) loop
are (clkcirc, memin, don1, area, meanx, meany);
end loop;
while (not don2) loop
tmps (clkcirc, memin, meanx, meany, don2, tp1, tp2, tp3, tp4);
end loop;
end if;

--end process;

--tmp1 <= to_float (1460200000000,tmp1);
--tmp2 <= to_float (1456800000000,tmp2);
--ar <= to_float (66245000,ar);


--      k <= to_float (3.3196,k);
--      l <= to_float (0.00040785,l);
--      m <= to_float (0.00034948,m);
--      n <= to_float (0.000036129,n);
--      o <= to_float (3.3273,o);
--      p <= to_float (0.000097694,p);
--      q <= to_float (0.0004088,q);
        --tstout2 <= to_slv(z);
        --x <= z + y;
--k <= "n02"(ar,tmp1);
--o <= "n20"(ar,tmp2);

--      a <= "primeiro"(o,k);
```

```
--      b <= "segundo"(o,k,m);
--      c <= "terceiro"(q,n,p,l);
--      d <= "quarto"(q,n,p,l);
--      e <= "quinto"(q,n,p,l);
--      f <= "sexto"(o,k,q,n,p,l,m);
--      g <= "setimo"(p,q,n,l);

--      tstout1 <= to_slv(a);
--      tstout2 <= to_slv(d);
--      tstout3 <= to_slv(c);
--      tstout4 <= to_slv(d);
--      tstout5 <= to_slv(e);
--      tstout6 <= to_slv(f);
--      tstout7 <= to_slv(g);

end calcu_arch;
```

ANEXO F: Módulo de cálculo do valores intermediários

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.fixed_pkg.all;
use WORK.float_pkg.all;
use WORK.float_alg_pkg.all;

package intermed is

	function "n20" (area, tp1: float32) return float32;
	function "n02" (area, tp2: float32) return float32;
	function "n11" (area, tp3: float32) return float32;
	function "n30" (area, tp1: float32) return float32;
	function "n12" (area, tp3: float32) return float32;
	function "n21" (area, tp4: float32) return float32;
	function "n03" (area, tp2: float32) return float32;
end package intermed;


package body intermed is

function "n20" (area, tp1: float32) return float32 is
variable m: float32;
	begin
	m := tp1/(area * area);
	return m;
	end;

function "n02" (area, tp2: float32) return float32 is
variable m: float32;
	begin
	m := tp2/(area * area);
	return m;
	end;

function "n11" (area, tp3: float32) return float32 is
variable m: float32;
	begin
	m := tp3/(sqrt(area * area));
	return m;
	end;

function "n30" (area, tp1: float32) return float32 is
```

```
variable m: float32;
        begin
        m := tp1/(sqrt(area * area * area * area * area));
        return m;
        end;

function "n12" (area, tp3: float32) return float32 is
variable m: float32;
        begin
        m := tp3/(sqrt(area * area * area * area * area));
        return m;
        end;

function "n21" (area, tp4: float32) return float32 is
variable m: float32;
        begin
        m := tp4/(sqrt(area * area * area * area * area));
        return m;
        end;

function "n03" (area, tp2: float32) return float32 is
variable m: float32;
        begin
        m := tp2/(sqrt(area * area * area * area * area));
        return m;
        end;

end package body intermed;
```

ANEXO G: Módulo de cálculo dos momentos finais

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.fixed_pkg.all;
use WORK.float_pkg.all;

package momentos is
	function "primeiro" (n20, n02: float32) return float32;
	function "segundo" (n20, n02, n11: float32) return float32;
	function "terceiro" (n30, n12, n21, n03: float32) return float32;
	function "quarto" (n30, n12, n21, n03: float32) return float32;
	function "quinto" (n30, n12, n21, n03: float32) return float32;
	function "sexto" (n20, n02, n30, n12, n21, n03, n11: float32) return
float32;
	function "setimo" (n21, n30, n12, n03: float32) return float32;
end package momentos;


package body momentos is

function "primeiro" (n20, n02: float32) return float32 is
variable phi1: float32;
	begin
	phi1 := n20 + n02;
	return phi1;
	end;

function "segundo" (n20, n02, n11: float32) return float32 is
variable phi2: float32;
	begin
	phi2 := (n20 - n02)*(n20 - n02) + 4*((n11)*(n11));
	return phi2;
	end;

function "terceiro" (n30, n12, n21, n03: float32) return float32 is
variable phi3: float32;
	begin
	phi3 := ((n30 - 3*n12)*(n30 - 3*n12)) + ((n21 + n03)*(n21 + n03));
	return phi3;
	end;

function "quarto" (n30, n12, n21, n03: float32) return float32 is
variable phi4: float32;
```

```
        begin
        phi4 := ((n30 + n12)*(n30 + n12)) + ((n21 + n03)*(n21 + n03));
        return phi4;
        end;

function "quinto" (n30, n12, n21, n03: float32) return float32 is
variable phi5: float32;
        begin
        phi5 := (n30 - 3*n12)*(n30 + n12)*(((n30 + n12)*(n30 + n12)) - 3*((n21 +
n03)*(n21 + n03))) + (3*n21 - n03)*(n21 + n03)*(3*((n30 + n12)*(n30 + n12)) -
((n21 + n03)*(n21 + n03)));
        return phi5;
        end;

function "sexto" (n20, n02, n30, n12, n21, n03, n11: float32) return float32 is
variable phi6: float32;
        begin
        phi6 := (n20 - n02)*((n30 + n12)*(n30 + n12) - (n21 + n03)*(n21 + n03)) +
4*n11*(n30 + n12) + (n21 + n03);
        return phi6;
        end;

function "setimo" (n21, n30, n12, n03: float32) return float32 is
variable phi7: float32;
        begin
        phi7 := (3*n21 - n30)*(n30 + n12)*((n30 + n12)*(n30 + n12) - 3*((n21 +
n03)*(n21 + n03))) + (3*n12 - n30)*(n21 + n03)*(3*((n30 + n12)*(n30 + n12)) -
((n21 + n03)*(n21 + n03)));
        return phi7;
        end;

end package body momentos;
```

ANEXO H: Módulo memória wishbone

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY wbram IS
	PORT
	(
		data		: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
		wren		: IN STD_LOGIC  := '1';
	wraddress		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
	rdaddress		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
	);
END wbram;


ARCHITECTURE SYN OF wbram IS

	SIGNAL sub_wire0 : STD_LOGIC_VECTOR (63 DOWNTO 0);

	COMPONENT altsyncram
	GENERIC (
		intended_device_family		: STRING;
		ram_block_type		: STRING;
		operation_mode		: STRING;
		width_a		: NATURAL;
		widthad_a		: NATURAL;
		numwords_a		: NATURAL;
		width_b		: NATURAL;
		widthad_b		: NATURAL;
		numwords_b		: NATURAL;
		lpm_type		: STRING;
		width_byteena_a		: NATURAL;
		outdata_reg_b		: STRING;
		indata_aclr_a		: STRING;
		wrcontrol_aclr_a		: STRING;
		address_aclr_a		: STRING;
		address_reg_b		: STRING;
		address_aclr_b		: STRING;
		outdata_aclr_b		: STRING;
		read_during_write_mode_mixed_ports		: STRING;
		power_up_uninitialized		: STRING
	);
```

```
        PORT (
                        wren_a          : IN STD_LOGIC ;
                        clock0 : IN STD_LOGIC ;
                address_a       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        q_b     : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
                        data_a: IN STD_LOGIC_VECTOR (63 DOWNTO 0)
        );
        END COMPONENT;

BEGIN
        q    <= sub_wire0(63 DOWNTO 0);

        altsyncram_component : altsyncram
        GENERIC MAP (
                intended_device_family => "Cyclone",
                ram_block_type => "M4K",
                operation_mode => "DUAL_PORT",
                width_a => 64,
                widthad_a => 8,
                numwords_a => 256,
                width_b => 64,
                widthad_b => 8,
                numwords_b => 256,
                lpm_type => "altsyncram",
                width_byteena_a => 1,
                outdata_reg_b => "CLOCK0",
                indata_aclr_a => "NONE",
                wrcontrol_aclr_a => "NONE",
                address_aclr_a => "NONE",
                address_reg_b => "CLOCK0",
                address_aclr_b => "NONE",
                outdata_aclr_b => "NONE",
                read_during_write_mode_mixed_ports => "DONT_CARE",
                power_up_uninitialized => "FALSE"
        )
        PORT MAP (
                wren_a => wren,
                clock0 => clock,
                address_a => wraddress,
                address_b => rdaddress,
                data_a => data,
                q_b => sub_wire0
        );



END SYN;
```

ANEXO I: Módulo interface wishbone

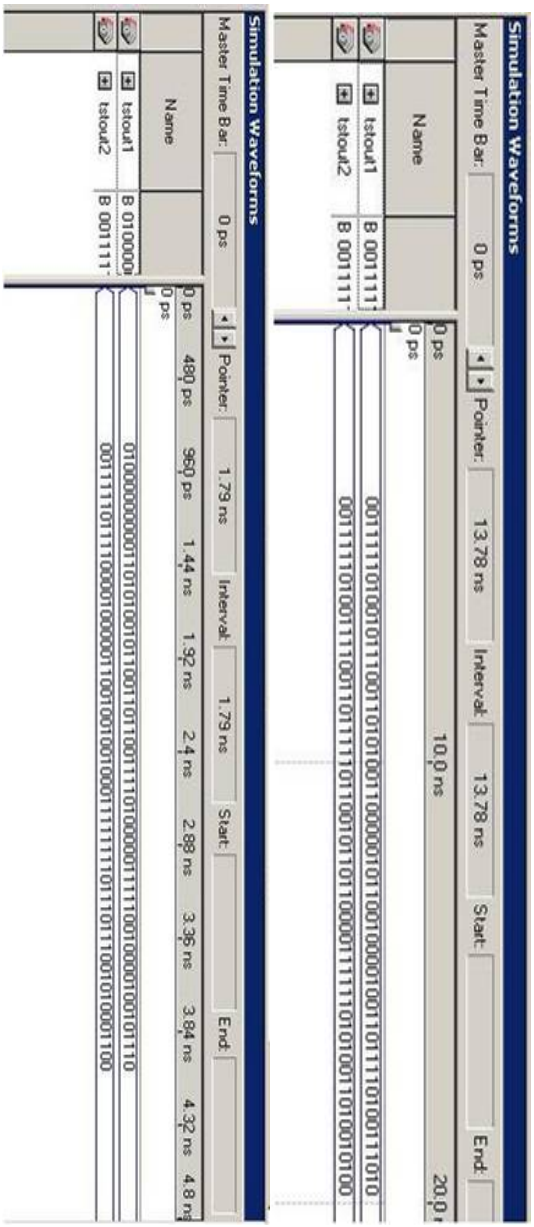
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-----------------------------------------------------------------
---- package para instanciar
-----------------------------------------------------------------
package wb_interface is
  COMPONENT interface_wb IS
  GENERIC (RAM_WIDTH : integer := 64;
          RAM_ADDRESS_WIDTH : integer := 7);
   PORT(
     --WB signals
     RST_I    : IN  STD_LOGIC;
     CLK_I    : IN  STD_LOGIC;
     DAT_I    : IN  STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
     DAT_O    : OUT STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
     ACK_O    : OUT STD_LOGIC;
     ADR_I    : IN  STD_LOGIC_VECTOR(RAM_ADDRESS_WIDTH-1
downto 0);
     CYC_I    : IN  STD_LOGIC;
     STB_I    : IN  STD_LOGIC;
     WE_I     : IN  STD_LOGIC;
     CTI_I    : IN STD_LOGIC_VECTOR(2 downto 0) -- Cycle type identifier
     );
   END component;
END package;


-----------------------------------------------------------------
---- componente
-----------------------------------------------------------------
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY interface_wb IS
   GENERIC (RAM_WIDTH : integer := 64;
          RAM_ADDRESS_WIDTH : integer := 7);
          PORT(
            ---WB signals
                  RST_I          : IN     STD_LOGIC;
                  CLK_I                   : IN     STD_LOGIC;
          DAT_I    : IN STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
```

```
DAT_O    : OUT      STD_LOGIC_VECTOR(RAM_WIDTH-1 downto 0);
                ACK_O                  : OUT STD_LOGIC;
                ADR_I    : IN
        STD_LOGIC_VECTOR(RAM_ADDRESS_WIDTH downto 0);
                CYC_I                  : IN    STD_LOGIC;
                STB_I          : IN    STD_LOGIC;
                WE_I           : IN    STD_LOGIC;
                CTI_I    : IN  STD_LOGIC_VECTOR(2 downto 0) -- Cycle type
identifier
                );
END interface_wb;


ARCHITECTURE wb_momentos OF interface_wb IS

signal ADR_COUNTER : STD_LOGIC_VECTOR(RAM_ADDRESS_WIDTH
downto 0);
signal next_ACK_O : STD_LOGIC;

type state_type IS
      (wb_ram_idle, wb_ram_wr, wb_ram_rd);

signal wb_ram_state, wb_ram_nxstate : state_type;

attribute syn_encoding : string;
attribute syn_encoding of wb_ram_state : signal IS "onehot";

CONSTANT LPM_RAM_WIDTHAD : INTEGER :=
(RAM_ADDRESS_WIDTH-2);

COMPONENT wbram
                        PORT (
                data            : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
                wren            : IN STD_LOGIC  := '1';
        wraddress               : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        rdaddress               : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                clock           : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
        );
        END COMPONENT;

BEGIN

ram0 : wbram

                PORT MAP (
                data     => DAT_I,
                wren     => (STB_I and CYC_I and WE_I),
```

```
            wraddress      => ADR_I(RAM_ADDRESS_WIDTH downto 0),
            rdaddress      => ADR_COUNTER,
            clock    => CLK_I,
            q        => DAT_O
      );

--**********************************************
wb_ram_state_machine_generator:
--**********************************************
PROCESS (CLK_I,RST_I,wb_ram_nxstate)
begin
  IF (CLK_I'event and CLK_I = '1') THEN
   IF RST_I = '1' THEN
     wb_ram_state    <= wb_ram_idle;
   ELSE
     wb_ram_state    <= wb_ram_nxstate;
   END IF;
       END IF;
END PROCESS;


--**********************************************
wb_ram_next_state_signals_generator:
--**********************************************
PROCESS (CLK_I,RST_I,wb_ram_nxstate)
begin
   CASE wb_ram_state IS
     WHEN wb_ram_idle =>
       if (CYC_I='1' and STB_I='1' and WE_I='1') then
         wb_ram_nxstate <= wb_ram_wr;
         next_ACK_O <= '1';
       elsif (CYC_I='1' and STB_I='1' and WE_I='0') then
         wb_ram_nxstate <= wb_ram_rd;
         next_ACK_O <= '1';
       else
         wb_ram_nxstate <= wb_ram_idle;
         next_ACK_O <= '0';
       end if;


     WHEN wb_ram_wr =>
       if (CYC_I='0' or (CYC_I='1' and CTI_I="111")) then
         wb_ram_nxstate <= wb_ram_idle;
         next_ACK_O <= '0';
       else
         wb_ram_nxstate <= wb_ram_wr;
         next_ACK_O <= '1';
       end if;
```

```
        WHEN wb_ram_rd =>
          if (CYC_I='0' or (CYC_I='1' and CTI_I="111")) then
            wb_ram_nxstate <= wb_ram_idle;
            next_ACK_O <= '0';
          else
            wb_ram_nxstate <= wb_ram_rd;
            next_ACK_O <= '1';
          end if;


      END CASE;
    END PROCESS;

  --**********************************************
  ACK_O_gen:
  --**********************************************
  PROCESS (CLK_I,RST_I,wb_ram_nxstate)
  begin
    IF (CLK_I'event and CLK_I = '1') THEN
     IF RST_I = '1' THEN
       ACK_O   <= '1';
     ELSE
       ACK_O   <= next_ACK_O;
     END IF;
          END IF;
  END PROCESS;


ADR_COUNTER_generation: process (CLK_I, RST_I)
  begin
  if CLK_I'event and CLK_I='1' then
    if RST_I='1' then
      ADR_COUNTER <= (OTHERS => '0');
    elsif (wb_ram_state = wb_ram_idle and CYC_I = '1') then
      ADR_COUNTER <= ADR_I(RAM_ADDRESS_WIDTH downto 0);
    elsif ( STB_I='1') then
      ADR_COUNTER <= ADR_COUNTER + 1;
    end if;
  end if;
  end process;

end wb_momentos;
```

ANEXO J: Simulações dos momentos 1 e 2; 3 e 4



Referências bibliográficas:

[1] Cappelatti, E. A. (2001), *Implementação do Padrão de Barramento PCI para interação Hardware/Software em Dispositivos Reconfiguráveis.* Dissertação de Mestrado, Pontifícia Universidade Católica do Rio Grande do Sul.

[2] PIACENTINO, Michael R., Gooitzen S. Van der Wal, Michael W. Hansen. (1999), *Reconfigurable Elements for a Video Pipeline Processor*. IEEE Symposium on FPGAs for Custom Computing Machines, pp 1-10.

[3] ORDONEZ, E. D. M. & Silva, J.L. (2000), *Reconfigurable Computing – Experiences and Perspectives*. Fundação de Ensino Eurípides Soares da Rocha (FEESR), Marília, 294pg., Agosto.

[4] HAUCK, S. (2000), *Reconfigurable Computing: A Survey of Systems and Software***.** Submitted to ACM Computing Surveys.

[5] IEEE Computer Society (2000), *A survey of Configurable Computing: Technology and Applications.* Computer, April.

[6] Dandalis, Andreas and Viktor K. Prasana. (2001). *Configuration Compression for FPGA-based Embedded Systems*, ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2001, pp 173-182, February.

[7] ANDERSSON, R. (1985). *Real-Time Gray-Scale Video Processing Using a Moment-Generating Chip.* IEEE Journal of Robotics and Automation, vol. 1, no. 2, junho. pp. 79-85.

[8] CHEN, K. (1990). *Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments.* Pattern Recongnition, vol. 23, no. ½, 1990, pp. 109-119.

[9] JIANG, X.; BUNKE, H. (1991). *Simple and Fast Computation of Moments.* Pattern Recognition, vol. 24, no. 8, 1991, pp. 801-806.

[10] WONG, W. H.; SIU, W. C.; LAM, K. M. (1995). *Generation of moment invariants and their uses for character recognition.* Pattern Recognition Letters, vol 16, n. 2.

[11] PCI SIG, *PCI Local Bus Specification. Revision 3.0.* http://www.pcisig.com Novembro/2004.

Fernandez, S; Modueri, C. (2003) Una Plataforma de desarrollo para el bus PCI. Instituto de Engenieria Electrica, Facultad de Ingenieria, Montevideo, Uruguay.

[13] Pedrino, E. C. (2003) Arquitetura Pipeline para Processamento Morfológico de Imagens Binárias em Tempo Real utilizando Dispositivos de Lógica Programável Complexa. Dissertação de Mestrado, EESC-USP.

[14] XILINX (1997), *XC4000E and XC4000X Series Field Programmable Gate Arrays Data Sheet.* Versão 1.4, Novembro.

[15] ALTERA (2000), *Altera Flex10K.* Disponível em www.altera.com, Junho/2004.

[16] Plessey Semiconductors (1989), *ERA 60100 Preliminary Data Sheet*. Swindon, England.

[17] Algotronix Ltd (1989), *CAL 1024 Data Sheet*. Edinburgh, Scotland.

[18] Concurrent Logic (1991), *CFA 6006 Field Programmable Gate Array Data Sheet.* Sunnyvale, California.

[19] Muroga H. et al. (1991), *A Large Scale FPGA with 10K Core Cells with CMOS 0.8um 3-Layered Metal Process.* Custom Integrated Circuits Conference CICC'91, pp.6.4.1-6.4.4, Maio.

[20] Rose, J. et al. (1993). *Architecture of Field-Programmable Gate Arrays*. Proceedings Of The IEEE, v.81, n.7, p. 1013-1029, Julho.

[21] Brown, S.; Rose, J. (1996). *FPGA and CPLD Architectures : A Tutorial*. IEEE Design & Test Of Computers, v.13, n.2, p. 42-57.

[22] DeHon A. (2000), *The Density Advantage of Configurable Computing*. IEEE Computer, pp.41-49, Abril.

[23] Estrin G. et al. (1963), *Parallel Processing in a Restructurable Computer System*. IEEE Transactions on Electronic Computers, pp. 747-755, Dezembro.

[24] A. Dehon - "*Reconfigurable Architecture for General-Purpose Computing*", Ph.D. thesis, Massachusetts Institute of Technology, 1996. JAIN, A., DUIN, R., AND MAO, J. Statistical pattern

[25] Villasenor J. and Mangione-Smith W. H. (1997), *Configurable Computing*. Scientific America, pp. 54-59, Junho.

[26] Mangione-Smith W. H. et al. (1997), *Seeking Solutions in Configurable Computing*. Computer, pp 38-43, Dezembro.

[27] Radunovic B. (1999), *An Overview of Advances in Reconfigurable Computing Systems.* Proceedings of the 32 Hawaii International Conference os System Sciences, pp. 1-10.

[28] Teifel, J. and Manohar, R. (2004), *Highly Pipelined Asynchronous FPGAs.* ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2004.

[29] Metzgen P. (2004), *A High Performance 32-bit ALU for Programmable Logic.* ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2004.

[30] Cong J., Fan Y., Han G., Zhang Z. (2004), *Application-Specific Instruction Generation for Configurable Processor Architectures.* ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2004.

[31] Wirthlin M. J. and Hutchings B. L. (1998), *Improving Functional Density Using Run-Time Circuit Reconfiguration.* IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Vol 6, No 2, Junho.

[32] Gokhale M. et al. (1997), *SPLASH: A Reconfigurable Linear Logic Array*. Disponível por ftp em ftp.super.org/pub/fpga/splash-1/splash-1.ps., Maio/2004.

[33] Vuillemin J. et al. (1996), *Programmable Active Memories: Reconfigurable*

Systems Come of Age. IEEE Transactions o VLSI Systems, vol. 4, pp. 56-69, Março

[34] Dalpasso M. et al. (2000). *Hardware/Software IP Protection*, Design Automation Conference – DAC'2000, Los Angeles, California, USA, pp 593-596, Outubro

[35] CHOW, P. et al. (1999a). *The Design of an SRAM-Based Field-Programmable Gate Array – Part I : Architecture.* IEEE Transactions On Very Large Scale Integration (VLSI) Systems, v.7, n.2, June.

[36] Bergamaschi R. A. and Lee W. R. (2000), *Designing System-on-Chip Using Cores*. Design Automation Conference – DAC'2000, Los Angeles, California, USA, pp 420-425.

[37] Ritter J., Molitor P. (2003) *A Pipelined Architecture for Partitioned DWT Based Lossy Image Compression using FPGA's.* ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2003.

[38] *Recognition: A review*. IEEE Transactions on Pattern Analysis and Machine Intelligence 22, 1 (January 2000), 4–37.

[39] Watanabe, S. *Pattern Recognition: Human and Mechanical*. Wiley, New York, 1985.

[40] PARKER, J.R. (1994). *Practical Computer Vision Using C*, Wiley.

[41] YUCEER, C.; OFLAZER, K. (1993). A rotation, scaling and translation invariant pattern classification system. Pattern Recognition, v. 26, n. 5.

[42] HU, M. (1961). *Pattern recognition by invariant moments*.Proc. IRE Transactions on Information Theory, 179–187.

[43] CHIN, R. T,; DYER, C. R. (1986). *Model-based recognition in robot vision*. Computing Surveys, vol 18, n. 1, 1986.

[44] PROKOP, R. J.; REEVES, A. P. (1992). *A survey of moment-base techniques for unoccluded object representation and recognition*. Graphical Models and Image Processing, vol 54, n. 5.

[45] C-H Tech , R. T. Chin, "On Image Analysis by Methods of Moments", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10 , No. 4, July, 1988, pp. 495-513.

[46] C.-H. Liu , W.-H. Tsai, "3D Curved Object Recognition from Multiple 2D Camera Views", *in Advances in Computer*, Academic Press, 1990, pp. 177-187.

[47] Guingo, B. C. Reconhecimento Automático de Placas de Veículos Automotores. Dissertação de Mestrado, UFRJ, 2004.

[48] GONZAGA, A and Costa,J.A.F. (1996). *Moment invariants applied to the recognition of objects using neural networks, Applications of Digital Image Processing* XIX, SPIE Proceedings, Ed. Andrew G. Tescher, Vol. 2847, pp.223-233.

[49] Marar, J.F., Papassoni, A.S., Castro, E.A de. - "*Estudo comparativo entre a transformada Hough e momentos invariantes em sistemas de reconhecimento de caracteres manuscritos.*", II Congresso Brasileiro de Computação – CBComp 2002.

[50] OPENCORES website, www.opencores.org.

[51] Keith Underwood (2004), *FPGAs vs. CPUs: Trends in Peak Floating Point Performance.* ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2004.

[52] VHDL-200X website, www.eda.org/fphdl/