PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DO PADRÃO DE BARRAMENTO PCI PARA INTERAÇÃO HARDWARE/SOFTWARE EM DISPOSITIVOS RECONFIGURÁVEIS

por

EWERTON ARTUR CAPPELATTI

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Fernando Gehm Moraes
Orientador

PORTO ALEGRE

2001

IMPLEMENTAÇÃO DO PADRÃO DE BARRAMENTO PCI PARA INTERAÇÃO HARDWARE/SOFTWARE EM DISPOSITIVOS RECONFIGURÁVEIS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Ciência da Computação, Área de Sistemas Digitais e Arquitetura de Computadores, Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, pela Comissão formada pelos professores:

Prof. Fernando Gehm Moraes – PUCRS/PPGCC/GAPH Orientador

Prof. Renato Perez Ribas – UFRGS

Prof. Dario Francisco G. de Azevedo – PUCRS/PPGE

Prof. Ney Laert Vilar Calazans – PUCRS/PPGCC/GAPH

Porto Alegre, xx de junho de 2001

Dados Internacionais de Catalogação na Publicação (CIP)

"Aprender é a única coisa
de que a mente nunca se cansa, nunca
tem medo e
nunca se arrepende."

Leonardo Da Vinci

AGRADECIMENTOS

Aos meus pais Iracema e Enio, por tudo que sou, a minha esposa Lisete e filhas Laura e Amanda, pela compreensão.

Agradeço ao Centro Universitário FEEVALE a oportunidade de participar, como aluno, do convênio firmado com o Programa de Pós-Graduação em Ciência da Computação – PPGCC da Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS, para a qualificação de seus docentes. Em especial, agradeço ao Prof. Dr. Cléber Cristiano Prodanov pela concretização deste convênio e pelo estímulo dado aos colegas professores participantes.

Meu agradecimento também aos colegas de turma: Anderson Cabral, Carlos Henrique Schwatrzhaupt, Delfim Luís Torok, Fábio de Freitas, Giovane Mesquita, Joaquim Brum, José Garibaldi, Josyane Barros, Marta Bez, Pedro Schuck, Reynaldo Novaes, Ricardo Pohlmann, Ricardo de Oliveira, Rodrigo Saad, Sandra Miorelli e Ezio Romeiro pelo apoio e companheirismo constantes.

Aos professores e funcionários do PPGCC, bem como aos colegas do Grupo de Apoio ao Projeto de *Hardware* - GAPH, especialmente ao Professor Ney Calazans, ao colega Delfim Torok e ao bolsista de iniciação científica Leandro Augusto de Oliveira, ficam, também, meus agradecimentos.

Um agradecimento, mais do que especial, ao meu orientador Professor Fernando Gehm Moraes, por sua dedicação e competência, fundamentais na concretização deste trabalho. Obrigado Amigo!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	*Application-Program Interface*
ASIC	*Application-Specific Integrated Circuit*
BAR	*Base Address Register*
BIOS	*Basic Input and Output System*
CAD	*Computer Aided Design*
CCM	*Custom Computer Machine*
CLB	*Configurable Logic Block*
CSH	*Configuration Space Header*
DCT	*Discrete Cosine Transform*
DEC	*Digital Equipment Corporation*
DECPeRLe	*Digital Equipment Corporation's Paris Research Laboratory*
EAB	*Embedded Array Block*
EEPROM	*Electrically Erasable Programmable Read Only Memory*
EISA	*Extended Industry Standard Architecture*
EPROM	*Erasable Programmable Read Only Memory*
FPGA	*Field Programmable Gate Array*
HEX	*Hexadecimal*
HOT	*Hardware Object Technology*
IBM	*International Business Machines*
ID	*Identificator*
I/O	*Input/Output*
IP	*Intellectual Propriety*
ISA	*Industry Standard Architecture*
LUT	*Look-up Table*
MCA	*Micro Channel Architecture*
Mb	*Mega bit*
MB	*Mega byte*
MHz	*Mega Hertz*
MPGA	*Mask-Programmable Gate Array*
PC	*Personal Computer*
PCI	*Peripheral Component Interconnect*
PCI-X	*Extended Peripheral Component Interconnect*
PCMCIA	*Personal Computer Memory Card International Association*
PLD	*Programmable Logic Device*
PMC	*Peripheral Component Interconnect Mezzanine Card*
PNP	*Plug and Play*
RAM	*Random Access Memory*
SCSI	*Small Computer System Interface*
SFFPCI	*Small Form Factor Peripheral Component Interconnect*
SIG	*Special Interest Group*
SoC	*System-on-chip*
SPCI	*Small Peripheral Component Interconnect*
SRAM	*Static Random Access Memory*
VHDL	*Very High Speed Integrated Circuits Hardware Description Language*
VLB	*Vesa Local Bus*
VLSI	*Very Large Scale Integration*

RESUMO

Esta dissertação apresenta o projeto de um módulo de *hardware* reutilizável, *soft core*, para a implementação do padrão PCI, 32 bits - 33 MHz. A principal motivação para o desenvolvimento deste *soft core* é prover aos projetistas de *hardware* um módulo que aumente a largura de banda na interação *hardware/software*. O trabalho apresenta as características gerais do padrão PCI, seguindo-se com a definição, na forma de diagrama de blocos, da arquitetura do *core*. A implementação deste *core* é feita utilizando-se a linguagem de descrição de *hardware* VHDL, validando-o através de simulação funcional. A simulação testa os ciclos básicos de leitura e escrita, tanto em modo simples quanto em rajada. Resultados preliminares de consumo de área e *timing* são apresentados. Apresenta-se também um ambiente de prototipação conectado ao barramento PCI, onde é mostrado o fluxo de projeto em aplicações que possuem componentes *software* e *hardware*.

Palavras-chave: PCI, FPGAs, *cores*, prototipação, arquiteturas reconfiguráveis, interação *hardware*/*software*.

ABSTRACT

This work presents the design of a soft core for the 32 bits - 33 MHz PCI interface. The main goal is to provide hardware developers with a standard functional block to be used in peripheral board design, specifically in the context of hardware/software codesign, minimizing the communication bottleneck between hardware and software parts. It begins presenting the general characteristics of the PCI interface, followed by the definition of the core architecture. The core is completely implemented using the hardware description language VHDL, and validated through functional simulation. This functional simulation tests the read and write cycles of the PCI bus, in simple and burst modes. Preliminary area and delay results, obtained from synthesis, are presented. Also, a prototyping environment connected to the PCI bus is employed to implement digital designs partitioned into hardware and software parts.

Keywords: PCI, FPGAs, cores, fast prototyping, reconfigurable architecture, hardware/software comunication.

1 INTRODUÇÃO

Os projetistas de sistemas digitais enfrentam sempre o desafio de encontrar o balanço correto entre velocidade e generalidade de processamento do 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 desenvolvido o *chip*, alterações na funcionalidade do circuito integrado não são possíveis. Logo, todo esforço despendido no seu projeto e implementação deve ser amortizado em um número elevado de unidades.

Uma solução intermediária, que representa um balanço entre custo *versus* versatilidade, são os *Field-Programable Gate Arrays* - FPGAs. Estes circuitos integrados configuráveis podem ser personalizados como diferentes ASICs. Esta tecnologia permite o projeto, teste e correção de circuitos integrados dedicados com um baixo custo de prototipação [22].

1.1 FPGAs

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, anti-fusível, porta flutuante e *flash* RAM. No caso dos FPGAs Xilinx (família 4000 e Virtex) [32], Altera (família Flex 10K) [3], Plessey [20], Algotronix [1], Concurrent Logic [8] e Toshiba [18], a tecnologia de programação utiliza *Static RAM*- SRAM. Sendo as SRAM 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 *Programmable Read-Only Memory* - PROM, *Erasable PROM* - EPROM, *Electrically EPROM* - EEPROM ou discos magnéticos. A maior desvantagem das SRAM é a área ocupada. São necessários pelo menos 5 transistores para implementar um 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 digitais e reconfiguração de arquiteturas, tanto estática quanto dinamicamente.

Como exemplo de arquitetura de um FPGA, podemos ilustrar os dispositivos Xilinx da família 4000 [32], conforme a Figura 1.

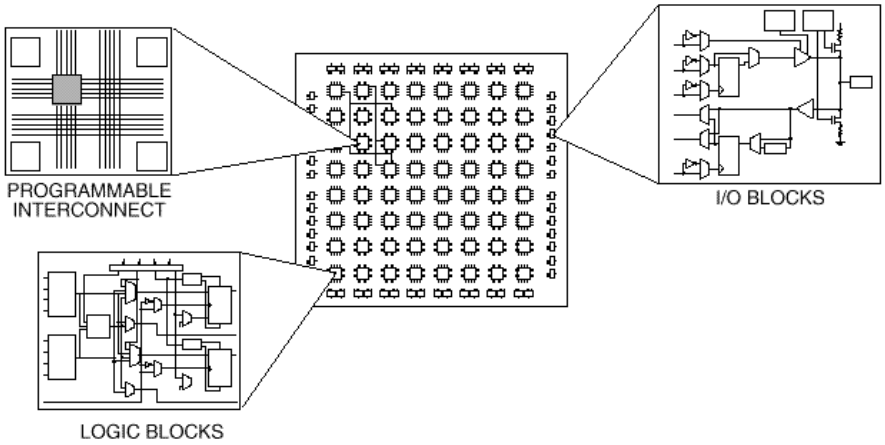

Figura 1 - Arquitetura de um FPGA XILINX, família 4000.

A estrutura dos FPGAs é similar aos *Mask-Programmable Gate Array* - MPGAs, consistindo de um arranjo de blocos lógicos que podem ser interconectados através de uma configuração para executar diferentes tarefas. A maior diferença entre os FPGAs e MPGAs é que o MPGA tem suas interconexões produzidas através da especificação de uma camada de metalização que necessita ser feita em uma *foundry*, enquanto que um FPGA é programado via chaves eletricamente programáveis. Os FPGAs podem alcançar níveis de integração muito mais elevados que os PLDs, embora possuam o roteamento de sua arquitetura e a implementação lógica mais complexos [22].

A arquitetura de um FPGA está dividida em 3 partes:

- **Blocos Lógicos Configuráveis**: Um bloco lógico de um FPGA pode ser tão simples como um transistor ou tão complexo quanto um microprocessador. Ele é capaz de implementar várias funções lógicas combinacionais (tipicamente através de *look-up tables*) ou seqüenciais (através de flip-flops).

- **Interconexões Programáveis**: SRAM – na qual a chave é um transistor de passagem controlado por um bit de RAM estática, Antifusível – quando eletricamente programado forma um caminho com baixa resistência e EPROM – quando a chave é um transistor com porta flutuante, que pode ser desligado ao se injetar uma carga em sua porta.

- **Blocos de Entrada/Saída Configuráveis**: os FPGAs possuem componentes de entrada/saída chamados I/O Blocks, formados por estruturas bidirecionais que incluem *buffer*, *flip-flop* de entrada, *buffer tri-state* e *flip-flop* de saída.

Ferramentas de CAD já disponíveis permitem implementar, testar, simular e corrigir projetos eletrônicos computacionais com grande facilidade sem que se tenha gastos elevados como

no desenvolvimento de ASICs. É possível simular a funcionalidade do projeto antes de produzi-lo, evitando assim desperdícios de recursos.

1.2 Arquiteturas reconfiguráveis

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 FPGA implementam o equivalente a uma única, e poderosa, instrução [11]. 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 [11] compara a implementação de um filtro digital, utilizando-se dispositivos programáveis e microprocessadores.

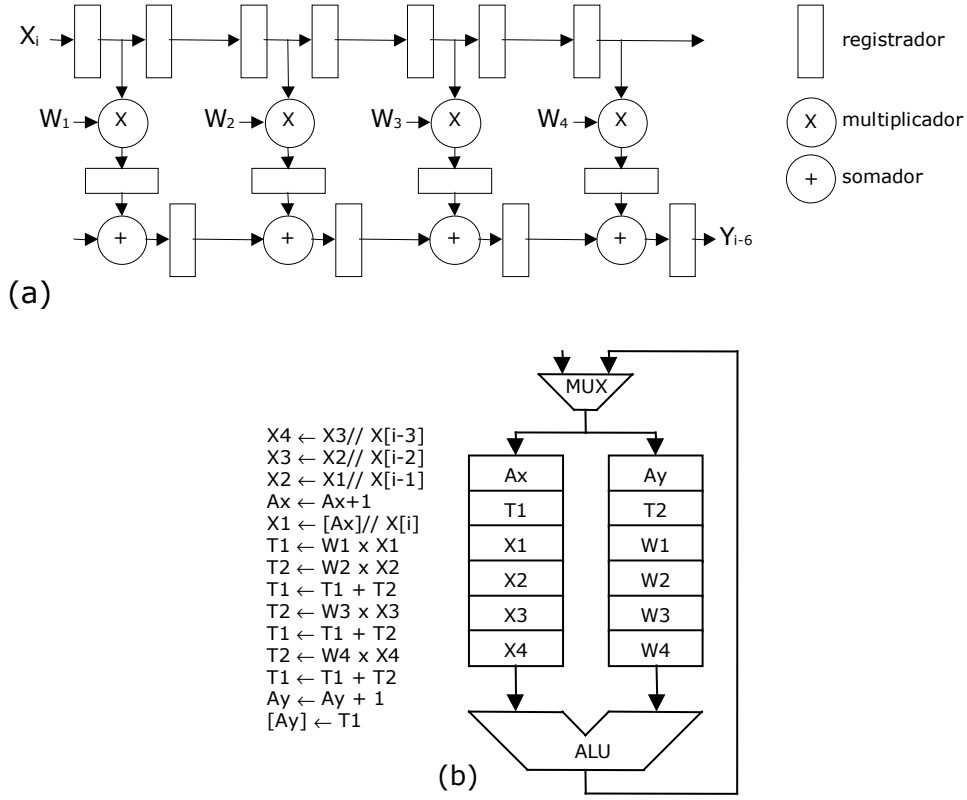

Figura 2 - Comparação de implementação entre FPGA e processador.
(a) computação paralela – FPGA; (b) computação temporal - microprocessador. Ambas implementações computam a expressão y[i]=w1.x[i] + w2.x[i-1] + w3.x[i-2] + w4 x[i-3], de um filtro com resposta a um impulso finito (FIR).

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 2.000.000 portas lógicas equivalentes em famílias VIRTEX-XILINX e APEX-ALTERA), possibilitou retomar o trabalho proposto no início da década de 60 por Gerald Estrin [12]. Estrin propôs um "computador com uma estrutura fixa e uma estrutura variável", no qual o *hardware* era dedicado tanto à abstração de um processador programável (inflexível) quanto a um componente que implementava a lógica digital (flexível). Esta arquitetura básica, que dá suporte a *hardware* programável e *software*, é o núcleo de muitos sistemas computacionais configuráveis subseqüentes [27]. 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*** [17][21] foram propostas e desenvolvidas. Entre os exemplos mais significativos estão [4][13][28][29][16][10][23].

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 [30].

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 comparados 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 [13]. 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 [28]. 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 dos 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 demonstra esta flexibilidade. Exemplos de aplicação que demonstram altos níveis de desempenho alcançados pelo DECPeRLe incluem [28]: 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, por exemplo, apresentaria níveis de desempenho muito mais baixos, pois teria que ser implementado praticamente como um processador genérico.

1.3 Interação Hardware-Software

A maior causa da redução do desempenho em sistemas computacionais é o tempo gasto em ler/escrever dados de/para memória e de/para periféricos (entrada/saída). Uma largura de banda de comunicação de dados pequena reduz o desempenho no processamento dos dados. Os sistemas operacionais multi-tarefas atuais e suas aplicações sofisticadas não requerem apenas processadores rápidos, exigem também uma maior vazão de dados de/para periféricos tais como discos rígidos e *hardware* de vídeo. Sendo assim, o barramento ISA usado em computadores pessoais, tornou-se um gargalo que se opõe ao aumento do desempenho geral do sistema [14].

A forma encontrada para continuar desenvolvendo processadores cada vez mais rápidos, cuja freqüência de operação muitas vezes é superior a dos componentes restantes do sistema, foi isolar o sub-sistema de entrada/saída do sub-sistema processador/memória/*cache*.

Neste contexto, foi desenvolvido o barramento *Peripheral Component Interconnect* - PCI, o qual aumenta a largura de banda provendo uma via para os dados capaz de transmitir até 528 MB/s (a 66 MHz, largura de barramento de 64 bits). Adicionalmente a essa otimização, o PCI também permite acesso direto à memória cache e dá suporte a uma arbitragem distribuída de barramento. Os componentes do barramento PCI e a interface das placas PCI são também independentes do processador [25].

Outra tendência observada hoje é o desenvolvimento de SoCs, onde o processador e a lógica programável são implementados no mesmo circuito integrado, minimizando assim o custo de transferência de dados. Pode-se citar como exemplo comercial a iniciativa Excallibur, da Altera [2].

1.4 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 Propriety* – IP) [9][5].

Outra tendência observada no projeto de circuitos VLSI é a integração completa de sistemas digitais em um único *chip* (*Sistem-on-Chip* – SoC) [5]. 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.

1.5 Motivação e Objetivos

Apresentou-se nas Subseções anteriores os temas que motivaram o desenvolvimento deste trabalho: FPGAs, arquiteturas reconfiguráveis, interação *hardware/software* e *cores*.

Para a implementação eficiente de arquiteturas reconfiguráveis é necessário que haja um meio rápido e eficiente para a comunicação *hardware/software*. Este meio de comunicação pode ser implementado como um *core* para FPGA, disponível para a comunidade acadêmica.

Esta dissertação versa sobre o desenvolvimento de um módulo pré-projetado e reutilizável de *hardware*, *core*, que implementa o padrão de barramento PCI. O objetivo é permitir uma rápida comunicação *software/hardware* através da utilização deste *core*. A utilização do padrão PCI reduz o tempo de transferência de dados entre os componentes *hardware* e *software* de um sistema digital, aumentando o desempenho global do sistema. Na medida em que se disponibilizam *cores* PCI, aplicações que necessitam de um protocolo ou interface de alto desempenho podem agregar à sua estrutura este *core* para conectarem-se ao barramento PCI de computadores hospedeiros. Entre as aplicações pode-se citar, por exemplo, a aceleração de algoritmos que possuem gargalos de desempenho. O gargalo de desempenho pode ser migrado para o *hardware*, e a interface PCI realiza a comunicação entre a aplicação do usuário, em *software*, e a parte crítica da aplicação, em

hardware. Outro fator a ser considerado é o alto custo financeiro para a aquisição de um *core* PCI. Por exemplo, a empresa XILINX comercializa um *firm core* PCI por cerca de US$ 10.000,00.

Além disto, projetar e construir um core PCI é um excelente meio de dominar a tecnologia de projeto de sistemas digitais com especificação de temporização muito estritas, implicando no uso de tecnologia no estado da arte para projeto, validação funcional, validação de temporização e validação de protótipo.

Esta dissertação está organizada da seguinte forma. O Capítulo 2 apresenta a descrição do barramento PCI, detalhando suas caraterísticas arquiteturais e os ciclos básicos de operação. Uma vez o barramento PCI descrito, o Capítulo 3 apresenta o *core* desenvolvido, além de outros encontrados na bibliografia. Este Capítulo é a contribuição maior deste trabalho, pois detalha o sistema digital implementado. O Capítulo 4 apresenta um ambiente para desenvolvimento de aplicações que utilizam o barramento PCI como meio de comunicação. Como plataforma de desenvolvimento, apresentaremos o ambiente HOT II-XL [26]. Finalmente, o Capítulo 5 apresenta as conclusões deste trabalho, assim como sugestões de trabalhos futuros.

2 ARQUITETURA PCI

Este Capítulo tem por objetivo apresentar as características do barramento PCI. Serão apresentadas, inicialmente, as características gerais deste, comparando-o com outros padrões de barramento. A seguir, o conjunto de sinais presentes no barramento PCI é listado, detalhando-se a função de cada sinal. A parte final deste Capítulo detalha os ciclos de operação do barramento, como leitura e escrita, em modo simples e rajada. A compreensão do funcionamento do barramento PCI é de fundamental importância para o desenvolvimento do *core*, o qual será apresentado no Capítulo seguinte.

2.1 Características Gerais

O *Peripheral Component Interconnect* - PCI é um barramento de alto desempenho, empregado para conectar componentes de uma placa mãe, bem como componentes destas à placas de extensão de um computador.

Este barramento foi proposto, primeiramente, pela empresa INTEL em dezembro de 1991 durante o *Intel Technical Forum*, e a primeira versão de sua especificação data de junho de 1992. Os membros do PCI *Steering Committee* que desenvolveram a primeira versão da especificação PCI foram a Compaq, DEC, IBM, INTEL e NCR. Em junho de 1992, o PCI tornou-se um padrão industrial aberto, orientado pelo PCI *Special Interest Group* – PCI SIG (www.pcisig.com). Em abril de 1993, chega a versão V2.0 do PCI . A revisão desta especificação, denominada V2.1, data de 1995. Atualmente, o padrão PCI encontra-se na especificação V2.2 [19].

O protocolo *Extended* PCI – PCI-X é um projeto avançado do padrão de barramento PCI. Permite que projetos com 64 bits de largura de barramento operem com freqüência de até 133 MHz. Este desempenho é alcançado pela implementação de um protocolo registrador-a-registrador. O PCI-X opera indistintamente no protocolo PCI V2.2 ou no próprio padrão PCI-X.

A Tabela 1 apresenta as principais características de algumas arquiteturas que antecederam o padrão PCI [15].

Tabela 1 - Arquitetura de barramentos mais comuns em computadores PC.

Arquitetura	Sistema típico	Largura do Barramento (bits)	Freqüência de operação (MHz)	Conectores de expansão	Característica
ISA (1984)	IBM PC AT	8/16	8	7	Mais comum e antigo
SCSI-1 (1986) Fast SCSI, Ultra SCSI, Ultra-2 SCSI (SCSI-3)	PCs de alto desempenho a partir do 80386	8/16	5/10/20/40	7/15	Comum, muitos padrões, freqüências de operação e larguras de barramentos de dados
MCA (1987)	IBM OS/2	16/32	10	3-7	Raro, antigo padrão da IBM
EISA (1988)	PCs baseados em 80386	32	8	7	Parcialmente compatível com o ISA
VLB (1992)	PCs baseados em 80486	32/64	40/50	2	Comum em computadores baseados em i486
PCI (1992)	PCs baseados em Intel Pentium e estações de trabalho modernas	32/64	33/66	4, mais *bridges*	Encontrado nos padrões: CompactPCI, PC/104-Plus, Small PCI, CardBus, PCI Mezzanine Card (PMC)

A Tabela 2 apresenta o desempenho do padrão PCI para diferentes larguras de barramento e freqüências de operação. Apesar de haver a norma para freqüência de operação a 133 MHz, no decorrer do texto mencionaremos apenas as freqüências 33 e 66 MHz.

Tabela 2 - Desempenho do padrão PCI na transferência de dados.

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

Uma parte fundamental do projeto PCI é a *bridge* (ponte) que conecta o barramento PCI ao barramento do processador (*PCI-to-host bridge*). Os periféricos PCI conectam-se diretamente ao barramento PCI, sendo *Plug-and-Play* – PNP (conectar e usar). Uma vez que a *bridge* é um componente presente na placa mãe, qualquer processador poderá ter acesso a todos componentes PCI do computador. Isto torna este padrão independente de processador. Para novos processadores basta apenas substituir a *bridge PCI-to-host*. O restante do sistema permanece inalterado. Mesmo que o processador seja mais rápido, não há problema, já que uma ponte *host-to-PCI* isola o conjunto processador/cache dos periféricos. A Figura 3 apresenta um esquema genérico de um barramento PCI.

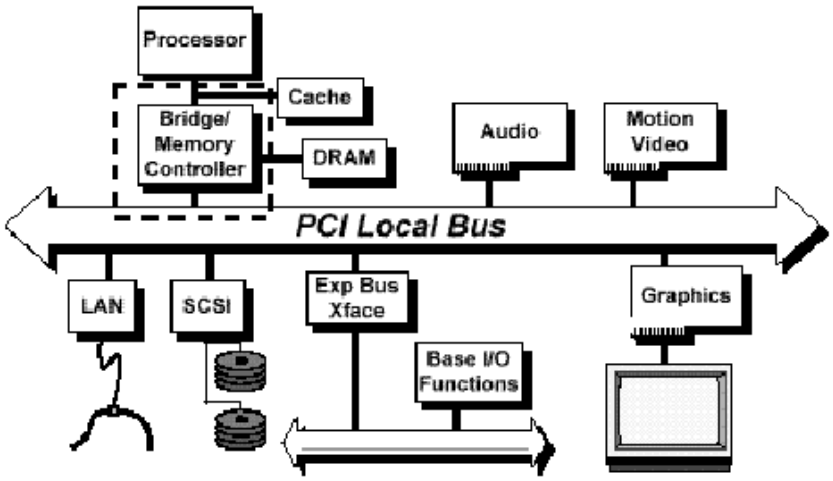

Figura 3 - Diagrama em blocos de um sistema PCI, em destaque a ponte.

Os protocolos de barramentos anteriores ao PCI permitiam apenas ciclos de acessos simples de leitura e escrita. Diferentemente, o PCI permite uma transferência de dados (leitura/escrita) em modo *burst* (rajada), o que melhora o seu desempenho. Em um acesso simples dos outros barramentos, há necessidade de se informar um novo endereço a cada transferência. No modo *burst*, múltiplas transferências para/de endereços consecutivos são realizadas, tendo um comprimento indefinido. A transferência em modo *burst* continua até que algum dispositivo PCI no barramento solicite o final da transferência.

O barramento PCI possui as seguintes características:

- Dados e endereços são multiplexados nas mesmas linhas de sinal (AD), e os comandos de barramento e largura dos dados são multiplexados nas mesmas linhas de sinal (C/BE#). Isto traz como conseqüência a divisão dos ciclos de acesso em fases de endereçamento de dados. Durante a fase de endereçamento o endereço do dispositivo alvo e o comando a ser executado são colocados no barramento. Durante a fase de dados, os dados e a indicação de quantos *bits* se pode utilizar (8, 16, 24 ou 32 *bits*) são colocados no barramento. A indicação do número de *bits* a ser utilizado na fase de dados é especificada pelo sinal C/BE#.
- O espaço de endereçamento é de 32 bits com opção de ser estendido para 64 bits.
- A largura do barramento de dados é definida com 32 bits com opção de ser estendido para 64 bits.
- O padrão de acesso aos dispositivos conectados ao barramento PCI é em modo *burst* (rajada), o qual permite múltiplos acessos a dados em uma única operação de escrita ou leitura.
- Enquanto nenhuma operação ocorre no barramento PCI, as linhas de sinal são mantidas em nível lógico alto devido a resistores *pull-up* ou em um nível estável pelo *master*.
- Geração e verificação de paridade para endereços, dados e sinais de controle .
- Com exceção do sinais SERR#, RST# e INT# (descritos na Seção 2.5.1), todos os sinais são síncronos em relação ao *clock* (CLK).

2.2 Operação Plug and Play do barramento PCI

O termo *Plug-and-Play* - PNP, refere-se à capacidade do sistema computacional de determinar automaticamente os recursos necessários para cada dispositivo instalado no barramento PCI. Estes recursos são mapeados de forma a evitar conflitos no sistema, o que acontece com os dispositivos que não são PNP, os chamados *legacy* (legados), que necessitam ser configurados por *jumpers*. São exemplos de dispositivos legados os controladores de portas serial/paralela e os controladores conectadas aos *slots* ISA.

A especificação PCI define registradores de configuração que devem estar presentes em cada dispositivo PCI. Estes registradores têm por objetivo portar informações acessíveis pelo mecanismo PNP do sistema hospedeiro para alocar recursos a cada dispositivo conectado ao barramento PCI.

São três os componentes necessários para um sistema PCI PNP:

- Registradores de configuração;
- *PCI Basic Input and Output System* - PCI BIOS;
- *Software* para componentes PNP.

2.2.1 Registradores de Configuração

Segundo a especificação do padrão PCI, cada dispositivo compatível com este barramento contém uma memória de 256 bytes, utilizada pelos registradores de configuração. O conjunto completo destes registradores está apresentado na Tabela 3.

Tabela 3 - Registradores de configuração de um dispositivo PCI compatível.

Endereço base (HEX)	Tamanho (*bytes*)	Nome do Registrador	Parâmetro PNP (S/N)
00	2	*Vendor ID*	S
02	2	*Device ID*	S
04	2	*PCI Command*	N
06	2	*PCI Status*	N
08	1	*Revision*	N
09	3	*Class Code*	N
0C	1	*Cache Line Size*	N
0D	1	*Latency Timer*	N
0E	1	*Header Type*	N
0F	1	*Built In Self Test (BIST)*	N
10	4	*Base Address Register 0 (BAR 0)*	S
14	4	*Base Address Register 1 (BAR 1)*	S
18	4	*Base Address Register 2 (BAR 2)*	S
1C	4	*Base Address Register 3 (BAR 3)*	S
20	4	*Base Address Register 4 (BAR 4)*	S
24	4	*Base Address Register 5 (BAR 5)*	S
28	8	*Reserved*	N
30	4	*Expansion ROM Base Address*	N
34	8	*Reserved*	N
3C	1	*Interrupt Line Register*	S
3D	1	*Interrupt Pin*	S
3E	1	*Minimum Grant*	N
3F	1	*Maximum Latency*	N

Existem quatro tipos de registradores que são importantes para o sistema PNP que são:

- ***Device* e *Vendor ID***. São empregados juntos para identificar o fabricante de um dispositivo em particular. O Vendor ID é atribuído pelo PCI SIG, o que garante um único número de identificação para cada fabricante. Cada fabricante atribui a seu Device ID valores que garantem um único identificador para cada um de seus produtos.

- ***BAR***. Possui um duplo propósito. A partir do *power on*, o BAR identifica os recursos do sistema necessários para cada dispositivo. Cada dispositivo pode utilizar até seis registradores BAR para identificar até seis espaços de endereços individuais de vários tamanhos para serem empregados pelo dispositivo. O valor inicial armazenado em cada BAR por ocasião do *power on* identifica qual o tipo de recurso está sendo solicitado. O bit menos significativo classifica o dispositivo como sendo de entrada/saída ou um espaço de memória. Os bits restantes indicam quanto espaço é solicitado. Para completar a configuração PNP o *host* escreve o valor do endereço base para o dispositivo no BAR.

- ***Interrupt Line Register***. Este registrador é utilizado para identificar a linha de interrupção a ser utilizada pelo dispositivo PCI compatível. Em sistemas *Personal Computer* – PC, este registrador recebe o valor da interrupção de *hardware* utilizada pelo dispositivo.

Outros registradores do espaço de configuração contêm outras informações como *Class Code*, que serve para identificar o tipo de dispositivo, e registradores para a requisição do barramento, como *Minimum Grant* e *Latency timer* (temporizador de latência) que são parâmetros necessários para o arbitramento do barramento PCI. Estes 3 registradores supra definidos, bem como outros parâmetros, não são necessários para as funções PNP.

Os dispositivos PCI reconhecem três espaços de endereçamento diferentes:

- Endereçamento de Memória (*Memory Address*);
- Endereçamento de Entrada/Saída (*I/O Memory*);
- Endereçamento de Memória de Configuração (*Configuration Memory*).

Os espaços de endereçamento de memória e de entrada/saída são definidos como nos processadores Intel e utilizados em acessos normais de dados. Todos os dispositivos conectados ao barramento PCI, sejam estes componentes *on board*, ou placas padrão PCI inseridas nos *slots* da placa mãe de um computador hospedeiro, possuem uma Memória de Configuração (*Configuration Memory*). Esta Memória de Configuração tem um tamanho de 256 *bytes*, e armazena informações sobre os recursos dos dispositivos PCI presentes no sistema. A Memória de Configuração possui uma parte pré definida, que é formada pelos seus primeiros 64 *bytes* e os 192 *bytes* restantes formam a parte dependente do dispositivo.

A Figura 4 ilustra a parte pré-definida da Memória de Configuração. O conteúdo destes registradores corresponde ao apresentado na Tabela 3. Estes registradores são acessados por seus endereços nos ciclos de acesso à configuração, como por exemplo, a inicialização do computador.

A Memória de Configuração deve estar sempre acessível e, quando lida, todas as 256 posições de memória devem retornar dados válidos. As posições de memória destacadas na Figura 4 devem sempre conter dados referentes ao dispositivo em questão. Para dispositivos multifunção, devem ser implementadas memórias de configuração separadas para cada função.

Bytes	***Bits***			
	31	**16**	**15**	**0**
00h	*Device ID*		*Vendor ID*	
04h	*Status*		*Command*	
08h	*Class Code*			*Revision ID*
0Ch	*Built-IN Self-Test*	*Header Type*	*Latency Timer*	*Cache Line Size*
10h	*Base Address Register - BAR*			
14h				
18h				
1Ch				
20h				
24h				
28h	*Card Bus CIS Pointer*			
2Ch	*Subsystem ID*		*Subsystem Vendor ID*	
30h	*Expansion ROM Base Address*			
34h	*Reserved*			
38h	*Reserved*			
3Ch	*Max Lat*	*Min Gnt*	*Interrupt Pin*	*Interrupt Line*

Figura 4 -Espaço de Configuração pré definido para todos os dispositivos compatíveis com o padrão PCI.

2.2.2 PCI BIOS

A configuração PNP dos dispositivos conectados ao barramento PCI é controlada pelo PCI BIOS no momento da inicialização do sistema (*boot time*). A Figura 5 apresenta o diagrama do PCI BIOS.

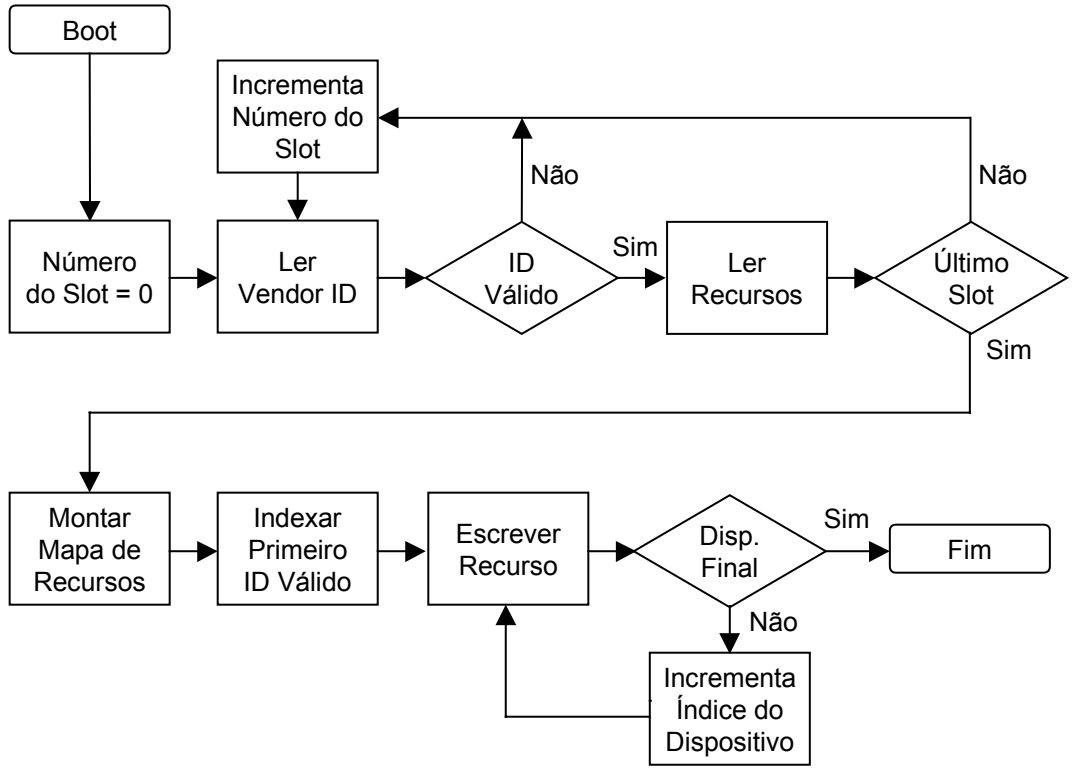

Figura 5 - Fluxograma da configuração PNP do PCI BIOS.

O BIOS inicia a seqüência de configuração PNP endereçando individualmente cada *slot* PCI de um conector PCI da placa mãe, de maneira a determinar se um dispositivo válido está nele inserido. A verificação do dispositivo é feita através da leitura do registrador *Vendor ID*. Um valor $FFFF_{(HEX)}$ (devido a resistores *pull up*) indica um componente inválido, ou nenhum instalado. Se outro valor for retornado pelo registrador *Vendor ID*, o sistema fará a leitura do BAR e do *Interrupt Register* do dispositivo.

Este procedimento se repete até que todos os *slots* PCI tenham sido acessados, e com isso o sistema torna-se informado sobre I/O, memória e interrupções solicitados individualmente pelos dispositivos conectados ao barramento.

Assim, o sistema gera um mapa para cada dispositivo, garantindo a exclusividade deste espaço a cada um deles, evitando assim conflitos. O PCI BIOS escreve o endereço base de cada espaço atribuído no BAR apropriado e a interrupção é escrita no *Interrupt Register*. Terminadas estas tarefas, encerra-se a configuração PNP do *hardware*.

2.2.3 Software para componentes PNP

O *software* a nível de sistema utilizado para operar um dispositivo necessita de informações como endereços de I/O, endereços de memória e interrupções. Em sistemas não PNP, os recursos necessários são determinados pelo *hardware*. Historicamente, vários métodos têm sido utilizados para definir o espaço de endereços do hardware tais como *jumpers* e *dip switches*. Neste caso o *software* é direcionado para utilizar endereços pré determinados. Isto permite ao *hardware* operar assumindo que não há outro dispositivo configurado com os mesmos recursos.

O PNP pressupõe uma abordagem diferente para o *software*. Nos dispositivos PNP, o mapeamento dos recursos de *hardware* dentro do espaço do sistema não é pré determinado pelo *hardware*. O espaço alocado para cada dispositivo depende de vários fatores, incluindo a arquitetura do sistema, o PCI BIOS e a presença de outros dispositivos PNP conectados ao sistema. Devido a estes fatores, os controladores (*drivers*) de *hardware* são os responsáveis por determinar os recursos a serem alocados para cada dispositivo a partir da inicialização do sistema. A Figura 6 mostra o fluxograma do algoritmo a ser executado na inicialização do computador em sistemas PNP.

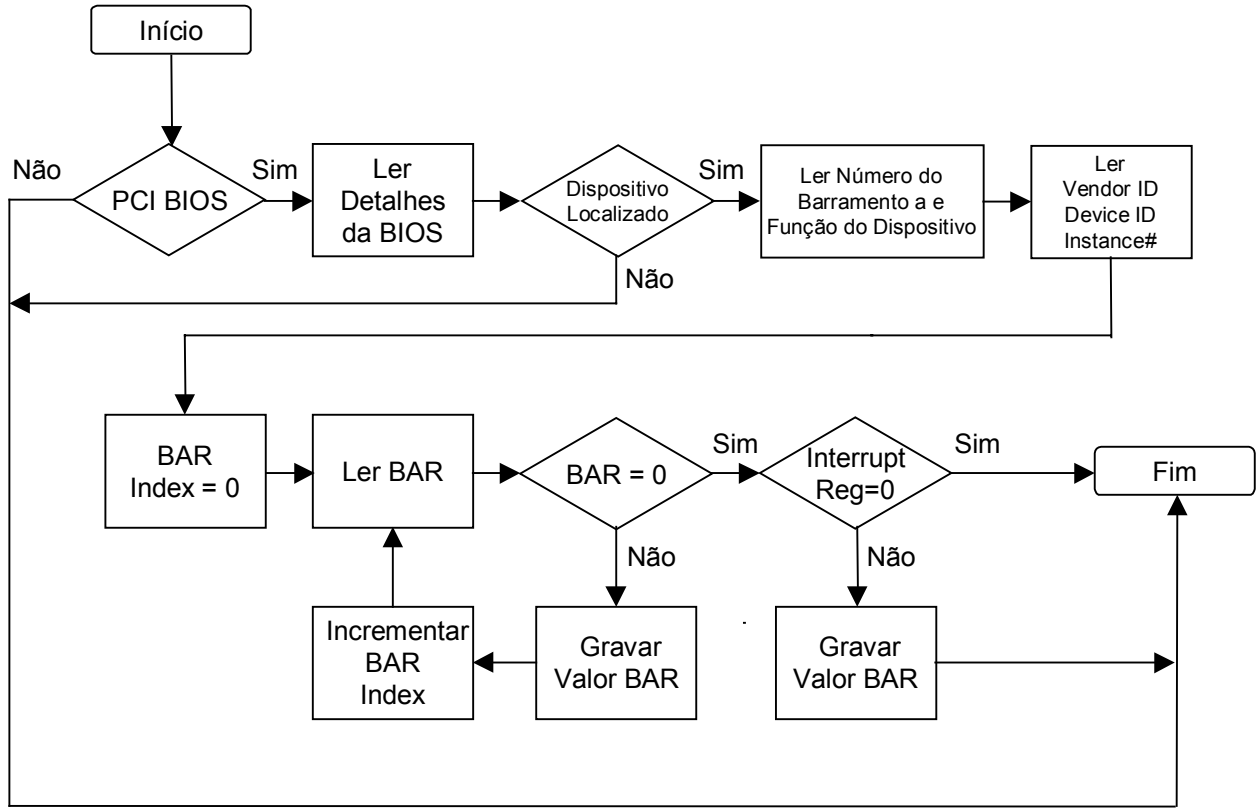

Figura 6 - Fluxograma do *software* de um sistema PNP.

A primeira tarefa a ser realizada pela BIOS em um sistema PNP é verificar a presença do PCI BIOS. Isto é feito com o emprego de uma função de baixo nível da BIOS, a qual verifica a presença de uma PCI BIOS. A função do BIOS determinará o número da revisão do barramento PCI bem como, se há mais de um barramento PCI presente no sistema, o numero do último barramento. O número da revisão PCI refere-se à *PCI Local Bus Specification* (Especificação PCI do Barramento Local) utilizada no projeto da BIOS. O número do último barramento PCI refere-se ao fato que, embora seja transparente ao usuário, muitos sistemas contêm mais do que um barramento PCI em sua placa mãe. Ligado a isto está, também, a limitação que a especificação impõe de quatro *slots* PCI por barramento. Logo, se o sistema for dotado de mais *slots*, ele certamente estará separado em mais de um barramento.

A segunda tarefa é a localização dos dispositivos PCI conectados ao sistema. Isto se faz através da chamada de uma função PCI BIOS e especificando um *Vendor ID* e um *Device ID* únicos e um valor de índice. Dado que cada tipo de dispositivo conectado ao barramento PCI contém uma única combinação, estes identificadores podem ser utilizados para localizar um dispositivo desejado em um sistema. O valor do índice é necessário devido ao fato de que o barramento pode possuir múltiplas ocorrências de um mesmo dispositivo. O índice identifica cada dispositivo a ser acessado de forma unívoca. A função do PCI BIOS retorna o número do barramento e o índice do dispositivo localizado. Estes dois parâmetros são utilizados para garantir um acesso único aos registradores de configuração de um dispositivo particular.

A tarefa final do nível de *software* é a de determinar os recursos atribuídos a cada dispositivo. Isto é feito através da leitura de valores armazenados nos BAR de cada dispositivo. Há outra função do PCI BIOS que retorna o valor do BAR baseado no número do barramento e no valor da função do dispositivo determinados na tarefa anterior. Os dispositivos PCI não utilizam todos os espaços disponíveis no BAR. Um valor igual a zero no BAR indica que o dispositivo não solicitou qualquer recurso a ser atribuído ao BAR. Uma leitura de um BAR que retorna o valor padrão zero, indica que o sistema PNP não atribuiu nenhum espaço ao BAR. Caso contrário, o valor retornado pelo BAR indica qual o tipo de recurso, I/O ou memória foi atribuído, e o endereço base do início do espaço que será utilizado pelo dispositivo. O nível de *software* deve, então, utilizar estes endereços para acessar o dispositivo.

2.3 Característica físicas das placas padrão PCI

Na medida em que os barramentos aumentam sua largura, os conectores, circuitos integrados e trilhas na placa mãe aumentam, elevando assim os custos de produção. O padrão PCI, para reduzir custos, multiplexa endereços e dados.

A primeira versão do barramento PCI trabalhava com uma tensão de 5Vcc. A partir da versão V2.0, de junho de 1995, ficou determinado que a alimentação das placas periféricas PCI fosse de 3,3Vcc. A Figura 7 apresenta detalhes quanto a tensão de alimentação.

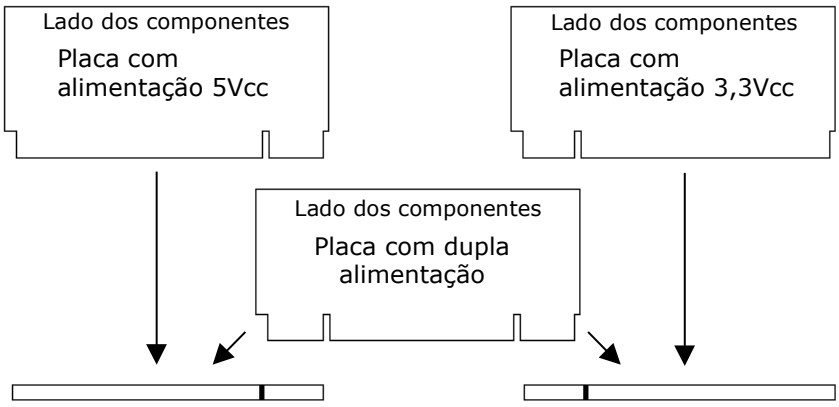

Figura 7 - O polarizador colocado no conector determina qual o nível de tensão presente [19].

Quanto às dimensões físicas, dois tamanhos de placas podem ser encontrados: 12” (tamanho padrão) e 7” de comprimento. A Figura 8 ilustra estas medidas.

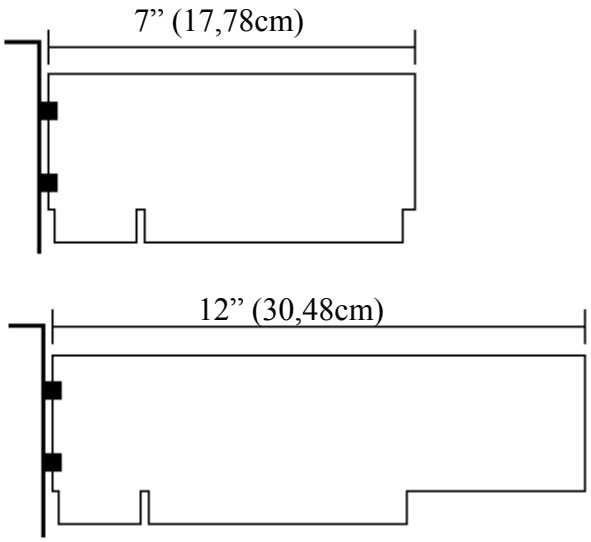

Figura 8 - Tamanhos padronizados para placas PCI.

Uma outra especificação, denominada SPCI, originalmente chamada de *Small Form Factor* – SFF PCI, é uma definição mecânica para um tipo de placa e conector com dimensões de um cartão de crédito para ser empregado em espaços reduzidos e/ou equipamentos portáteis. O padrão é funcionalmente compatível com o protocolo PCI Revisão 2.1. A Revisão 1.0 do SPCI foi editada pelo PCI-SIG em 1° de maio de 1996. Há muitas semelhanças físicas entre o SPCI e os *Personal Computer Memory Card International Association* – PCMCIA. As dimensões são as mesmas, porém o SPCI utiliza um conector com 108 pinos contra os 68 do PCMCIA. Existem duas espessuras para os cartões SPCI: Tipo A, com 5,0 mm e Tipo B com 10,5 mm, correspondendo aos padrões Tipo II e Tipo III do padrão PCMCIA, respectivamente.

Todas as publicações referentes ao padrão PCI estão disponíveis no endereço http://www.pcisig.com. O PCI SIG é o responsável pelas especificações de protocolo, características elétricas e mecânicas do padrão PCI.

2.4 Arbitragem do barramento

Toda operação PCI ocorre entre dois componentes distintos: o *master* e o *target*. O *master* é o dispositivo que inicializa e gerencia o barramento PCI, podendo ser o processador ou uma *bridge*. Já o *target* é o dispositivo alvo, aquele que responde a um acesso, como por exemplo uma controladora de vídeo, memórias e I/O. A Figura 9 ilustra estes componentes.

Figura 9 - O *master* comanda o barramento e troca informações com o *target*.

Somente um *master* pode controlar o barramento PCI em cada ciclo de barramento que esteja sendo executado. Outro *master*, que não aquele ativo no momento, pode solicitar a arbitragem através do sinal LOCK#. Dá-se o nome de arbitragem ao controle e gerenciamento do barramento PCI durante os ciclos de barramento. Esta arbitragem ser feita por qualquer dispositivo *master* que necessite do barramento. Esta solicitação é feita para o árbitro central, que é a entidade que está gerenciando o barramento no momento da solicitação.

Dois sinais estão envolvidos na solicitação da arbitragem do barramento: REQ# e GNT#. O sinal REQ# é o sinal utilizado por um *master* para solicitar o barramento, que recebe como resposta positiva à solicitação a ativação do sinal GNT#. O dispositivo *master* que deseja gerenciar o barramento deve ativar o sinal REQ# somente quando ele estiver pronto para iniciar, imediatamente, um novo ciclo. As linhas dos sinais REQ# e GNT# são individuais para cada dispositivo e roteadas ponto a ponto até o árbitro central, que é uma *bridge*. A ativação dos sinais REQ# e GNT# é sincronizada com o sinal de CLK.

Existe mais de um algoritmo de distribuição de arbitragem pela *bridge* no PCI. Por exemplo, supondo que existem 3 dispositivos que podem operar como *master* (A, B e C). O árbitro central garantirá primeiramente ao dispositivo A o acesso ao barramento. Se A não quiser assumir o barramento ou já concluiu sua utilização, o árbitro central passará o controle para B. Após A e B concluírem suas tarefas o árbitro passará o controle para C. Dado que A, B e C já utilizaram o barramento, o ciclo volta para A, e assim por diante, como se executasse um *pooling*.

Outro algoritmo associa a cada dispositivo um nível de prioridade. Quando dois *masters* requisitarem o barramento simultaneamente, aquele que tiver a maior prioridade assume o controle do barramento. Para garantir que os outros *masters* tenham seu direito garantido, o árbitro central não cede a um *master* o controle do barramento sem que todos os outros tenham sido servidos.

Existem outros protocolos de arbitragem, porém todos tem o cuidado de não deixar de atender a solicitação de algum *master* (latência muito grande), de garantir o tempo necessário na utilização do barramento (garantir latência) e de, independente da solicitação, oferecer o controle do barramento a qualquer *master*.

2.5 Sinais do barramento PCI

O PCI tem duas larguras de barramento, 32 e 64 *bits*, e freqüências de operação de 33MHz e 66MHz, respectivamente. Os dispositivos PCI com barramento de 32 *bits* possuem 124 pinos, e os com barramento de 64 *bits* tem 188 pinos. Um dispositivo *target*, 32 bits, deve possuir no mínimo 47 pinos disponíveis à *interface* do barramento (todos os pinos obrigatórios menos GNT# e REQ# - estes sinais serão definidos na Seção seguinte), e um dispositivo *master* deve ter um mínimo de 49 pinos (todos os pinos obrigatórios). Para se chegar ao número total de pinos, devem ser considerados diversos pinos de alimentação 3,3 Vcc, 5,0 Vcc e terra [25].

2.5.1 Descrição dos sinais

A Figura 10 ilustra os sinais obrigatórios (para *master* e *target*), bem como os sinais para operação em barramento de *64 bits* [14]. Os sinais marcados com # são ativos em nível lógico 0, e os que estão em negrito fazem parte do conjunto mínimo de sinais que um dispositivo compatível com o padrão PCI deve manipular em modo *target*.

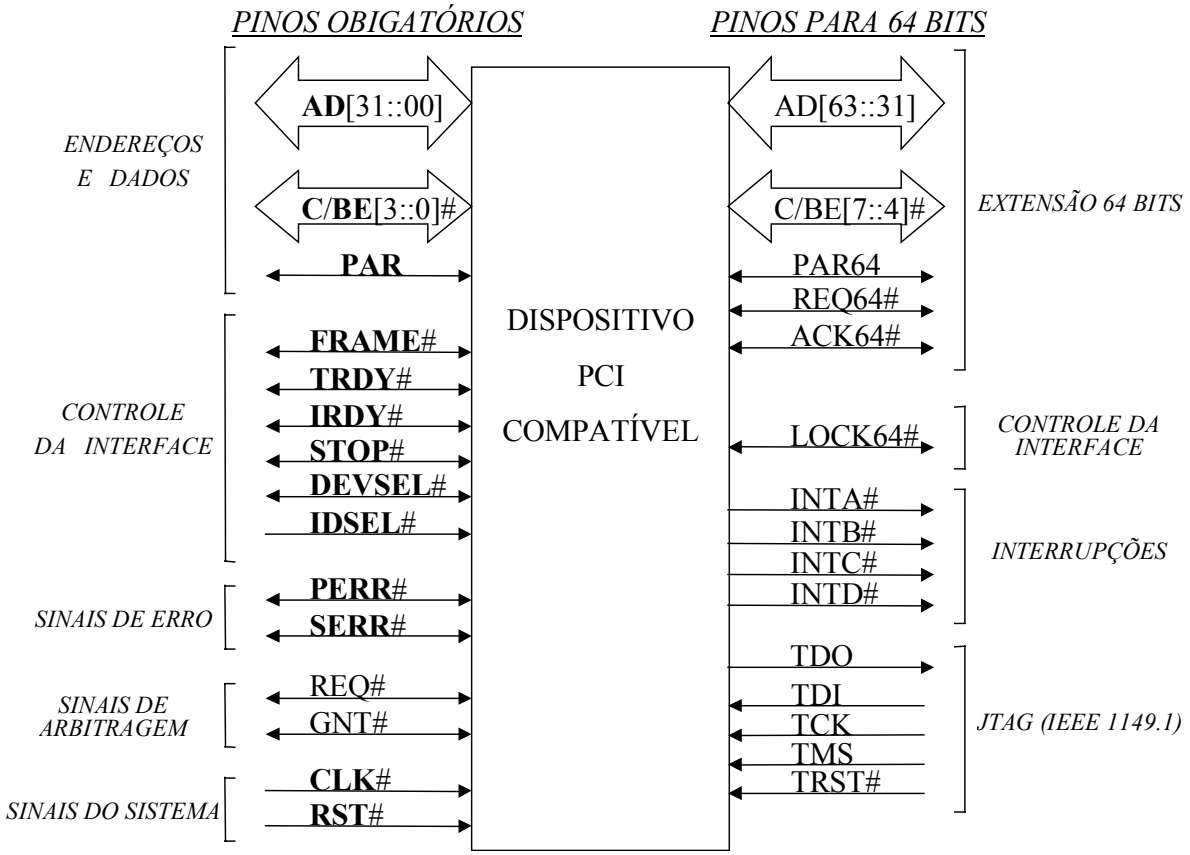

Figura 10 - Sinais presentes em dispositivos compatíveis com PCI, 32/64 bits.

- **CLK**: *Clock* do sistema (33/66 MHz). O protocolo do barramento PCI é definido como síncrono e todos os sinais no barramento PCI são sensíveis à borda de subida do pulso de *clock*, exceto os sinais SERR# (*system error*), RST# (*reset*) e as linhas de sinal de INT# (*interrupt*).
- **RST#**: Inicializa todos os registradores de configuração. Sinal assíncrono em relação ao *clock* do sistema.
- **AD[31::00]**: O barramento de endereços é multiplexado com o de dados. Uma transação de barramento se inicia com uma fase de endereçamento seguida por uma ou várias fases de dados. O barramento PCI suporta leitura/escrita em modo *burst*.
- **C/BE#**: *Command/Byte Enable* – sinais multiplexados. Durante a fase de endereçamento de uma transação de barramento, este sinal possui um comando (**C**) que identifica qual a operação que se inicia; durante a fase de dados de uma transação determina qual(is) a(s) via(s) do barramento de dados que possuem dados válidos (**BE#**).
- **PAR**: Paridade – paridade par associada à concatenação de AD[31::0] e C/BE[3::0].
- **FRAME#**: Indica o início e o fim de uma transação de barramento. É comandado pelo *master*. FRAME# vai a nível lógico 1 quando o *master* está pronto para completar a fase final da transação.
- **IRDY#**: *Initiator Ready* – durante uma transação de escrita, o *master* do barramento aciona este sinal colocando-o em "0", indicando que um dado válido está disponível no barramento. Durante uma transação de leitura, o *master* do barramento aciona este sinal, indicando que está pronto para receber um dado do *target*. Durante transações de leitura e escrita, *wait states* podem ser acrescentados no IRDY# enquanto estiver ativo.
- **TRDY#**: *Target ready* – Durante uma transação de leitura, o *target* aciona este sinal para indicar que ele disponibilizou um dado válido no barramento PCI. Durante uma transação de escrita, o *target* aciona este sinal para indicar que ele está pronto para receber um dado do barramento PCI. Durante transações de leitura e escrita, *wait states* podem ser acrescentados no TRDY#, enquanto estiver ativo.
- **IDSEL**: *Initialization Device Select* - Utilizado na inicialização do sistema (*boot*) como um *chip select* durante o acesso aos registradores de configuração (leitura ou escrita).
- **DEVSEL#**: Sinal gerado pelo *target* quando seu endereço é decodificado. Se este sinal permanecer inativo durante 6 ciclos de *clock* após a transferência de dados ter iniciado, o *master* aborta a transferência de dados.
- **STOP#**: Sinal gerado pelo *target* para solicitar ao *master* o fim de uma transação.
- **PERR#**: *Parity error* – Sinaliza erro de paridade de dados durante as transações do barramento, exceto quando em *Special Cycle*.
- **SERR#**: *System error* – Sinaliza erro de paridade de endereçamento, paridade de dados ou qualquer erro severo.

- GNT#: *Grant* - Sinaliza ao *master* que o controle do barramento está garantido.
- REQ#: *Request* - Sinaliza ao árbitro atual do barramento que um novo *master* necessita acesso ao barramento.

Uma importante observação quanto à multiplexação das linhas de dados e endereços é que quando o *master* solicita dados do *target*, necessita-se de um ciclo extra no protocolo para a inversão da direção do barramento. Este fato fica mais claro na discussão sobre os ciclos de escrita e leitura, abordados na Seção 2.8.

A especificação PCI permite que o tamanho dos dados em suas vias seja identificado dinamicamente. Não há troca de dados (*swap*) nas vias de dados. Logo, todos os dispositivos PCI compatíveis devem dar suporte a uma largura de barramento de dados de 32 bits.

O barramento AD[31::0] contém 4 vias de dados com largura de um byte. AD[7::0] compreende a primeira via, AD[15::8] a segunda, AD[23::16] a terceira e AD[31::24] a quarta via. Estas vias podem ser acessadas independentemente, conforme o valor especificado em C/BE#.

2.6 Cálculo da paridade

A especificação do padrão PCI provê uma capacidade de detecção de erros limitada. As linhas C/BE# e AD são protegidas com paridade par sinalizadas pelos sinais PAR e PAR64, sendo que PAR é para a versão 32 bits, e PAR64 é utilizado nas operações em 64 bits. As linhas PERR# e SERR# são utilizadas para sinalizar erros de paridade e erros com a plataforma hospedeira. As informações de erro são armazenadas em registradores e podem ser acessadas pelo BIOS, pelo sistema operacional e pelo *software* de aplicação.

A paridade é utilizada em todos os ciclos do barramento PCI. A geração da paridade não é opcional e deve ser computada e sinalizada em uma das linhas PAR/PAR64. Contudo, a paridade não precisa ser verificada por todos os dispositivos conectados ao barramento. Os dispositivos que não verificam e reportam erros de paridade são aqueles que satisfazem as seguintes condições:

- Recursos que estão inseridos diretamente na placa mãe e não em placas de inserção (*add-on cards*), ou
- Recursos que não processam dados permanentemente, por exemplo, placas de vídeo.

Em outras palavras, recursos que não causarão perda no desempenho no processamento da plataforma quando um erro de paridade ocorrer e não suportam verificação de paridade.

A primeira fase de uma transação de barramento chama-se de fase de endereçamento, na qual o *master* coloca um endereço nas linhas AD e um comando nas linhas C/BE. Um ciclo de *clock* depois, o *master* coloca na linha PAR o bit de paridade correspondente, calculado sobre os

bits de AD e C/BE#. Este atraso de um ciclo de *clock* deve-se ao fato de que o *target* precisa calcular a paridade entre AD e C/BE recebidos e depois receber o sinal PAR para poder comparar com a paridade que calculou. Se o bit recebido na linha PAR for igual ao da paridade calculada pelo *target*, a linha PERR# (*Parity Error*) é mantida desativada, isto é, em nível lógico 1. A segunda fase de uma transação de barramento é a fase de dados. A partir desta fase a paridade é calculada entre os dados colocados em AD e os sinais de *byte enable* colocados em C/BE. Para cada dado e *byte enable*, a paridade é calculada e um sinal PAR é gerado, seja em ciclos de leitura ou de escrita.

Nas figuras da Seção 2.8 está ilustrado o funcionamento do cálculo da paridade.

2.7 COMANDOS DE BARRAMENTO (COMMAND)

Os comandos de barramento indicam ao *target* o tipo de transação que o *master* está requisitando. No início de cada fase de endereçamento, as linhas C/BE[3::0]# carregam o código do comando. A Tabela 4 apresenta os tipos de comando, seguida por uma breve descrição de cada um [15][19].

Tabela 4 - Códigos dos comandos de barramento PCI.

C/BE[3::0]#	Comando
0000	*Interrupt Acknowledge*
0001	*Special Cycle*
0010	*I/O Read*
0011	*I/O Write*
0100	*Reserved*
0101	*Reserved*
0110	*Memory Read*
0111	*Memory Write*
1000	*Reserved*
1001	*Reserved*
1010	*Configuration Read*
1011	*Configuration Write*
1100	*Memory Read Multiple*
1101	*Dual Address Cycle*
1110	*Memory Read Line*
1111	*Memory Write and Invalidate*

- ***Interrupt Acknowledge***: Este comando faz uma requisição ao controlador de interrupções do sistema.

- ***Special Cycle***: Este comando provê um mecanismo de *broadcast* no PCI.
- ***I/O Read***: Comando utilizado para ler dados de um dispositivo mapeado no espaço de endereçamento I/O.
- ***I/O Write***: Comando utilizado para escrever dados em um dispositivo mapeado no Espaço de Endereçamento de I/O.
- ***Reserved***: Para utilização futura. Caso o código deste comando seja utilizado no barramento, o acesso do dispositivo PCI *target* será abortado pelo *master*.
- ***Memory Read***: Comando utilizado para ler dados de algum dispositivo mapeado no Espaço de Endereçamento de Memória.
- ***Memory Write***: Comando utilizado para escrever dados de algum dispositivo mapeado no Espaço de Endereçamento de Memória.
- ***Configuration Read***: Este comando é utilizado para ler o Espaço de Configuração de cada dispositivo PCI. Um dispositivo PCI é selecionado durante um acesso de configuração quando o sinal IDSEL é ativado e AD[1::0] é "00".
- ***Configuration Write***: Comando utilizado para a transferência de dados ao Espaço de Configuração de cada dispositivo PCI. O endereçamento para transações de escrita de configuração é o mesmo que para transações de leitura de configuração.
- ***Memory Read Multiple***: Este comando é semanticamente idêntico ao comando *Memory Read*, exceto que adicionalmente indica que o *master* pretende fazer buscas a mais de uma linha de memória antes de desconectar.
- ***Dual Address Cycle***: Comando empregado na transferência de endereços de 64 *bits* para dispositivos que suportam 64 *bits* de endereçamento quando o endereço não se encontra no espaço de endereços abaixo de 4 GB, ou seja, possui pelo menos 1 dos 32 bits mais significativos em "1". Os dispositivos *target* que suportam somente 32 *bits* de endereçamento devem tratar este comando como reservado, não podendo responder a este comando.
- ***Memory Read Line***: Este comando é semanticamente idêntico ao comando *Memory Read*, exceto que adicionalmente indica que o *master* pretende fazer uma busca completa à memória *cache* do *host*.
- ***Memory Write and Invalidade***: Este comando é semanticamente idêntico ao comando *Memory Write*, exceto que adicionalmente garante uma transferência mínima de uma linha completa de *cache*, isto é, o *master* pretende escrever todos os *bytes* na linha de *cache* endereçada em uma única transação de barramento sem interrupção pelo *target*.

Todos os dispositivos PCI (com exceção das *bridges* do barramento hospedeiro) devem responder como *target* no comando configuração (escrita ou leitura). Todos os outros comandos são opcionais.

A Tabela 5 apresenta os comandos mínimos para um dispositivo operar somente como *target*.

Tabela 5 - Comandos de barramento essenciais para um projeto PCI *target* [15].

C/BE[3::0]#	Comando
0010	*I/O Read*
0011	*I/O Write*
0110	*Memory Read*
0111	*Memory Write*
1010	*Configuration Read*
1011	*Configuration Write*

2.8 CICLOS BÁSICOS DE OPERAÇÃO EM 32 BITS

O barramento PCI suporta transferências de dados em modo *burst* (rajada), que consistem em uma fase de endereçamento seguida por uma ou mais fases de dados. O modo de transferência *burst* é o mesmo tanto para transferência de dados para um dispositivo *back-end* (aplicação do usuário), ou para os registros de configuração. Uma alta taxa de transferência é alcançada devido ao fato de que múltiplas fases de dados são realizadas para cada fase de endereçamento, ao invés de uma fase de dados para uma fase de endereçamento para o modo não *burst* (simples) [25].

Os ciclos descritos a seguir são:

- Ciclo de acesso de Leitura em Modo Simples;
- Ciclo de acesso de Escrita em Modo Simples;
- Ciclo de acesso de Leitura em Modo *Burst*;
- Ciclo de acesso de Escrita em Modo *Burst*;
- Ciclo de acesso à Configuração e
- Ciclo Especial.

2.8.1 Ciclo de Acesso de Leitura em Modo Simples

A Figura 11 apresenta os sinais durante este ciclo. Um ciclo de acesso simples de leitura inicia quando o *master* do barramento PCI ativa o sinal FRAME#, colocando um tipo de comando (COMMAND) válido nas linhas C/BE[3::0]# e um endereço válido de dispositivo nas linhas AD[31::0] (1). Esta fase denomina-se fase de endereçamento (*ADDRESS PHASE*) do ciclo de acesso e dura um ciclo de *clock*. Completada esta fase o *master* ativa o sinal IRDY# (*initiator ready*) indicando que está pronto para começar a ler os dados do *target*. No mesmo instante o sinal FRAME# é desativado (2). O dispositivo que contiver o endereço colocado pelo *master* em AD, indica que foi selecionado ativando o sinal DEVSEL# (*device selected*) (3). Imediatamente após a fase de endereçamento segue-se a fase de dados (*DATA PHASE*). No início desta fase o *master* do barramento habilita um subconjunto de *bytes* válidos pelo sinal C/BE[3::0]# e coloca as linhas AD[31::0] em alta impedância, pois o barramento deve ter sua direção invertida (o *master* enviou o endereço e o *target* enviará seus dados – barramento bidirecional) (4). Quando o *target* estiver pronto para enviar os dados ele ativa o sinal TRDY# (*target ready*) e os dados são enviados (5). Este ciclo termina com o *master* desativando o sinal IRDY# e os sinais de controle são removidos pelo *target* (6). Observar que ao final da fase de endereçamento e da fase de dados do ciclo de leitura, há o cálculo da paridade entre C&A (*command* e *address*) e BE&D (*byte enable* e *data*) (7).

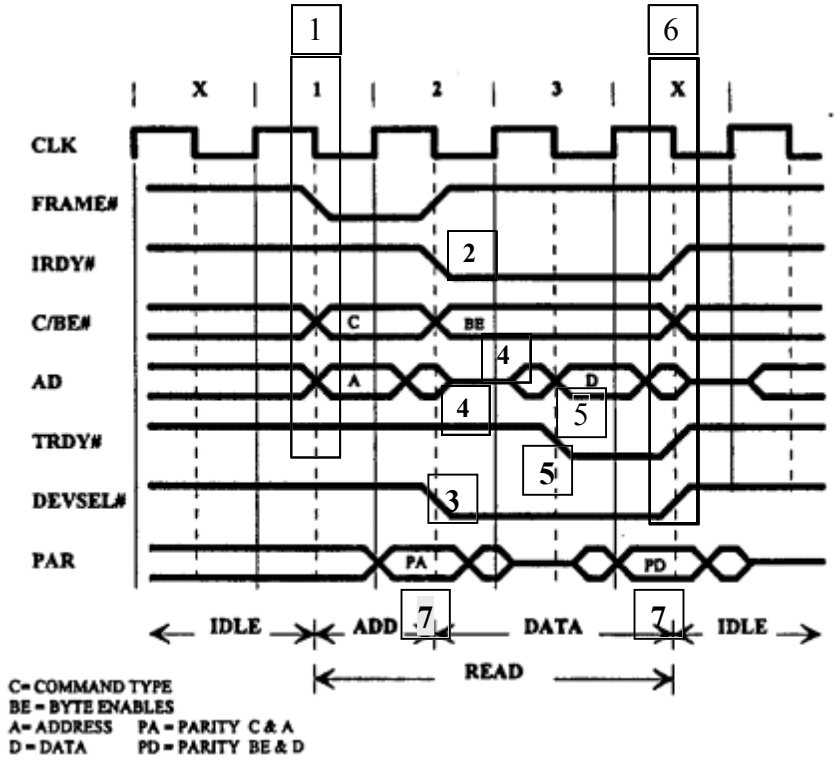

Figura 11 - Sinais do ciclo de acesso de leitura no modo simples. Os sinais são ativos em nível lógico zero, pois as linhas do barramento são mantidas em *pull-up*.

2.8.2 Ciclo de Acesso de Escrita em Modo Simples

Os sinais deste ciclo são mostrados na Figura 12. Um ciclo de acesso de escrita em modo simples inicia similarmente ao ciclo de leitura simples. Inicia quando o *master* do barramento PCI ativa o sinal FRAME#, colocando um tipo de comando (*COMMAND*) válido nas linhas C/BE[3::0]# e um endereço válido nas linhas AD[31::0] (1). Esta fase denomina-se fase de endereçamento (*ADDRESS PHASE*) do ciclo de acesso. Completada esta fase o *master* ativa o sinal IRDY# (*initiator ready*) indicando que está pronto para começar a ler os dados do *target*. No mesmo ciclo o sinal FRAME# é desativado (2). O dispositivo que contiver o endereço colocado pelo *master* em AD, indica que foi selecionado ativando o sinal DEVSEL# (*device selected*) (3). Quando o *target* estiver pronto para receber os dados ele ativa o sinal TRDY# (*target ready*) e os dados são enviados (4). Este ciclo termina com o *master* desativando o sinal IRDY# e os sinais de controle são removidos pelo *target* (5). Observar que ao final da fase de endereçamento e da fase de dados do ciclo de escrita há o cálculo da a paridade entre C&A (*command* e *address*) e BE&D (*byte enable* e *data*) (6).

Uma transação de escrita é similar à transação de leitura, exceto que não há a necessidade da inversão do barramento, durando um ciclo de *clock* a menos. Um dado válido poderá não estar presente a cada início de uma fase de dados, pois o *target* pode não estar pronto. Neste caso são inseridos *wait states* pelo *target*.

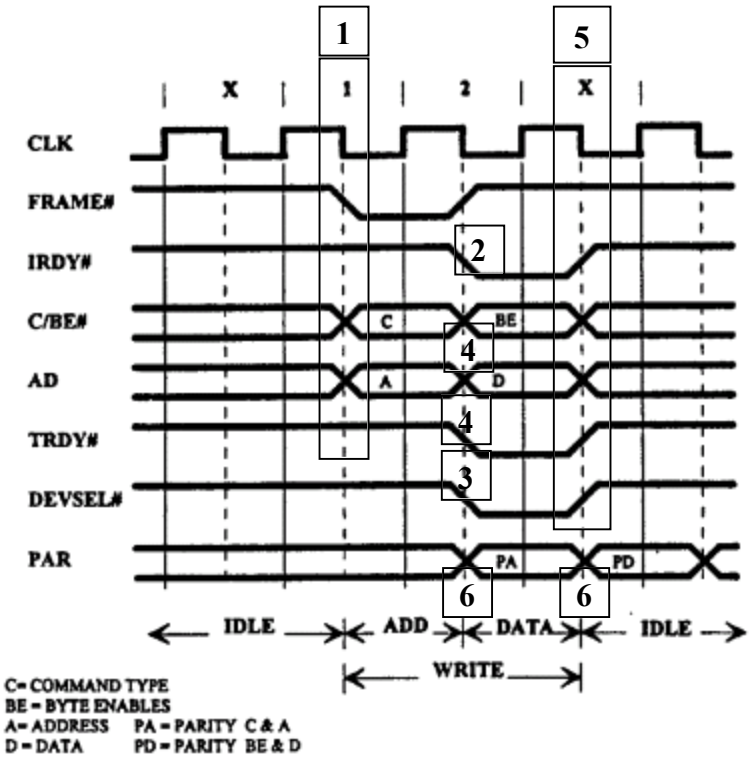

Figura 12 - Sinais do ciclo de acesso de escrita no modo simples.

2.8.3 Ciclo de Acesso de Leitura em Modo *Burst*

A Figura 13 mostra os sinais para um ciclo de leitura em modo *burst*. O modo *burst* permite que o *master* acesse o *target* através de uma seqüência de micro acessos (1 ciclo de *clock*). O ciclo inicia quando o *master* ativa o sinal FRAME#, coloca a informação de comando do ciclo em C/BE[3::0]# e um endereço válido nas linhas de sinal AD[31::0] (1). Este endereço estabelece o endereço base para todos os micro acessos. Um ciclo de *clock* depois, o *master* ativa o sinal IRDY# (*initiator ready*) indicando que está pronto para começar a ler os dados do *target* (2). A partir deste ponto inicia a fase de dados. No início desta fase, o *master* do barramento habilita um subconjunto de *bytes* válidos pelo sinal C/BE[3::0]# e coloca as linhas AD[31::0] em alta impedância, pois o barramento deve ter sua direção invertida (o *master* enviou o endereço e o *target* enviará seus dados – barramento bidirecional) (3). Quando o *target* selecionado estiver pronto para enviar os dados ele ativa os sinais DEVSEL# e TRDY#, respectivamente (4). A partir deste momento os dados são enviados (5). Este ciclo termina com o *master* desativando o sinal IRDY# (7) um ciclo de *clock* após o FRAME# (6) ter sido desativado, e os sinais de controle são removidos pelo *target* (7). Observar que ao final da fase de endereçamento e da fase de dados do ciclo de leitura, há sempre o cálculo da a paridade entre C&A (*command* e *address*) e BE&D (*byte enable* e *data*) para cada micro acesso (8).

O que diferencia um acesso de leitura no modo simples do acesso de leitura no modo *burst* é o tempo de permanência do sinal FRAME# ativado.

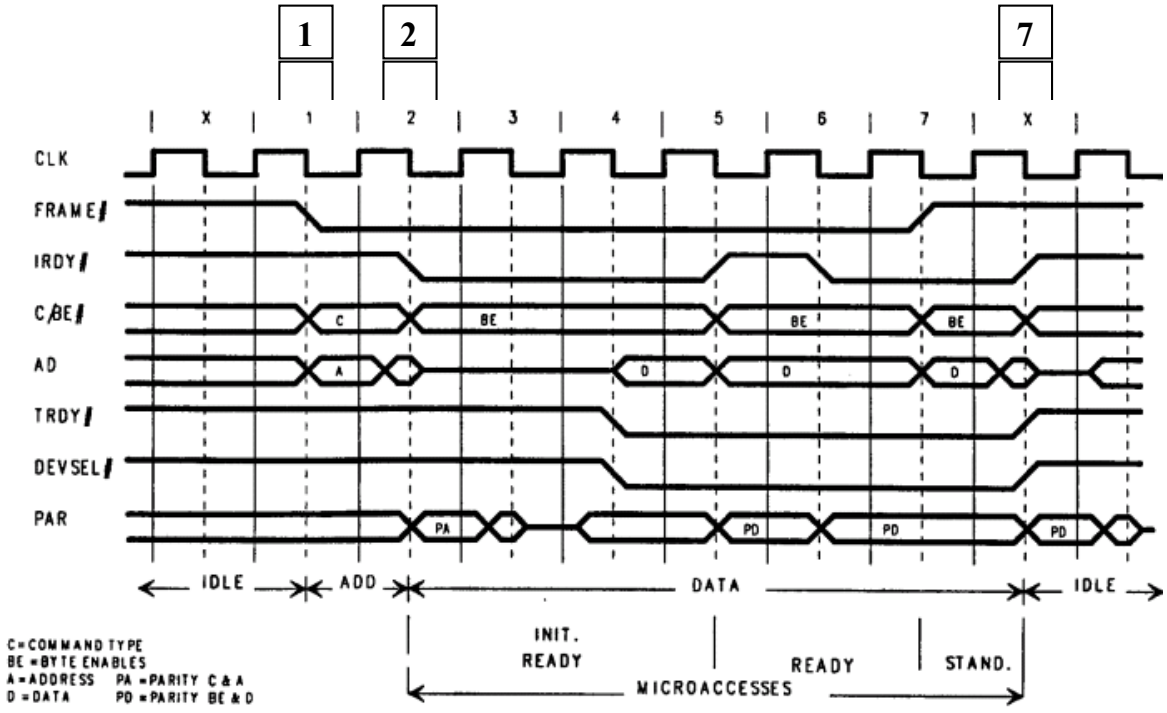

Figura 13 - Sinais do ciclo de acesso de leitura no modo *burst*. O sinal FRAME# permanece ativo (nível lógico 0) durante todo o ciclo de leitura, que se encerra um ciclo de *clock* depois, quando IRDY# é desativado.

No exemplo de transação de barramento ilustrado na Figura 13, verifica-se que em (9) o *master* desabilita por um ciclo de *clock* o sinal IRDY# caracterizando um ciclo de *wait state* (estado de espera). Neste caso, o *master* está indicando que não está apto a receber dados. O *target* suspende então, o envio de dados até que o *master* acione (coloque em nível lógico "0") o sinal IRDY#.

2.8.4 Ciclo de Acesso de Escrita em Modo *Burst*

A Figura 14 mostra os sinais para um ciclo de escrita em modo *burst*. O modo *burst* permite que o *master* acesse o *target* através de uma seqüência de micro acessos. O ciclo inicia quando o *master* ativa o sinal FRAME#, coloca a informação de comando do ciclo em C/BE[3::0]# e um endereço válido nas linhas de sinal AD[31::0] (1). Este endereço da fase de endereçamento estabelece o endereço base para todos os micro acessos. Quando o *target* selecionado estiver pronto para enviar os dados ele ativa os sinais DEVSEL# e TRDY#, respectivamente (2). Ativando o sinal IRDY# (*initiator ready*) o *master* indica que está pronto para começar a enviar os dados para o *target* (3). A partir deste ponto inicia a fase de dados. No início desta fase, o *master* do barramento habilita um subconjunto de *bytes* válidos pelo sinal C/BE[3::0]# e coloca nas linhas AD[31::0] o dado a ser enviado. A partir deste momento os dados são enviados (4). Este ciclo termina com o *master* desativando o sinal IRDY# (6) um ciclo de clock após o FRAME# (5) ter sido desativado, e os sinais de controle são removidos pelo *target.* Observar que ao final da fase de endereçamento e fase de dados do ciclo de escrita, há sempre o cálculo da paridade entre C&A (*command* e *address*) e BE&D (*byte enable* e *data*) para cada micro acesso (7). A desativação do sinal TRDY# (8) pelo *target* caracteriza um ciclo de *wait state* (ciclo de espera) necessário para poder atender o *mestre*. Tal procedimento pode ocorrer devido ao fato de que um dispositivo que esteja sendo acessado ser mais lento que o *master*.

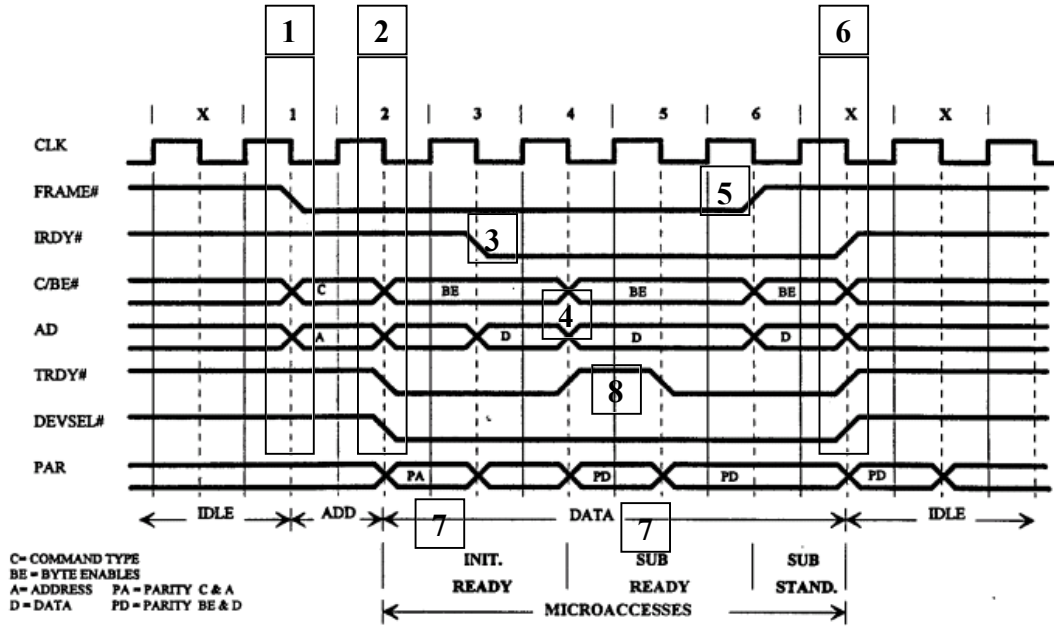

Figura 14 - Sinais do ciclo de acesso de escrita no modo *burst*. O sinal FRAME# permanece ativo (nível lógico 0) durante todo o ciclo de escrita que se encerra um ciclo de *clock* depois, quando IRDY# é desativado.

O que diferencia um acesso de escrita no modo simples do acesso de escrita no modo *burst* é o tempo que o sinal FRAME# permanece ativado.

2.8.5 Ciclo de Acesso à Configuração

A especificação do protocolo PCI suporta registradores de configuração em cada dispositivo PCI (PCI *bus master*, *target* ou *bridge* são definidos como dispositivos). Este ciclo executa seus acessos da mesma forma que os ciclos de leitura e escrita, podendo ser, também, em modo simples ou *burst*. Todo dispositivo PCI que é acessado por um ciclo de acesso à configuração possui 64 registradores de 4 *bytes* conforme mostrado na Seção 2.2.1. Estes registradores são acessados através das linhas AD[31::11] (*Initialization Device Select* - IDSEL), que atuam como um *chip select* tradicional. Durante os ciclos de configuração, o dispositivo alvo do acesso corrente pode não saber seu endereço, dado que ainda não foi configurado. Neste caso, o *target* é selecionado através do sinal IDSEL na inicialização (*power up*) do computador hospedeiro.

O ciclo de acesso à configuração tem como objetivo determinar as necessidades de cada dispositivo PCI conectado ao sistema.

Existem dois tipos de ciclos de acesso à configuração: Tipo 0 e Tipo 1. O Tipo 0 é efetuado em um dispositivo que se encontra no mesmo barramento em que o ciclo de configuração está sendo executado. O Tipo 1 é executado somente entre PCI/PCI *bridge*.

O ciclo de acesso à configuração começa por iniciativa da unidade central de processamento do computador hospedeiro, tendo como componente ativo as *bridges*, que funcionam como conexão entre os barramentos PCI, operando assim como *masters*. O ciclo inicia com o *master* (*bridge*) ativando o sinal FRAME#, colocando um comando válido em C/BE[31::0]# e estabilizando as linhas AD[31::0]#. Para um acesso à configuração, sinais em AD[31::11] são anexados à linhas IDSEL de cada dispositivo. Durante a fase de endereçamento, somente um dispositivo é selecionado por IDSEL.

Logo após a fase de endereçamento vem a fase de dados. O protocolo da fase de dados, tanto para o modo de acesso simples ou em modo rajada, é o mesmo que nos acessos à memória ou I/O.

A ativação do sinal FRAME# no ciclo de configuração ocorre mais tarde do que nos ciclos de acesso à memória ou I/O, o que leva as linhas IDSEL do *target* acessado anteriormente a um estado de alta impedância antes de utilizar estas linhas para o *target* atual.

2.8.6 Ciclos Especiais

O protocolo PCI define uma versão do ciclo de escrita chamado Ciclo Especial. Este ciclo é direcionado a todos os recursos disponíveis no barramento PCI, isto é, permite escrita simultânea em todos os dispositivos instalados neste barramento. Logo, o Ciclo Especial ocorre no mesmo barramento PCI em que o *master* que está executando o ciclo está instalado. Uma *bridge* não tem condições de propagar o Ciclo Especial entre dois barramentos.

Nenhum dispositivo PCI solicita o Ciclo Especial, logo o sinal TRDY# não é ativado por dispositivos *target* e a duração do ciclo é controlada pelo *master* do barramento via sinal IRDY#. Também, quando um Ciclo Especial é executado, nenhum dispositivo (*master*) pode solicitar o barramento.

Os sinais deste ciclo são mostrados na Figura 15. Um Ciclo Especial inicia com o *master* do barramento ativando a linha de sinal FRAME#, colocando um comando válido em C/BE[31::0]# e levando as linhas AD[31::0] a um nível estável (1). Um ciclo de clock depois, aciona o sinal IRDY# indicando que está pronto para este ciclo. A colocação do sinal FRAME# em nível lógico “1” (2) indica que a transação será no modo simples. Esta atividade compreende a fase de endereçamento do Ciclo Especial (ADD na Figura 15). Imediatamente após a fase de endereçamento vem a fase de dados. Em (3) e (4) há o cálculo da paridade entre C&A (*command* e *address*) e BE&D (*byte enable* e *data*), respectivamente. O ciclo termina com a desativação do sinal IRDY# pelo *master* (5). O protocolo da fase de dados em modo simples ou *burst* para Ciclos Especiais é o mesmo que os de ciclos de acesso a memória ou I/O para os mesmos dois modos de acesso. Ao contrário de um ciclo de acesso, os sinais TRDY#, DEVSEL# e STOP# permanecem desativados durante toda a fase de dados. O mínimo período de um ciclo especial é de cinco períodos de *clock*.

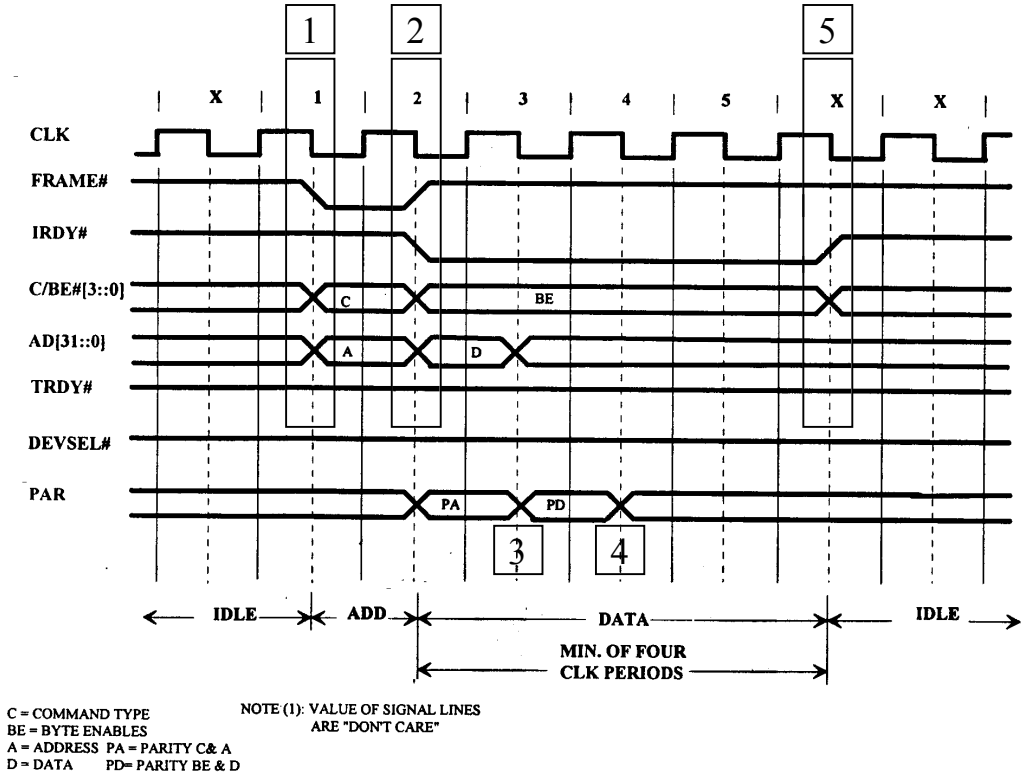

Figura 15 - Formas de onda de um Ciclo Especial do tipo simples.

3 IMPLEMENTAÇÃO DE MÓDULOS DE HARDWARE PARA O BARRAMENTO PCI

Este Capítulo define e classifica módulos de *hardware* denominados *cores*. Inclui também exemplos de *cores* que implementam o padrão PCI, descrito no Capítulo anterior. O objetivo principal deste Capítulo é o de apresentar o desenvolvimento de um *core*, principal contribuição deste trabalho, para arquiteturas reconfiguráveis, a ser disponibilizado para aplicações que necessitem bom desempenho na interação *hardware*/*software*. Cada etapa do desenvolvimento do *core* está acompanhada de seu código em VHDL e de sua simulação.

3.1 Cores

A complexidade atual dos circuitos digitais induz os projetistas de *hardware* a utilizarem módulos pré-projetados, denominados *cores* [7]. Muitos *cores* atualmente existentes são protegidos por leis de propriedade intelectual e só podem ser adquiridos com um alto investimento financeiro. Além disto estes *cores* são vendidos em forma de "caixa preta", muitas vezes não permitindo acesso a seu código fonte.

Sistemas baseados em *cores* apresentam benefícios tais como: redução do tempo de desenvolvimento de projetos pelo reuso de propriedade intelectual, diminuição do risco de inserção de erros ao longo do projeto e desenvolvimento de sistemas mais complexos.

Os *cores* podem ser classificados em 3 categorias [7]:

- ***Hard cores***. São otimizados para uma tecnologia específica e não podem ser modificados pelo projetista. Possuem a vantagem de garantir o desempenho do circuito. Sua desvantagem é a de não permitir qualquer tipo de modificação ou personalização já que estes *cores* possuem um *layout* pré definido. Este tipo de *core* é o que apresenta o melhor nível de proteção da propriedade intelectual, pois o *core* é uma descrição do *layout*, o que o torna praticamente impossível de reproduzir.
- ***Firm cores***. São um misto de código fonte e *netlist* gerado para a tecnologia empregada (que muda de fabricante para fabricante). Este tipo de *core* permite que a parte visível do código fonte seja modificada pelo projetista de *hardware*, podendo assim, ser adaptada ao projeto. Este tipo de *core* é o mais difundido hoje, pois representa um bom compromisso entre proteção da propriedade intelectual e desempenho.
- ***Soft cores***. São descritos com o emprego de linguagens de descrição de *hardware*, como VHDL ou Verilog, oferecendo máxima flexibilidade e independência de tecnologia. Os *soft cores*

apresentam baixa proteção da propriedade intelectual por serem uma descrição aberta, além de raramente se poder garantir seus parâmetros de desempenho, pois o *core* não teve a síntese realizada, ou pelo menos não teve a síntese realizada no exato ambiente do usuário final.

3.2 Desenvolvimento de um *soft core* PCI

Esta Seção apresenta um fluxo de projeto para *cores*. Esta metodologia vem de encontro a permitir que sejam alcançados os benefícios que as arquiteturas reconfiguráveis propiciam na implementação de *cores*.

O sucesso da implementação de um *core* PCI em arquiteturas reconfiguráveis requer mais do que um dispositivo compatível para este fim. Uma boa metodologia de projeto é igualmente importante para garantir o sucesso deste. As restrições de temporização (*timing*) têm sido e continuarão a ser o maior desafio, ainda mais com a indústria desenvolvendo produtos PCI para barramento de 64 *bits* e *clock* a 66 MHz e freqüências ainda mais altas.

Um fluxo de projeto de um sistema digital mediante o emprego de *cores* consiste de 3 estágios fundamentais [7]:

- **Captura do Projeto**. A Captura do Projeto consiste no projeto e síntese do nível mais alto de hierarquia do sistema, no projeto e síntese da aplicação do usuário e na integração deste com o *core*. A simulação funcional, que é parte da Verificação do Projeto, é executada durante este estágio. O *core* é tratado, nesta etapa, como uma caixa preta e não será um elemento sintetizável. Um modelo funcional do *core* é provido para que a simulação funcional possa ser feita.

- **Implementação do Projeto**. O segundo estágio é o de Implementação do Projeto. Durante este estágio, tarefas de síntese e tradução são executadas. Nesta etapa é importante inserir restrições de *timing* referentes aos atrasos dos caminhos críticos, de forma a guiar o posicionamento e o roteamento. Uma vez obtido o *netlist* da etapa de Captura do Projeto e o *netlist* do *core* unidos, executa-se o posicionamento e o roteamento do sistema, obtendo-se a descrição física do mesmo.

- **Verificação do Projeto**. O estágio final do fluxo de projeto é a Verificação do Projeto. Este estágio consiste em duas etapas principais: Verificação e Simulação. Na Verificação, a análise de *timing* estático determina se o projeto alcança o desempenho desejado. A Simulação verifica o *timing* e a funcionalidade do sistema.

A análise de *timing* estático é executada para um levantamento de caminhos críticos que são restringidos pelos parâmetros de desempenho do projeto. Por exemplo, nas ferramentas Xilinx, a análise de *timing* estático de um projeto em FPGA pode ser feita com o uso de restrições

TIMESPEC [35].

A simulação é executada durante a Captura do Projeto, para verificar a funcionalidade com atrasos unitários (ou nulos), e após o posicionamento e roteamento para verificar a temporização de uma forma detalhada e mais precisa. Estas etapas são, normalmente, adotadas em qualquer projeto envolvendo dispositivos reconfiguráveis.

3.2.1 Especificações Temporais

No caso de um componente PCI operar em 33 MHz, o período de *clock* é de 30 ns, e pode variar ao longo da operação (a especificação permite aos componentes PCI operarem em qualquer freqüência entre 0 e 33 MHz, ou 0 e 66 MHz). Durante os 30 ns de período do *clock*, a especificação permite 10 ns de propagação do sinal. Adicionado a esta propagação, também permite um *clock skew* (escorregamento de *clock*) de 2 ns entre dois componentes PCI. Isto significa que pode haver uma perda de até 40% do tempo de ciclo resultante da distribuição do sinal.

Os 18 ns restantes são divididos entre duas restrições: *clock-to-out* e *setup*. O valor de T_{ckq}, que representa o valor máximo de *clock-to-out*, deve ser de 11 ns. Isto significa que dados válidos devem estar disponíveis no pino de saída do dispositivo de acesso ao barramento PCI no máximo 11 ns depois de receber o sinal de *clock*. O valor T_{su}, que representa o valor máximo do tempo de *setup*, deve ser de 7 ns, isto é, dados válidos devem estar presentes pelo menos 7 ns antes da borda do *clock* [24].

A Figura 16 ilustra as restrições temporais da especificação de 33 MHz [24].

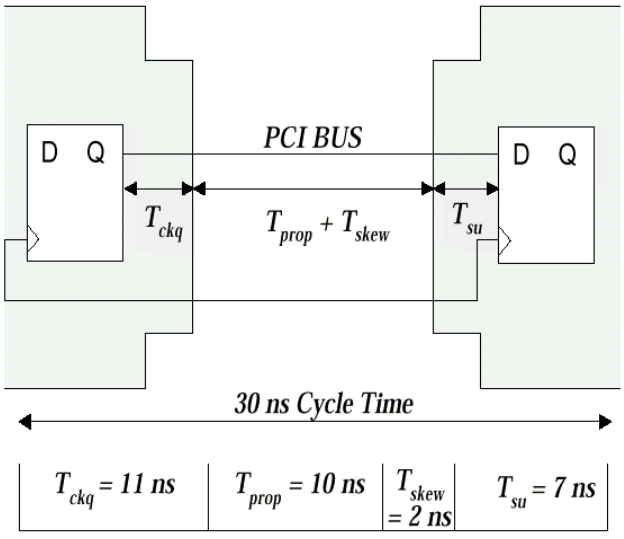

Figura 16 - Especificação temporal para a freqüência de operação de 33 MHz.

A especificação PCI exige uma distribuição dos pinos do componente que implemente a *interface* PCI. O comprimento do condutor do sinal desde o conector até o componente deve ser menor que 1.5" (aproximadamente 3,8 cm), com exceção do sinal de *clock* que pode ser um pouco maior. Todos estes requisitos são especificados para que os efeitos nas linhas de transmissão sejam controláveis, mas dificultam o projeto do circuito impresso e do projeto lógico no FPGA [15].

Deve-se ter o cuidado de, no momento de implementar o *core* no FPGA, especificar a posição relativa deste na periferia do circuito, junto aos pinos de entrada/saída do *core*, de forma a minimizar o comprimento das linhas de conexão, e assim atender aos requisitos temporais.

A Figura 17 apresenta o *layout* recomendado para a implementação de um *core* PCI em um FPGA.

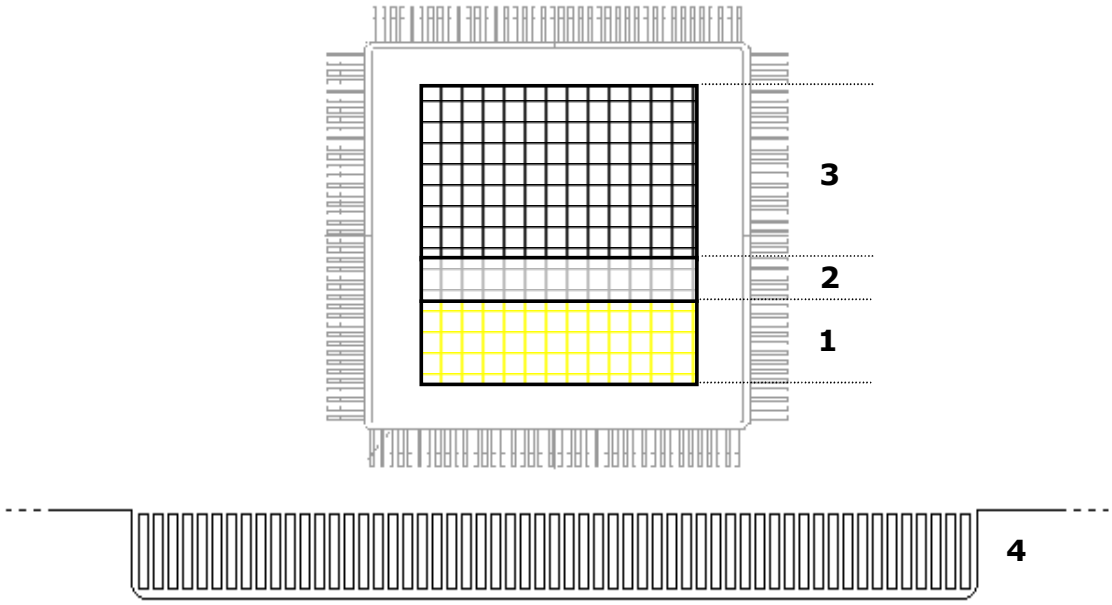

Figura 17 - Distribuição recomendada dos blocos: (1) *core* PCI, (2) interface core/aplicação do usuário e (3) aplicação do usuário. Devido a alta freqüência de operação, o FPGA deve ficar próximo ao conector (4).

3.3 Exemplos de cores PCI

Esta Seção apresenta quatro exemplos de *cores* PCI disponíveis comercialmente.

3.3.1 LogiCORE™ PCI XILINX

A Figura 18 apresenta a estrutura em diagrama de blocos do core PCI desenvolvido pela empresa Xilinx [31] (LogiCORE™ PCI XILINX). Este *core* pode operar tanto como *master* ou *target*.

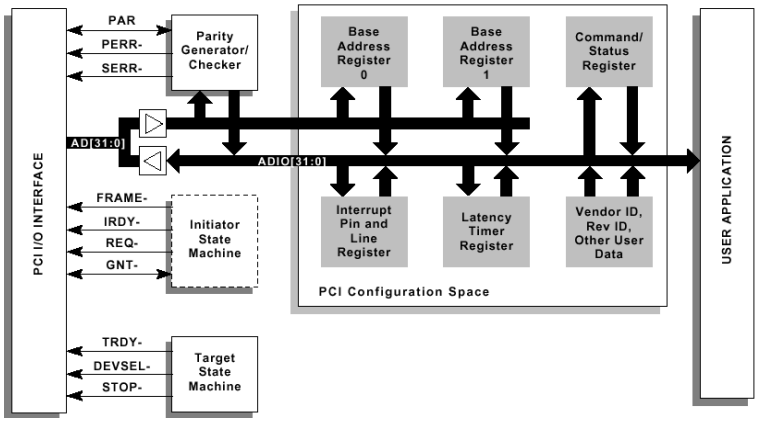

Figura 18 - Diagrama de blocos do core PCI Xilinx com barramento de dados/endereço de 32 bits.

O LogiCORE™ PCI é pré-implementado e testado em dispositivos FPGAs XilinxXC4000XLT. A pinagem e distribuição interna dos CLBs são pré-definidas. Caminhos críticos são controlados para garantir que os critérios de *timing* sejam sempre alcançados. Isto reduz significativamente o tempo na implementação do bloco PCI do projeto. Como resultado reduz significativamente o tempo global do projeto.

Este LogiCORE™ é dividido em cinco blocos (conforme ilustrado na Figura 18) que são:

- ***PCI I/O Interface Block***. Este bloco provê a conexão física do *core* com o barramento PCI, incluindo todos os sinais, sincronização entre entrada/saída e controle de saídas em alta impedância.
- ***Parity Generator/Checker.*** Gera e verifica a paridade entre o barramento AD (*address*/*data*), entre C/BE# (*command*/*byte enable*) e o sinal PAR. Sinaliza erros de paridade via PERR# (*parity error*) e erros de endereçamento via SERR# (*system error*). A paridade em todas as transações de barramento deve ser par.
- ***Target State Machine* (Máquina de Estados do Target)**. Este bloco gerencia e controla as funções de um dispositivo *target*.
- ***Initiator State Machine***. Gerencia e controla o barramento PCI nas funções de inicialização.
- ***PCI Configuration Space***. Este bloco provê os primeiros 64 bytes do *Configuration Space Header* (CSH) para suportar a inicialização PNP. Este espaço inclui os registradores *Command*, *Status* e 2 *Base Address Register* (BARs). Os BARs são responsáveis pela implementação de espaços de endereçamento de memória ou de entrada/saída (I/O). Cada BAR define o endereço base para a interface e permite ao *software* do sistema determinar a faixa de valores de endereços disponível à interface.

A Tabela 6 apresenta as características do LogiCORE™ e a quantidade de pinos e CLBs restantes para diferentes versões dos FPGAs da família 4000 e a Tabela 7 apresenta as

características da implementação do *core* para a família Virtex [33].

Tabela 6 - Opções de dispositivos Xilinx da família 4000 e recursos utilizados.

Especificação do Core		
Família de dispositivos	XC4000XLT	
CLBs utilizados	178 - 308	
IOBs utilizados	53/51	
Freqüência de operação	0 - 33 MHz	
Características utilizadas dos dispositivos	Barramentos de dados bi-direcionais	
	Select RAM	
	Boundary Scan (opcional)	
Dispositivos compatíveis	**Recursos restantes**	
	I/O	**CLB**
XC4013XLT PQ208	99/101	268-398
XC4013XLT PQ240	133/135	268-398
XC4028XLT HQ240	133/135	716-846
XC4062XLT HQ240	133/135	1996-2126
XC4062XLT BG432	293/295	1996-2126

Tabela 7 – Implementação do LogiCORE em dispositivos Xilinx da família Virtex.

Especificação do Core		
Família de dispositivos	Virtex	
Slices utilizados	381-403	
IOBs utilizados	88	
Freqüência de trabalho	0 – 66 MHz	
Características utilizadas dos dispositivos	Barramentos de dados bi-direcionais	
	Select IO	
	Block Select RAM+	
	Boundary Scan (opcional)	
Dispositivos compatíveis	**Percentual de recursos utilizados**	
	I/O	**Slices**
XCV300 – 5/6 BG432	28%	12%
XCV1000 – 5/6 FG680	17%	3%

3.3.2 Core PCI Altera

A empresa Altera (www.altera.com) produz IP *cores* que implementam uma interface PCI, dentre eles o pci_mt32 (PCI Master/Target 32 Bits). Empregando as famílias de dispositivos Altera® APEX 20K, FLEX 10K, e FLEX 8000, o pci_mt32 suporta transações de configuração, de I/O e de memória. Devido à alta densidade dos dispositivos FPGA da Altera empregados, os projetistas contam com muitos recursos de espaço após a implementação lógica da interface PCI. O pci_mt32 pode operar com freqüências de *clock* de 33 MHz ou 60 MHz, alcançando a vazão de dados de 132

MB/s em 32 bits/33 MHz, até 528 MB/s em 64 bits/66 MHz.

A Figura 19 mostra o diagrama funcional do pci_mt32 da Altera, e a Tabela 8 apresenta as características dos diversos tipos de implementações de *cores* PCI com dispositivos ALTERA.

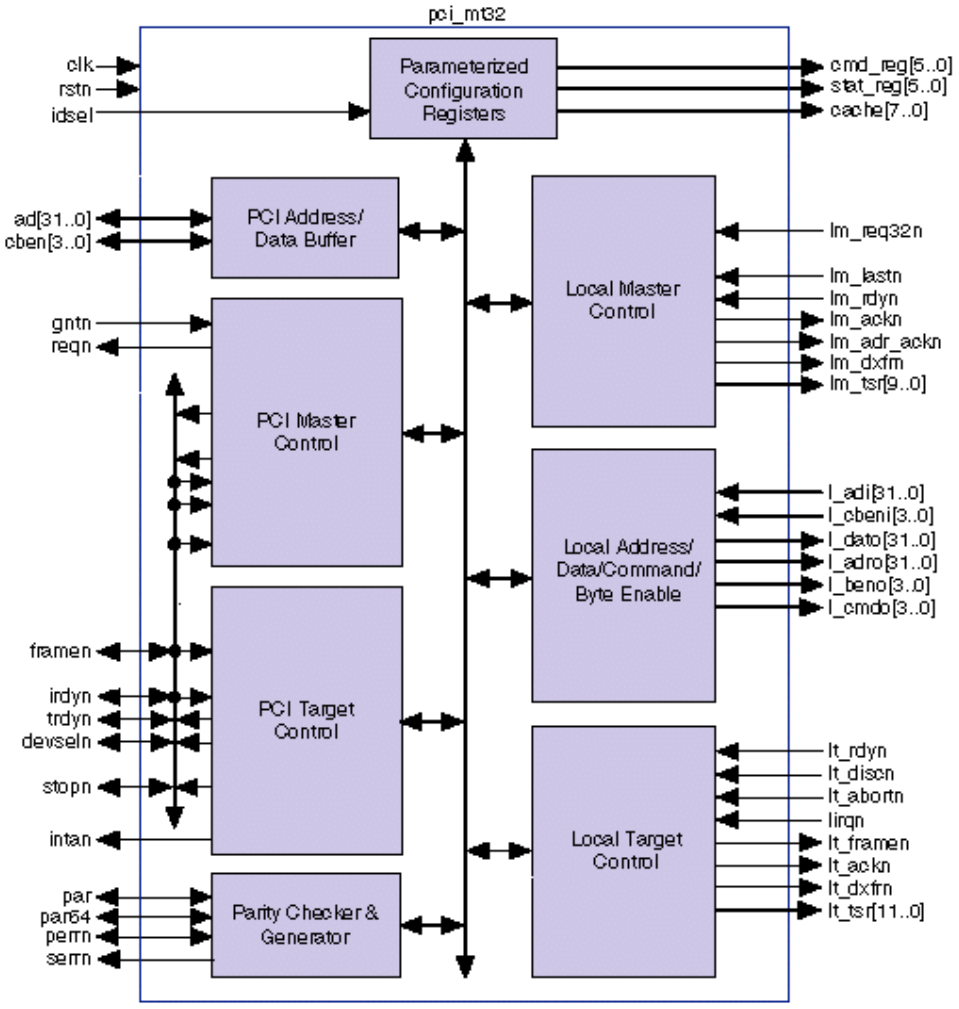

Figura 19 – Diagrama de blocos do pci_mt32 Altera.

Tabela 8 – Características e funções dos *cores* PCI Altera.

Característica	PCI/MT64	PCI/MT32	PCI/T64	PCI/T32
Largura do barramento [bits]	64	32	64	32
Freqüência de operação [MHz]	0-60*	0-60*	0-60*	0-60*
Função	*Master/Target*	*Master/Target*	*Target*	*Target*
Dispositivos	Flex 10KE, Apex 20KE, Acex 1K	Flex 10KE, Flex 6000, Apex 20KE, Acex 1K	Flex 10KE, Apex 20KE, Acex 1K	Flex 10KE, Flex 6000, Apex 20KE, Acex 1K
Elementos lógicos	1300	900	800	500

* A freqüência é de 60 MHz, conforme **www.altera.com/products/ip/altera/t-alt-pci_soln.html**.

A empresa Altera permite ao projetista testar um modelo funcional de seus *cores* através de um "*test-drive"*, ou seja, uma pré-avaliação do *core* antes de adquiri-lo.

3.3.3 Core PCI inSilicon

Os *cores* desenvolvidos pela inSilicon (www.insilicon.com) são módulos RTL sintetizáveis que possibilitam a interface entre a aplicação do usuário e o barramento PCI. A família de *cores* PCI inSilicon foram desenvolvidas para operar em 33 MHz e 66 MHz (32/64 *bits*). Estas freqüências de operação são alcançadas através da combinação da síntese lógica, planta baixa, posicionamento, interconexão e verificação *post layout*.

A Figura 20 apresenta, em forma de diagrama de blocos, o *core* inSilicon.

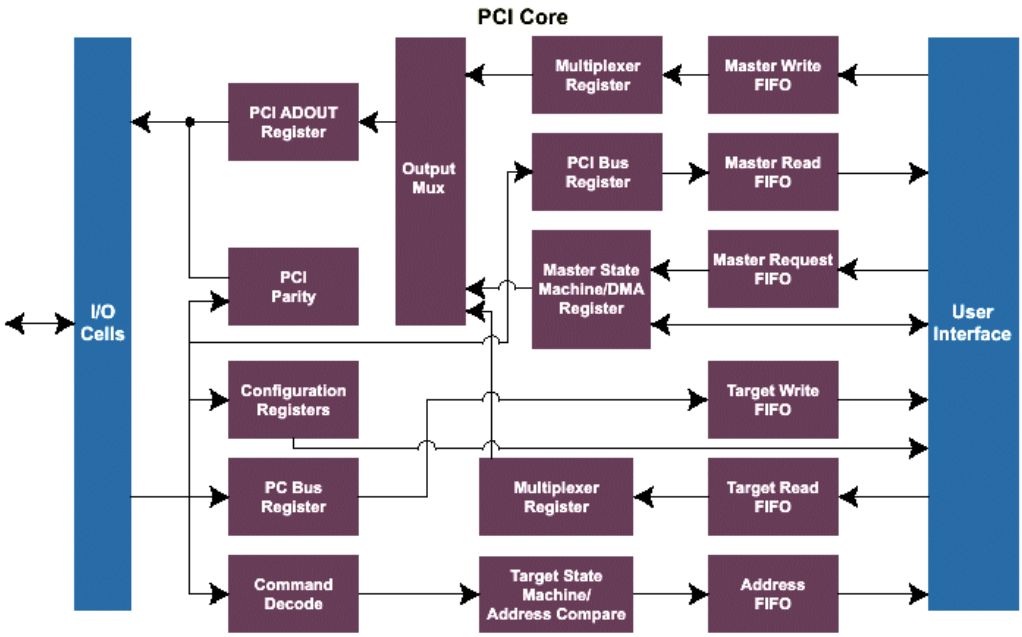

Figura 20 – Diagrama de blocos do *core* PCI inSilicon.

3.3.4 Core PCI PLD Applications

A Powerful Logic Design - PLD Applications (www.plda.com) produz um *core* PCI *master*/*target* para barramentos de 32/64-bit, 33/66-MHz. Este *core* prevê também uma interface entre o barramento PCI e o projeto da aplicação final do usuário (*back-end*).

Este produto traz consigo um conjunto de projetos para aplicações de usuário em VHDL, que o projetista poderá utilizar ou personalizar se assim necessitar. As referências destes projetos incluem interfaces que utilizam a família FLEX 10K da Altera, empregando seus *embedded array blocks* (EABs) como *buffer* de SRAM ou como *buffer* FIFO. A interface do *core* PCI *master*/*target* da PLD Applications é totalmente parametrizável.

A Figura 21 apresenta um diagrama de blocos da interface PLD 32/64-bit PCI *master*/*target*, e a Tabela 9 traz dados referentes à utilização dos recursos dos FPGAs.

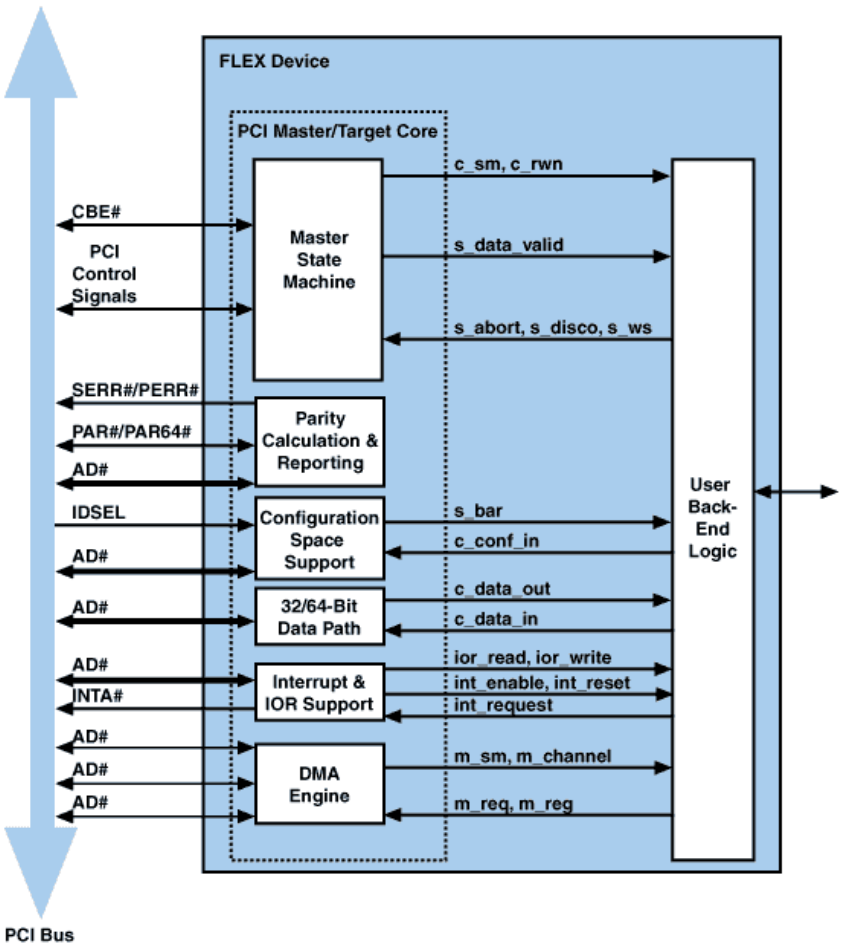

Figura 21 – Bloco diagrama do core PCI da PLD implementado em FPGA Altera.

Tabela 9 – Resultados da ocupação dos recursos disponíveis nos FPGAs.

Dispositivo	Células Lógicas	Desempenho [MHz]
EPF10K30A	1,150	>33
EPF10K30E	1,150	>33
EPF10K50	1,150	>33
EPF10K50V	1,150	>33
EPF10K100B	1,150	>33
EPF10K130E	1,150	>33
EPF10K200E	1,150	>33
EPF6016	1.150	>33 (Nota 1)
EPF6024A	1.150	>33 (Nota 2)

Notas: (1) Compatibilidade com 66 MHz somente com dispositivos da linha FLEX 10KE-1.
(2) Mesmo desempenho no modo 64 *bits*.

3.4 Core PCI desenvolvido

Esta Seção apresenta a contribuição maior desta dissertação, que é o projeto de um módulo de *hardware* reutilizável, *soft core*, para a implementação do padrão PCI [6]. A implementação deste *core* é feita utilizando-se a linguagem de descrição de *hardware* VHDL, validando-o através de simulação funcional. A simulação testa os ciclos básicos de leitura e escrita, tanto em modo simples quanto em rajada. A etapa seguinte deste trabalho é a validação do *core* em um ambiente de prototipação composto de FPGA e barramento PCI.

O *core* desenvolvido é compatível com a especificação PCI para um dispositivo funcionar como *target*, para uma freqüência de 33MHz e largura de barramento de 32 *bits*. Optou-se por um dispositivo *target* com estas características devido ao fato de ser o que apresenta menor complexidade de operação e implementação no PCI.

3.4.1 Diagrama de Blocos do Core PCI desenvolvido

A Figura 22 apresenta, em forma de diagrama de blocos genérico, a composição básica de um *core* PCI, representado pelos conteúdos do retângulo pontilhado. Esta estrutura é a adotada pelo Core PCI desenvolvido.

Descrição dos módulos do Core PCI desenvolvido:

- **Máquina de Estados de Controle.**
  Um dispositivo PCI compatível pode, se assim for projetado, funcionar como *master* (assume o controle da transação do barramento: leitura, escrita ou acesso às configurações) ou como *target* (dispositivo alvo de alguma transação de barramento), ou até mesmo com as duas funções. Se o dispositivo for sempre passivo, apenas a máquina de estados *target* necessita ser implementada.

- **Gerador e Verificador de Paridade**.
  Faz a geração e verificação da paridade sobre o barramento de endereços e dados, linhas de comando e os sinais de paridade gerados ao longo das transferências de dados em uma transação de barramento. Em caso de erros de paridade este bloco os sinaliza.

- **Registradores do Espaço de Configuração**.
  Este bloco provê os registradores do *Configuration Space Header* (CSH). Neste espaço, estão informações necessárias para a inicialização e configuração *plug-and-play*. Fazem parte destas informações comandos, *status* e os registradores de endereços base (*Base Address Register* - BAR). O BAR indica como implementar espaços de memória ou espaços de endereçamento de I/O no *host*. Também é função deste módulo realizar a comparação dos endereços de dispositivo com o endereço colocado no barramento de endereços.

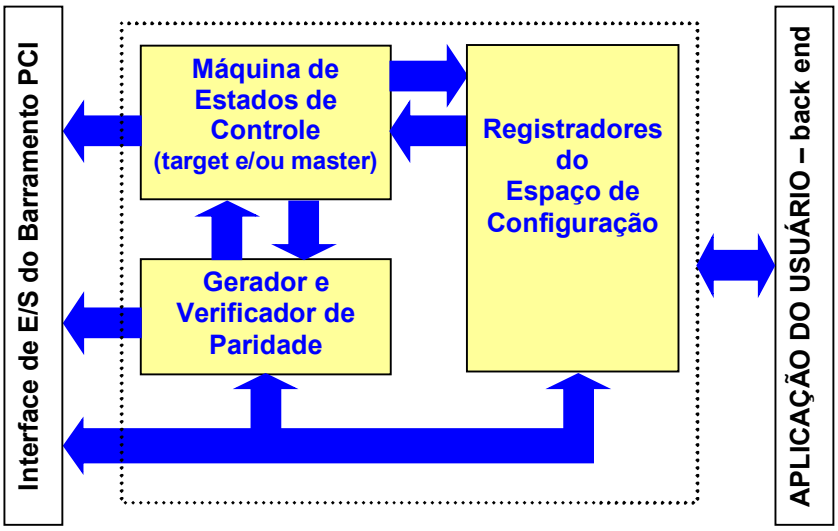

Figura 22 - Diagrama de blocos de uma interface PCI.

3.4.2 Máquina de Estados de Controle

A máquina de estados de controle do *core* operando como *target* está ilustrada na Figura 23. Esta máquina tem por função gerar os sinas de controle para ciclos de leitura, escrita e espera. Descreve-se abaixo os estados representados na FSM da Figura 23.

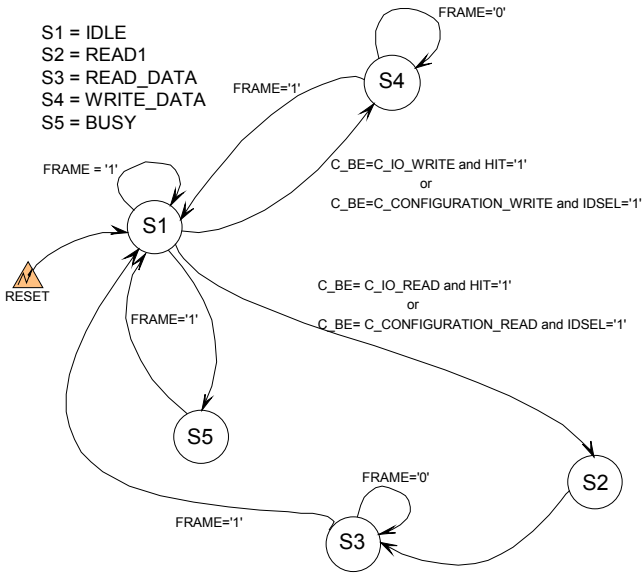

Figura 23 - Máquina de estados de um dispositivo PCI *target* compatível.

- **IDLE** (S1)
  O *target* captura os sinais do barramento para determinar quando o *master* está requerendo um acesso à configuração ou entrada/saída de dados. Este é o estado inicial. Quando uma transação

inicia, o endereço presente no barramento é comparado com o espaço de endereços (contidos nos BAR) do dispositivo PCI. Se este endereço estiver contido neste espaço (HIT='1'), o *target* vai para o estado *WRITE_DATA* (S4) se o comando enviado pelo *master* for de escrita (C_BE=C_IO_READ and HIT='1' or C_BE=C_CONFIGURATION_WRITE and IDSEL='1') ou para *READ1* (S2) se o comando for de leitura (C_BE=C_IO_READ and HIT='1' or C_BE=C_CONFIGURATION_READ and IDSEL='1'). Se o endereço não for igual ao do dispositivo vai para o estado *BUSY* (S5).

- **READ1** (S2)
  Este estado é necessário para que seja feita uma inversão do barramento de dados e endereços.

- **WRITE_DATA** (S4)
  Neste estado o *master* envia dados para o *target* (escreve). Enquanto o sinal de *FRAME* estiver ativo (nível baixo) o *target* é mantido no estado de escrita, caracterizando uma transferência de dados em modo *burst*. Ao término do ciclo de escrita o dispositivo vai para o estado *IDLE*.

- **READ_DATA** (S3)
  Neste estado o *master* solicita dados para o *target* (lê). Enquanto o sinal de *FRAME* estiver ativo (nível baixo) o *target* é mantido no estado de leitura, caracterizando uma transferência de dados em modo *burst*. Deste estado, o *target* automaticamente retorna ao estado *IDLE*.

- **BUSY** (S5)
  Enquanto uma transação está acontecendo (*FRAME* ativo), o dispositivo que não foi selecionado pelo *master* mantém-se neste estado. Quando a transação se completar (*FRAME* desativado), o *target* retorna ao estado *IDLE* para aguardar por uma fase de endereçamento da próxima transação.

O código em VHDL, referente à máquina de estados do *target* é o seguinte:

```
------------------------------------------------------------------
-- MÁQUINA DE ESTADOS DE CONTROLE --
------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.PCI_PACK.all;

entity TARGET is
    port(  AD:              in STD_LOGIC_VECTOR (31 downto 0);
           C_BE:            in STD_LOGIC_VECTOR (3 downto 0);
           INT_ADDR:        in STD_LOGIC_VECTOR (31 downto 0);
           CLK:             in STD_LOGIC;
           FRAME:           in STD_LOGIC;
           IDSEL:           in STD_LOGIC;
           IRDY:            in STD_LOGIC;
           RST:             in STD_LOGIC;
           HIT:             in STD_LOGIC;
           ESTADO:          out FSM_TARGET_TYPE
           );
end;

architecture TARGET_arch of TARGET is

    signal FSM_T,FSM_TARGET:      FSM_TARGET_TYPE;
```

```
    signal READ, WRITE:                 std_logic;

    begin

    -- sincroniza o estado da máquina com a borda de descida
    ESTADO <= FSM_T;
    process(CLK)
            begin
            if(CLK'event and CLK='0') then
                    FSM_T <= FSM_TARGET;
            end if;
    end process;

    -- o controle é feito na borda de subida, quando os dados e controle estão estáveis
    FSM_TARGET_MACHINE: process (CLK, RST)
            begin
            if RST='0' then
                    FSM_TARGET <= IDLE;
                    READ <= '0';
                    WRITE<= '0';

            elsif CLK'event and CLK = '1' then

                    case FSM_TARGET is

                            -- IDLE: standby
                            when IDLE =>
                            if FRAME = '1' then
                                    FSM_TARGET <= IDLE;
                            else
                                if (C_BE=C_IO_WRITE and HIT='1') or (C_BE = C_CONFIGURATION_WRITE and IDSEL ='1')
                                        then FSM_TARGET <= WRITE_DATA;
                                    elsif (C_BE=C_IO_READ and HIT='1') or (C_BE = C_CONFIGURATION_READ and IDSEL
                                        ='1') then
                                        FSM_TARGET <= READ1;
                                    else
                                        FSM_TARGET <= BUSY;
                                    end if;
                            end if;

                            -- estado para inversão do barramento AD
                            when READ1 =>
                            FSM_TARGET <= READ_DATA;

                            -- o target coloca dados no barramento AD
                            when READ_DATA =>
                            if FRAME='0' then
                                    FSM_TARGET <= READ_DATA;
                            elsif FRAME='1' then
                                    FSM_TARGET <= IDLE;
                            end if;

                            -- o target lê os dados que o master coloca em AD
                            when WRITE_DATA =>
                            if FRAME='0' then
                                    FSM_TARGET <= WRITE_DATA;
                            elsif FRAME='1' then
                                    FSM_TARGET <= IDLE;
                            end if;

                            -- o barramento está sendo acessado por outro dispositivo
                            when BUSY =>
                            if FRAME='0' then
                                    FSM_TARGET <= BUSY;
                            elsif FRAME='1' then
                                    FSM_TARGET <= IDLE;
                            end if;

                            when others => null;

                    end case;
            end if;
    end process;
end TARGET_arch;
```

A Figura 24 mostra os sinais pertinentes à troca do estado IDLE (S1) para o de escrita WRITE_DATA (S4).

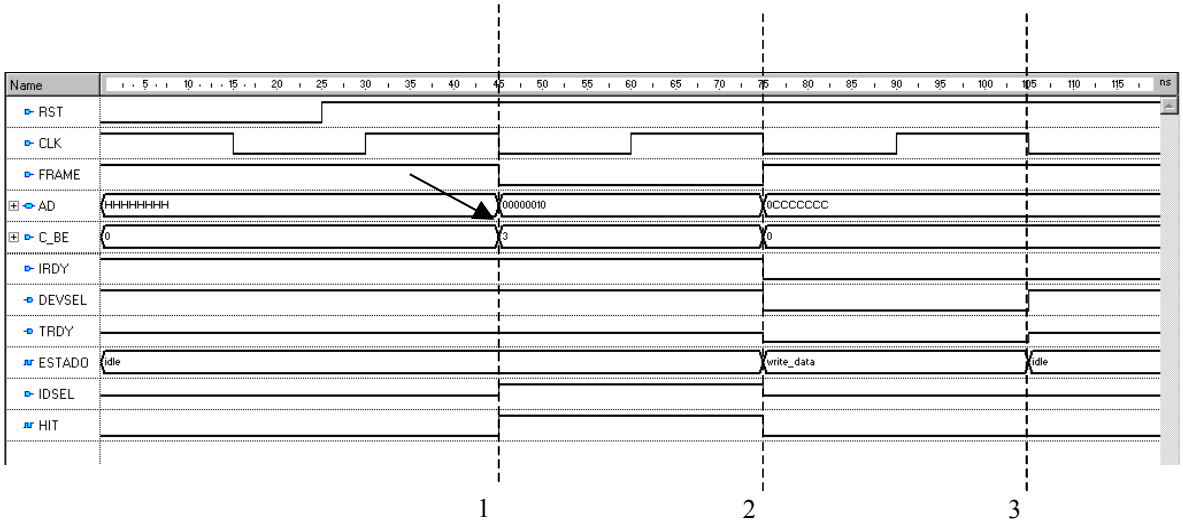

Figura 24 – Sinais da passagem do estado S1 para o estado S4.

A informação “3” na linha C/BE (C_IO_WRITE) (1), associada à seleção do dispositivo *target* com endereço “00000010” (HIT=1), gera a troca de estado (2). Após o dado ser recebido pelo *target*, a máquina de estados volta para S1 (3).

O barramento de dados/endereço quando não é acionado nem pelo *target* nem pelo *master* fica em nível lógico ‘1’ devido à um resistor de *pull-up*, ou seja, ‘1’ fraco. Este nível é representado por ‘H’ na simulação.

A Figura 25 mostra os sinais pertinentes à troca do estado IDLE (S1) para o de leitura READ_DATA (S3).

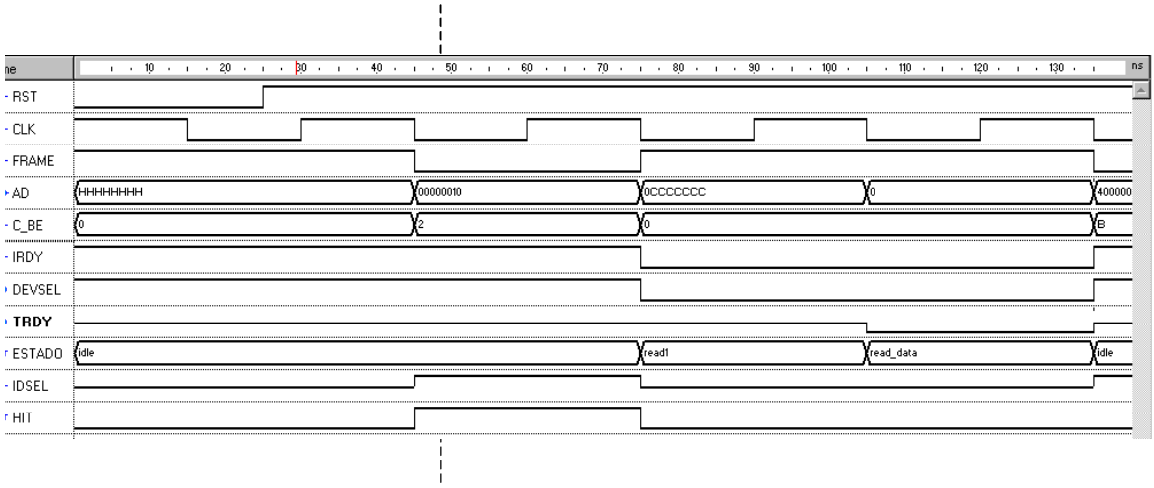

Figura 25 - Sinais da passagem do estado S1 para o estado S3.

A informação “2” na linha C/BE (C_IO_READ) (1), associada à seleção do dispositivo *target* com endereço “00000010” (HIT=1), gera, inicialmente, a troca de estado de S1 para o S2 (READ_1) (necessário para a inversão da direção dos dados no barramento AD) (2). Na próxima descida do sinal de *clock*, a máquina de estados chega a S3 (3). Após o dado ser recebido pelo *master*, a máquina de estados volta para S1 (4).

A Figura 26 mostra os sinais pertinentes à troca do estado IDLE (S1) para o BUSY (S5).

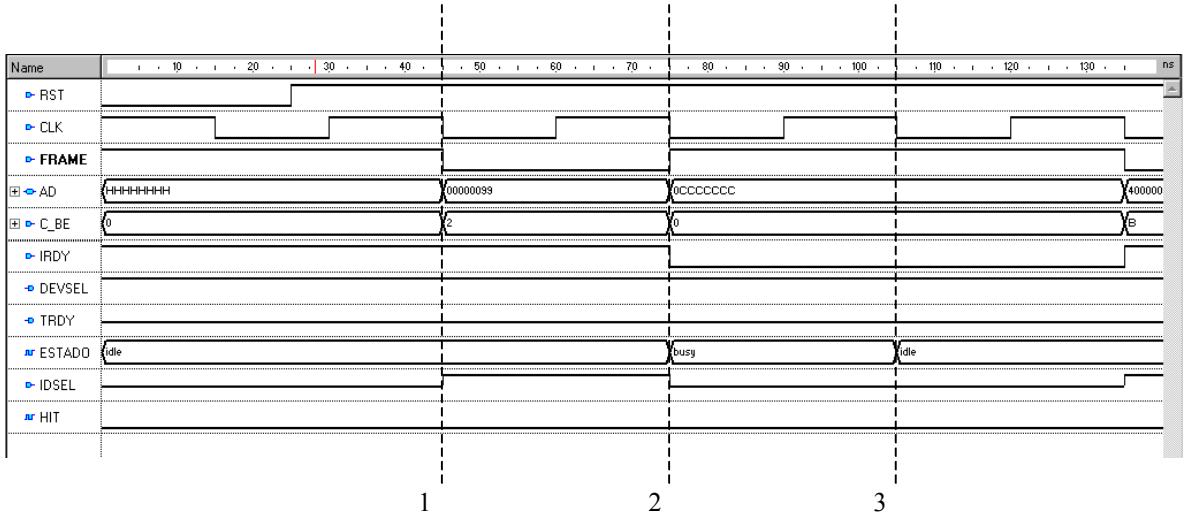

Figura 26 – Sinais que levam a máquina de estados ao estado BUSY (S5).

Como não houve um casamento entre o endereço colocado pelo *master* em AD (1), com o endereço interno do dispositivo *target*, o sinal HIT não foi ativado (HIT=0). Sendo assim, a máquina de estados vai para o estado BUSY (2) até que a transação de barramento se encerre (3).

3.4.3 Bloco gerador/verificador de paridade

Em toda transação de barramento PCI há geração e verificação de paridade entre as linhas AD e C/BE, sendo que a paridade deve ser par. O *bit* de paridade é sinalizado pelo sinal PAR. A primeira fase de uma transação de barramento chama-se de fase de endereçamento, na qual o *master* coloca um endereço nas linhas AD e um comando nas linhas C/BE. Um ciclo de *clock* depois, o *master* coloca na linha PAR o *bit* de paridade correspondente, calculado entre AD e C/BE. Este atraso de um ciclo de *clock* deve-se ao fato de que o *target* precisa calcular a paridade entre AD e C/BE recebidos e depois receber o sinal PAR para poder comparar com a paridade que calculou. Se o bit recebido na linha PAR for igual ao da paridade calculada pelo *target*, a linha PERR# (*parity error*) é mantida desativada, isto é, em nível lógico 1. A segunda fase de uma transação de barramento é a fase de dados. A partir desta fase, a paridade é calculada entre os dados colocados em AD e os sinais de *byte enable* colocados em C/BE. Para cada dado e *byte enable*, a paridade é calculada e um sinal PAR é gerado, seja em ciclos de leitura ou escrita.

O código em VHDL, referente à máquina de estados do bloco gerador/verificador de paridade é o seguinte:

```
--------------------------------------------------------------------------
--  BLOCO GERADOR/VERIFICADOR DE PARIDADE  --
--------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.PCI_PACK.all;

entity paridade is
    port (
        CLK:            in STD_LOGIC;
        AD:             in STD_LOGIC_VECTOR (31 downto 0);
        PAR:            inout STD_LOGIC;
        PERR:           inout STD_LOGIC;
        SERR:           inout STD_LOGIC;
        C_BE:           in STD_LOGIC_VECTOR ( 3 downto 0);
        ESTADO:         in FSM_TARGET_TYPE;
        FRAME:          in STD_LOGIC;
        IRDY:           in STD_LOGIC;
        TRDY:           in STD_LOGIC
        );
end;


architecture paridade of PARIDADE is
    signal par16:           STD_LOGIC_VECTOR(15 downto 0);
    signal par8:            STD_LOGIC_VECTOR( 7 downto 0);
    signal par4:            STD_LOGIC_VECTOR( 3 downto 0);
    signal par0:            STD_LOGIC_VECTOR( 3 downto 0);
    signal parout:          STD_LOGIC;

    signal par_interno:     STD_LOGIC;      -- saida do FF (para atrasar um ciclo de clk)

    signal eant:            FSM_TARGET_TYPE;
    signal TRDYANT:std_logic;
    signal IRDYANT, IT: std_logic;
    signal tserr,tperr: std_logic;

    begin

    --
    --  calcula a paridade dos 36 bits, AD e CBE
    --
    par16       <= AD(31 downto 16) xor AD(15 downto 0);        -- xor 16x16, gera 16 bits
    par8        <=par16(15 downto 8) xor par16(7 downto 0);     -- xor  8x8 , gera 8 bits
    par4        <=par8(7 downto 4) xor par8(3 downto 0);        -- xor  4x4 , gera 4 bits
    par0        <= par4 xor C_BE;                               -- xor  4x4 , gera 4 bits
    parout      <= par0(3)xor par0(2) xor par0(1) xor par0(0);  -- xor final dos 4 gerados...

    --
    --  armazena o IRDY valido
    --
    process(CLK)
        begin
        if(CLK'event and CLK='1') then
            IT<=IRDY;
        end if;
    end process;

    --
    --  atrasa os sinais de controle 1 ciclo de clock, sincronizando-os na borda de descida
    --
    process(CLK)
        begin
        if (CLK'event and CLK='0') then
            eant <= ESTADO;
            TRDYANT<=TRDY;
            IRDYANT<=IT;
            par_interno <= parout;      -- sincroniza na borda de descida a paridade dos 36 bits

            PERR <= tperr;              -- sincroniza os sinais de erro de paridade
            SERR <= tserr;

        end if;
```

```
    end process;

    -- *quando o ciclo for de leitura, o target coloca a paridade calculada na linha PAR,
    --  em outra situação coloca a linha par em alta impedância
    -- *quando o ciclo for de escrita, o master coloca a paridade calculada na linha PAR,
    --  em outra situção coloca a linha em alta impedância

    PAR <= par_interno when (eant = READ_DATA and IRDYANT='0' and TRDYANT='0') else 'Z';

    -- o erro de paridade é detectado na borda de subida, para posteriormente ser
    -- enviado na borda de descida - ver registrador acima

    process(CLK)
        begin
        if (CLK'event and CLK='1') then

                -- se o par_interno antigo for diferente da pariade enviada, então houve erro
                if(par_interno/=PAR and eant=WRITE_DATA) then
                        tperr <= '0';
                else tperr <= 'Z';
                end if;

                if(par_interno/=PAR and (ESTADO=READ1 or (ESTADO=WRITE_DATA and eant=IDLE))) then

                        tserr <= '0';
                else tserr <= 'Z';
                end if;

        end if;
    end process;

end paridade;
```

As Figuras seguintes apresentam os sinais de duas simulações nas quais se verificam duas situações diferentes com relação ao cálculo de paridade e a sinalização de um eventual erro de paridade sinalizado pelo sinal PERR# (*parity-error*).

A Figura 27 mostra um ciclo de leitura onde, em (1), está presente o valor do *bit* de paridade, determinado pelo cálculo da paridade entre as informações de endereço e comando (A&C) colocados pelo *master* no ciclo anterior (fase de endereçamento). O valor do *bit* de paridade presente em (2), foi determinado pelo cálculo da paridade entre as informações presentes nas linhas de dados e habilitação de *byte* (D&BE), um ciclo após estas informações terem sido colocadas nestas linhas (fase de dados), também pelo *master*. Observa-se que a linha PERR#, que normalmente se encontra em nível '1 fraco' (devido a um resistor de *pull-up*), em (3) teve seu nível lógico levado à '0', sinalizando um erro de paridade ocorrido entre D&BE.

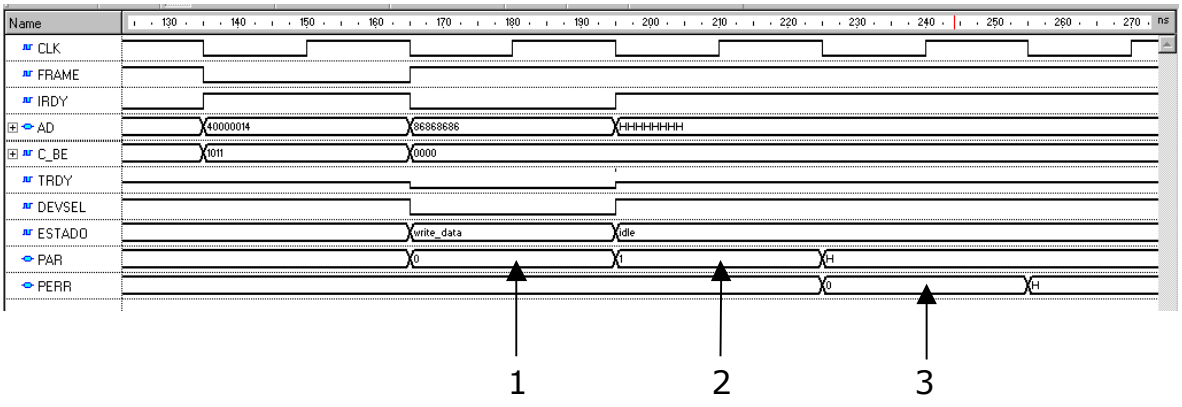

Figura 27 – Exemplo de ciclo de barramento no qual houve um erro de paridade, sinalizado por PERR#.

A Figura 28 mostra um ciclo de escrita onde, em (1), está presente o valor do *bit* de paridade, determinado pelo cálculo da paridade entre as informações de endereço e comando (A&C) colocados pelo *master* no ciclo anterior (fase de endereçamento). O valor do *bit* de paridade presente em (2) (calculado após a inversão do barramento), foi determinado pelo cálculo da paridade entre as informações presentes nas linhas de dados e habilitação de *byte* (D&BE), um ciclo após estas informações terem sido colocadas nestas linhas (fase de dados), também pelo *master*. Observa-se que a linha PERR# (3), que normalmente se encontra em nível '1 fraco' (devido a um resistor de *pull-up*), em nenhum momento teve seu nível lógico levado à '0', caracterizando uma transação de barramento sem erro de paridade.

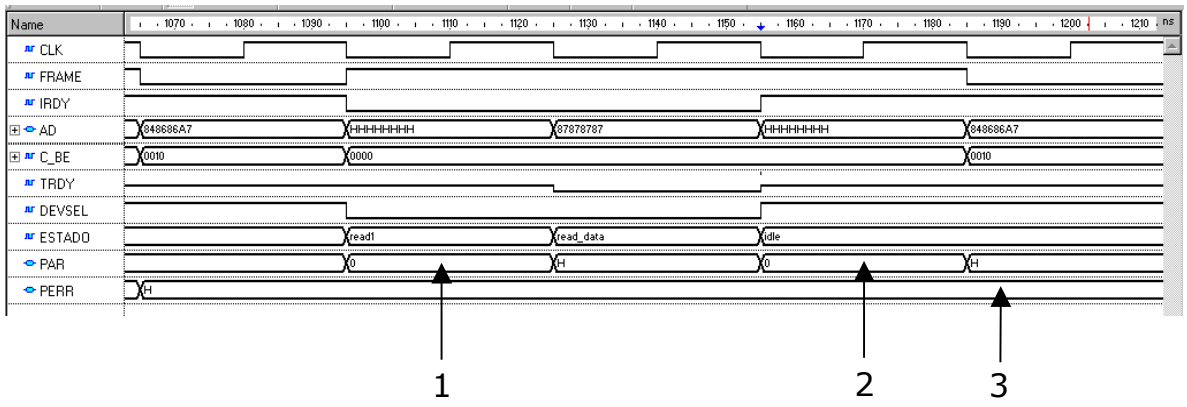

Figura 28 - Exemplo de ciclo de barramento no qual não houve um erro de paridade. O sinal PERR# manteve-se sempre em nível '1 fraco' ('H' na simulação).

3.4.4 Espaço de configuração

Este bloco provê os registradores do *Configuration Space Header* (CSH). A Figura 4, da Página 14, apresenta a estrutura dos primeiros 64 *bytes* dos registradores de configuração.

O PCI reconhece três diferentes espaços de endereços: endereços de memória (*memory address*), endereços de I/O (I/O *address*) e memória de configuração (*configuration memory*). Os dois primeiros, como nos microprocessadores da Intel, são para acessos normais de dados. O terceiro, que fica localizado em todas as placas de extensão (*add-on cards*) e outros componentes conectados ao barramento PCI, contém as configurações do dispositivo ajustadas pela PCI BIOS.

O código em VHDL do banco de registradores que compõe o Espaço de Configuração do Core PCI desenvolvido é o seguinte:

```
-------------------------------------------------
-- BANCO DE REGISTRADORES  --
-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.PCI_PACK.all;
```

```
entity BANCO_REG is
    port (
        RST:        in STD_LOGIC;                          -- reset do sistema
        CLK:        in STD_LOGIC;                          -- clock do sistema
        CBE:        in STD_LOGIC_VECTOR (3 downto 0);      -- command/byte enable
        ESTADO:     in FSM_TARGET_TYPE;                    -- estado da FSM
        AD_IN:      in STD_LOGIC_VECTOR (31 downto 0);     -- endereço de entrada
        AD_OUT:     out STD_LOGIC_VECTOR (31 downto 0);    -- endereço de saída
        HIT:        out STD_LOGIC;                         -- testa se esta no espaço de endereçamento
        END_DEC:    out STD_LOGIC_VECTOR (5 downto 0)-- endereço decodificado do registrador a ser usado
        );
end;

architecture BANCO_REG of BANCO_REG is

signal endereco_conf:           std_logic_vector(5 downto 0);
signal comando:                 std_logic_vector(3 downto 0);
signal BAR0, BAR1:              std_logic_vector(31 downto 0);
signal BAR0BE, BAR1BE:          std_logic_vector(31 downto 0);      -- BAR em formato big-endian
signal IO_space_control:        std_logic;
signal PERR_response:           std_logic;
signal SERR_control:            std_logic;
signal data_parity_status:      std_logic;
signal sign_SERR_status:        std_logic;
signal detected_PERR_status:    std_logic;
signal TAM:                     std_logic_vector (31 downto 0);
signal AD_TEMP:                 std_logic_vector (31 downto 0);
signal X:                       std_logic_vector (31 downto 0);
signal END_DECODIF:             std_logic_vector (5 downto 0);
begin

    -- Espaço de endereçamento que o dispositivo vai solicitar

    TAM <= x"00000080";

    -- converte os bars do formato little-endian (no qual eles devem ser armazenados) para
    -- o formato big-endian para poder comparar a faixa de endereços

    BAR0BE <= BAR0(7 downto 0) & BAR0(15 downto 8) & BAR0(23 downto 16) & BAR0(31 downto 24);
    BAR1BE <= BAR1(7 downto 0) & BAR1(15 downto 8) & BAR1(23 downto 16) & BAR1(31 downto 24);

    -- Testa se o endereço enviado em AD corresponde ao espaço de
    -- endereçamento do dispositivo

    HIT <=      '1' when ( (CBE=C_IO_READ or CBE=C_IO_WRITE) and ((AD_IN>=BAR0BE and AD_IN<=BAR0BE + TAM) or
                    (AD_IN>=BAR1BE and AD_IN<=BAR1BE + TAM))  )
                else '0';

    END_DEC(5)<='1' when (AD_IN>=BAR1BE and AD_IN<=BAR1BE + TAM)   -- se estiver na faixa do BAR1 então
                else '0';                                          -- 1xxxxx, se BAR0 então 0xxxxx

    X <= (AD_IN-BAR0BE) when END_DECODIF(5)='0' else (AD_IN-BAR1BE);
  END_DEC(4 downto 0) <= X(6 downto 2);                            -- os 5 bits (4 a 0) decodificam
                                                                   -- qual o registrador de 32 bits acessar
    process(CLK)
    begin
        if(CLK'event and CLK='1') then
            if(ESTADO=IDLE) then
                -- Guarda o endereço do registrador que será lido ou escrito
                -- Só são necessários os bits 2 a 7
                endereco_conf <= AD_IN(7 downto 2);
                -- Guarda também o comando que será executado
                comando <= CBE;
            end if;
        end if;
    end process;

    process(CLK)
    begin
        if(CLK'event and CLK='1') then
            if(RST='0') then
                -- inicializar registradores aqui
                bar0                    <= x"05000000";
                bar1                    <= x"05000000";
                IO_space_control        <= '1';
                PERR_response           <= '1';
                SERR_control            <= '1';
                data_parity_status      <= '0';
                sign_SERR_status        <= '0';
```

```
                        detected_PERR_status        <= '0';
                else
                        if(comando=C_CONFIGURATION_WRITE) then   -- escrita do espaço de configuração
                                if(ESTADO=WRITE_DATA) then
                                        if(endereco_conf="000100") then
                                                BAR0(31 downto 2) <= AD_IN(31 downto 2);
                                        elsif(endereco_conf="000101") then
                                                BAR1(31 downto 2) <= AD_IN(31 downto 2);
                                        elsif(endereco_conf="000001") then
                                                IO_space_control        <= AD_IN(0);
                                                PERR_response           <= AD_IN(6);
                                                SERR_control            <= AD_IN(8);
                                                data_parity_status      <= AD_IN(24);
                                                sign_SERR_status        <= AD_IN(30);
                                                detected_PERR_status    <= AD_IN(31);
                                        end if;
                                end if;
                        elsif(comando=C_CONFIGURATION_READ) then   -- leitura no espaço de configuração
                                if(ESTADO=READ1) then
                                        if(endereco_conf="000000") then
                                                -- vendor ID (6AFF) e device ID (0001)
                                                AD_TEMP <= x"6AFF0001";
                                        elsif(endereco_conf="000001") then
                                                -- command e status
                                                AD_TEMP <= '0' & PERR_response & "00000" & IO_space_control &
                                                                "0000000" & SERR_control & "00000000" &
                                                                detected_PERR_status &  sign_SERR_status &
                                                                "00000" & data_parity_status;
                                        elsif(endereco_conf="000010") then
                                                -- revision ID (00) e class code (FF0000)
                                                AD_TEMP <= x"00FF0000";
                                        elsif(endereco_conf="001111") then
                                                -- interrupt line, interrupt pin, min_gnt, max_lat
                                                AD_TEMP <= x"FF000000";
                                        elsif(endereco_conf="000100") then
                                                AD_TEMP <= bar0;
                                        elsif(endereco_conf="000101") then
                                                AD_TEMP <= bar1;
                                        else
                                                --registrador  "000011" -- cache line size (00), latency timer (00), header
                                                                        -- type (00), BIST (00)
                                                --registrador  "001010" -- cardbus CIS pointer
                                                --registrador  "001100" -- expansion ROM address register
                                                --registrador  "001011" -- subsystem vendor ID (00000000)
                                                --registrador  "001101" -- CAP_PTR (00) e reservado (000000)
                                                --registrador  "001110" -- reservado
                                                --registrador  "010000" -- reservado
                                                --e qualquer outro caso
                                                AD_TEMP <= x"00000000";
                                        end if;
                                end if;
                        else
                        end if;
                end if;
        end if;
    end process;

    -- Este processo sincroniza AD_OUT com a borda de descida do clock.
    -- Isto é necessário porque a leitura é feita na borda de subida.
    process(CLK)
    begin
        if(CLK'event and CLK='0') then
                if(ESTADO=READ1) then
                        AD_OUT<=AD_TEMP;
                end if;
        end if;
    end process;

end BANCO_REG;
```

No banco de registradores do espaço de configuração existem registradores de leitura/escrita e somente leitura. Os registradores só de leitura são implementados com constantes no código VHDL. Se num ciclo de leitura um destes registradores for lido, será retornada a constante

correspondente a ele. Já os registradores de leitura/escrita são implementados como registradores convencionais. Eles recebem um valor inicial no *reset*. Observa-se que BAR0 e BAR1 recebem uma codificação no *reset* que significa que estão alocando 128 bytes de espaço de I/O (TAM=128). Todos os valores do espaço de configuração estão armazenados em formato *little-endian*. Para que seja possível utilizar operadores relacionais em VHDL, os valores são convertidos para *big-endian*. Esta conversão se dá nos sinais BAR0BE e BAR1BE.

A Figura 29 apresenta uma simulação de dois ciclos de escrita no espaço de configuração ((1) a (3) e (4) a (6)). Ambos iniciam com o sinal FRAME# sendo ativado no mesmo instante em que é colocado o comando C_CONFIGURATION_WRITE (B) nas linhas C/BE#. As linhas AD recebem um conjunto de *bits* que seleciona qual registador do dispositivo será escrito (BAR0 ou BAR1). O dispositivo é selecionado pelo sinal IDSEL, pois ainda não foi configurado um endereço para o dispositivo se comunicar com o *host*. Os *bits* 7 a 2 representam o registrador do espaço de configuração que será escrito enquanto que os *bits* 10 a 8 especificam qual a função dentro do dispositivo *target* (o dispositivo pode ter várias funções, cada uma com seu próprio espaço de configuração). Entre os *bits* 31 a 11 está a linha IDSEL que está sendo utilizada para acessar o dispositivo *target*. O dado colocado nas linhas AD em (2) será escrito no registrador BAR0, já o dado colocado em (5) será escrito no BAR1. Este valor indica o endereço inicial deste espaço de endereçamento do dispositivo. Pode-se ter até 8 espaços de endereçamento, um para cada BAR, que podem ser de I/O ou I/O mapeado em memória. No *reset* os registradores BAR devem ser inicializados pelo dispositivo, para um valor que indica o tipo de I/O que o dispositivo está requisitando e a quantidade de espaço. O *host* deve se encarregar de ler os registradores BAR do dispositivo antes de escrevê-los para saber estas informações.

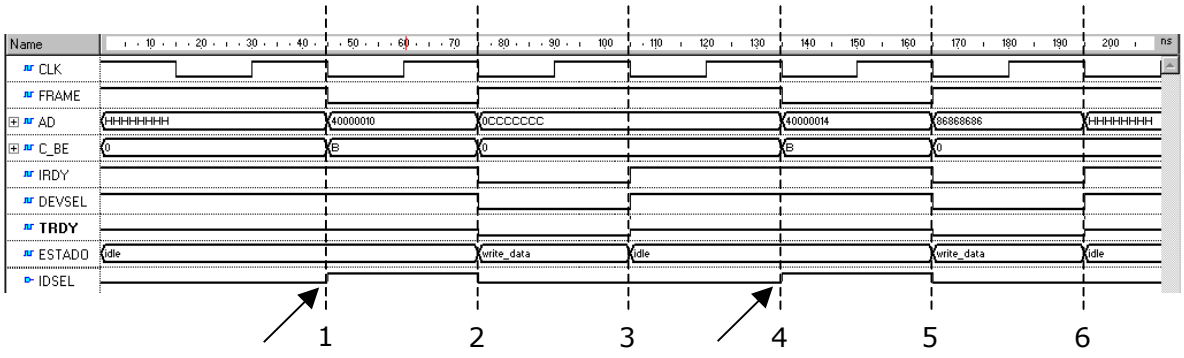

Figura 29 – Simulação de dois ciclos de escrita nos registradores de configuração.

3.4.5 Ciclos Básicos de Operação PCI em 32 Bits

As Subseções seguintes apresentam os resultados de simulação do *soft core* implementado. Este *core* está completamente descrito em VHDL, contendo a máquina de estados de controle (*target*), o bloco de geração/verificação de paridade, os registradores de configuração e uma

pequena aplicação *back-end*.

3.4.5.1 Ciclo de acesso de leitura simples

A Figura 30 apresenta os sinais durante este ciclo. Um ciclo de acesso simples de leitura inicia quando o *master* do barramento PCI ativa o sinal FRAME# (1 – na simulação), colocando um tipo de comando (COMMAND) válido nas linhas C/BE[3::0]# e um endereço válido nas linhas AD[31::0]. Esta fase denomina-se fase de endereçamento (*ADDRESS PHASE*) do ciclo de acesso. Completada esta fase o *master* ativa o sinal IRDY# (*initiator ready*) (2) indicando que está pronto para começar a ler os dados do *target*. O dispositivo que contiver o endereço colocado pelo *master* em AD, indica que foi selecionado ativando o sinal DEVSEL# (*device selected*). Imediatamente após a fase de endereçamento vem a fase de dados (*DATA PHASE*). No início desta fase o *master* do barramento habilita um subconjunto de *bytes* válidos pelo sinal C/BE[3::0]# (C/BE[0] indica *byte* 0, C/BE[1] indica *byte* 1, e assim sucessivamente) e coloca as linhas AD[31::0] em alta impedância, pois o barramento deve ter sua direção invertida (o *master* enviou o endereço e o *target* enviará seus dados – barramento bidirecional). Quando o *target* estiver pronto para enviar os dados ele ativa o sinal TRDY# (*target ready*) (3) e os dados são enviados. Este ciclo termina com o *master* desativando o sinal IRDY# (4) e os sinais de controle são removidos pelo *target.* Observar que ao final do ciclo de leitura é calculada a paridade (sinal PAR).

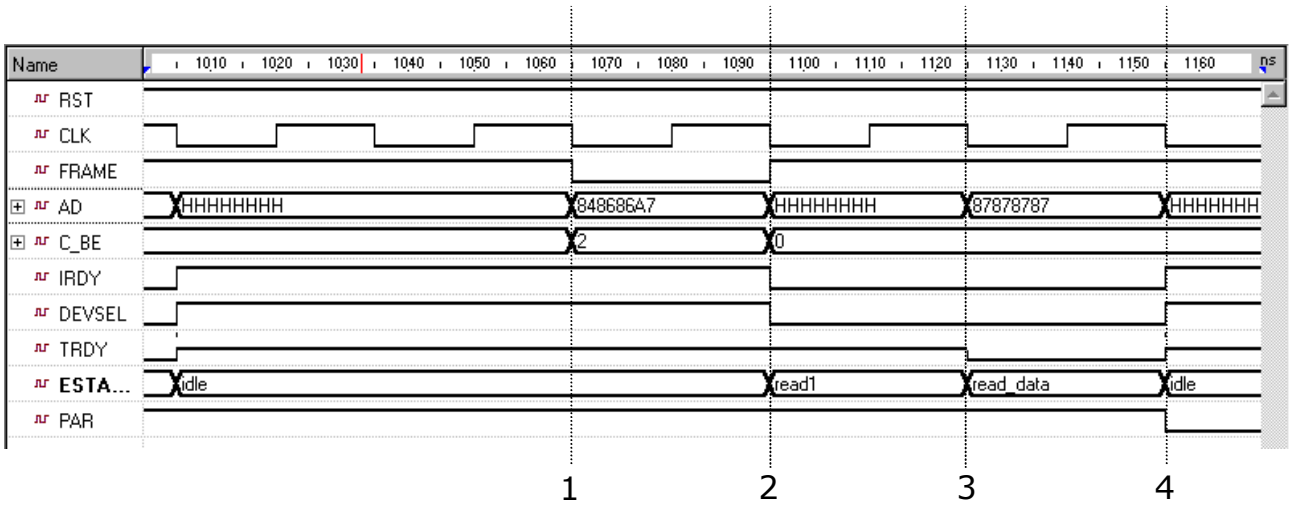

Figura 30 - Sinais de um ciclo de acesso de leitura no modo simples.

3.4.5.2 Ciclo de acesso de escrita simples

Os sinais deste ciclo são mostrados na Figura 31. Um ciclo de acesso de escrita simples inicia similarmente ao ciclo de leitura simples. Uma transação de escrita é similar à transação de leitura, exceto que não há a necessidade da inversão do barramento, durando um ciclo de *clock* a menos. Imediatamente após a fase de endereçamento (1) vem a fase de dados. No início da fase de

dados (2), o *master* habilita um subconjunto de *bytes* válidos em C/BE[3::0]# e aciona as linhas AD[31::0]. Um dado válido nem sempre está presente a cada início de uma fase de dados, pois o *target* pode não estar pronto. Neste caso são inseridos *wait states* pelo *target*. Esta característica ainda não está implementada. A fase final do ciclo de escrita simples encerra-se com a desativação do sinal IRDY# e os sinais de controle são removidos.

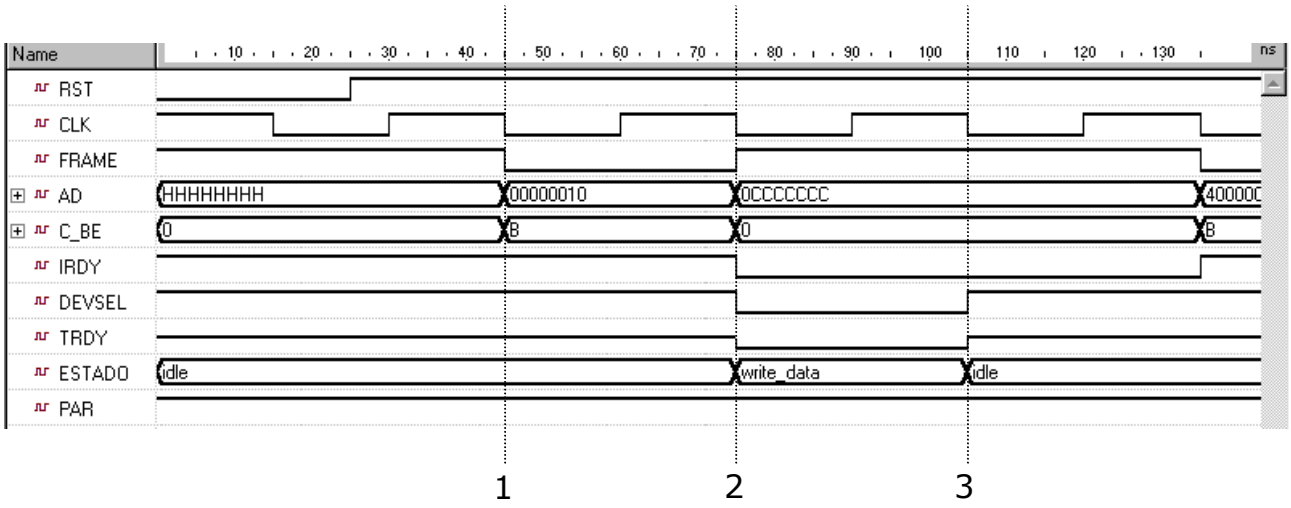

Figura 31 -Sinais de um ciclo de acesso de escrita no modo simples.

3.4.5.3 Ciclo de acesso de escrita em modo *burst*.

A

Figura 32 mostra os sinais para um ciclo de escrita em modo *burst*. O modo *burst* permite que o *master* acesse o *target* através de uma seqüência de micro acessos. O ciclo inicia quando o *master* ativa o sinal FRAME#, coloca a informação de comando do ciclo em C/BE[3::0]# e um endereço válido nas linhas de sinal AD[31::0]. Este endereço estabelece o endereço base para todos os micro acessos. Imediatamente após a fase de endereçamento vem a fase de dados. Para um ciclo de escrita, no início desta fase o *master* informa qual no *byte* a ser acessado em C/BE[3::0]#, e coloca os dados em AD[31::0]. Os ciclo de escrita em modo *burst* tem o seu término quando o sinal IRDY# é desativado um ciclo de *clock* após o sinal FRAME# ter sido desativado. O que diferencia um acesso no modo simples do acesso no modo *burst* é o tempo de permanência do sinal FRAME# ativado.

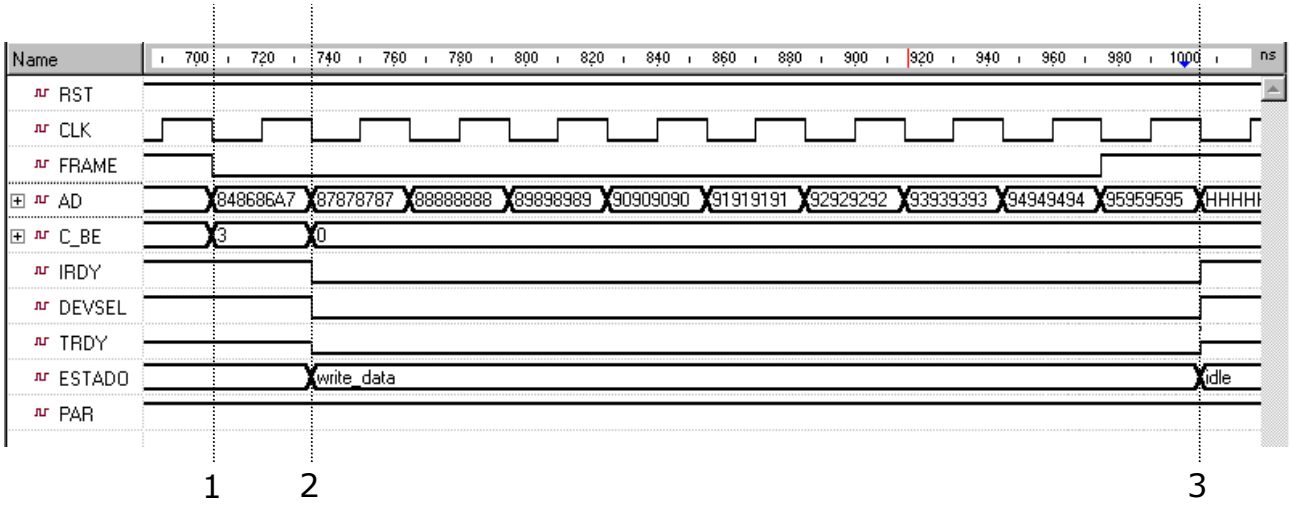

Figura 32 - Sinais de um ciclo de acesso de escrita no modo *burst*. O ciclo termina com a desativação do sinal IRDY# um período de *clock* após o sinal FRAME# ter sido desativado.

No ciclo ilustrado na Figura 32, 9 palavras com 32 *bits* de largura são transferidos. A transferência segue até a desativação do sinal IRDY#. Neste ciclo, o sinal C/BE# = ‘0’ indica que o *byte* menos significativo de cada palavra contém dado válido.

3.4.5.4 Ciclo de acesso de leitura em modo *burst*

A Figura 33 mostra os sinais para um ciclo de leitura em modo *burst*. O modo *burst* permite que o *master* acesse o *target* através de uma seqüência de micro acessos. O ciclo inicia quando o *master* ativa o sinal FRAME#, coloca uma informação de controle do ciclo em C/BE[3::0]# e um endereço válido nas linhas de sinal AD[31::0]. Este endereço estabelece o endereço base para todos os micro acessos. Imediatamente após a fase de endereçamento, as linhas AD[31::0] são colocadas em alta impedância para a inversão do sentido do fluxo de dados no barramento AD. Logo após, vem a fase de dados, quando o *target* envia os dados ao *master*. O ciclo de leitura em modo *burst* tem o seu término quando o sinal IRDY# é desativado um ciclo de *clock* após o sinal FRAME# ter sido desativado. O que diferencia um acesso no modo simples do acesso no modo *burst* é o tempo de permanência do sinal FRAME# ativado.

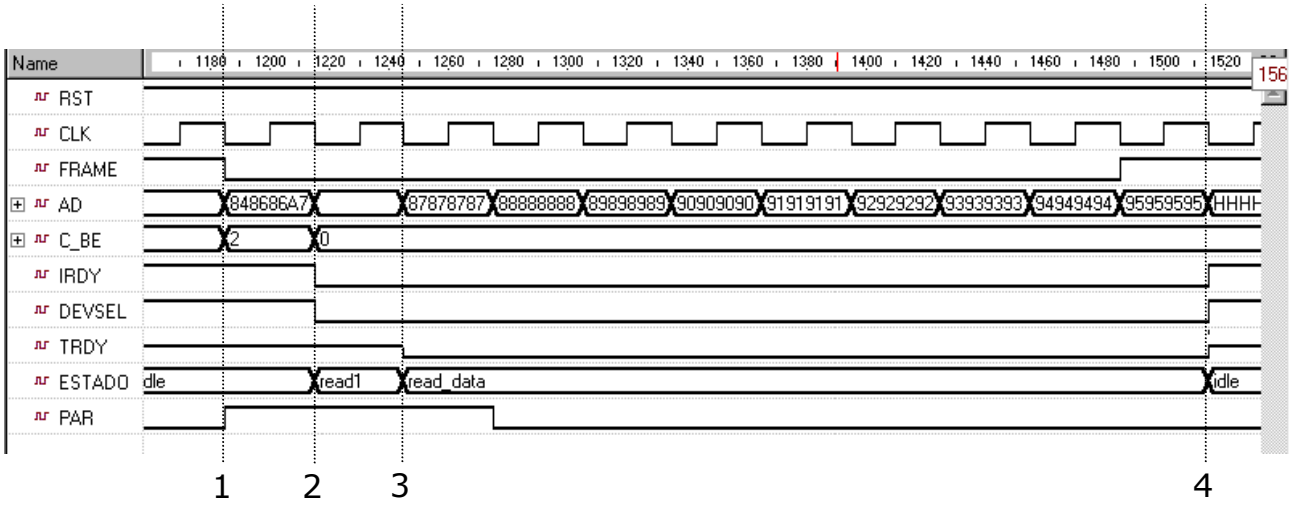

Figura 33 - Sinais de um ciclo de acesso de leitura no modo *burst*. O ciclo termina com a desativação do sinal IRDY# um período de *clock* após o sinal FRAME# ter sido desativado. No total, 9 palavras de 32 bits são transferidas.

3.5 Resultados preliminares da síntese do core PCI desenvolvido

Conforme ilustrado no Anexo I, o circuito "corePCI" contém, além dos módulos especificados nas Seções anteriores, uma aplicação *back-end*. Os resultados da síntese conterão a área deste módulo.

Esta aplicação *back-end* foi desenvolvida para servir de teste de escrita e leitura em registradores. Os registradores foram implementados através de LUT RAM na família XC4000 da Xilinx. LUT RAM é um mecanismo que permite a utilização das LUTs, que compõem os CLBs, como memória RAM. Cada LUT RAM de 4 entradas é capaz de armazenar 16 *bits*, e cada CLB contém 2 LUTs de 4 entradas. Logo, cada CLB pode armazenar 32 *bits*, o que é mais eficiente do que implementar registradores utilizando *process* com *clock* na lista de sensitividade, onde 1 *bit* utiliza um *flip-flop* do CLB, chegando a um máximo de 2 *bits* de registrador por CLB.

O código abaixo apresenta uma parte da aplicação *back-end*. Neste código é apresentado o componente RAM 32x1S, que é uma memória de 32 *bits*, com largura de 1 *bit* e porta simples. Esta memória consome 1 CLB. A aplicação *back-end* implementa 2 conjuntos de registradores de 32 *bits*. O primeiro conjunto de registradores é apresentado neste código (laço *for-generate*). Logo o consumo de área desta aplicação é de 64 CLBs.

```
component RAM32x1S                    --Define um componente RAM 32x1 bits, porta simples
    port (
        WCLK    : IN  std_logic;
        WE      : IN  std_logic;
        D       : IN  std_logic;
        O       : OUT  std_logic;
        A0      : IN  std_logic;
        A1      : IN  std_logic;
        A2      : IN  std_logic;
        A3      : IN  std_logic;
        A4      : IN  std_logic
         );
```

```
    END component;
...

begin

L1:for I in 0 to 31 generate                    -- Implementa 32 RAMs com largura de 1 bit e profundidade de 32 bits
  r1: RAM32x1S
  PORT MAP (      WCLK=> CLK,
                  WE     => en1,
                  D      => DADO_IN(i),
                  O      => do1(i),
                  A4     => add2(4),
                  A3     => add2(3),
                  A2     => add2(2),
                  A1     => add2(1),
                  A0     => add2(0) );
  end generate;
```

Os primeiros resultados de síntese resultaram em um consumo excessivo de área, quando comparada à referência Xilinx, que é de 178-308 blocos lógicos (família 4000, Tabela 6). Este consumo ficou em 307 blocos lógicos (371 CLBs – 64 CLBs do *back-end*). Com o auxílio da ferramenta de planta baixa (*floorplanning*) [34], observou-se que a maior parcela de área consumida era na região dos registradores do espaço de configuração.

Modificando-se o código para que o *core* contenha apenas um espaço de endereçamento, a síntese resultou em uma ocupação de área entre 178 e 196 blocos lógicos (de fato 114 e 132 sem o *back-end*). A sensível redução de área deve-se ao fato que há um número muito grande de operações aritméticas e de comparação (em 32 bits) com os dois BAR. Removendo-se um destes BAR, e os registradores associados ao mesmo, reduziu-se o número de operações em 32 *bits*, e o consumo de área também foi reduzido. A freqüência de operação do *core*, nas primeiras iterações, sem restrições impostas à síntese, foi na ordem de 16 MHz, aquém do esperado 33 MHz. Para aumentar a freqüência de funcionamento, optou-se por restringir a área de posicionamento dentro do FPGA, utilizando-se a ferramenta de *floorplaning* (mesma estratégia adotada na Figura 17). A Figura 34 ilustra esta restrição de posicionamento.

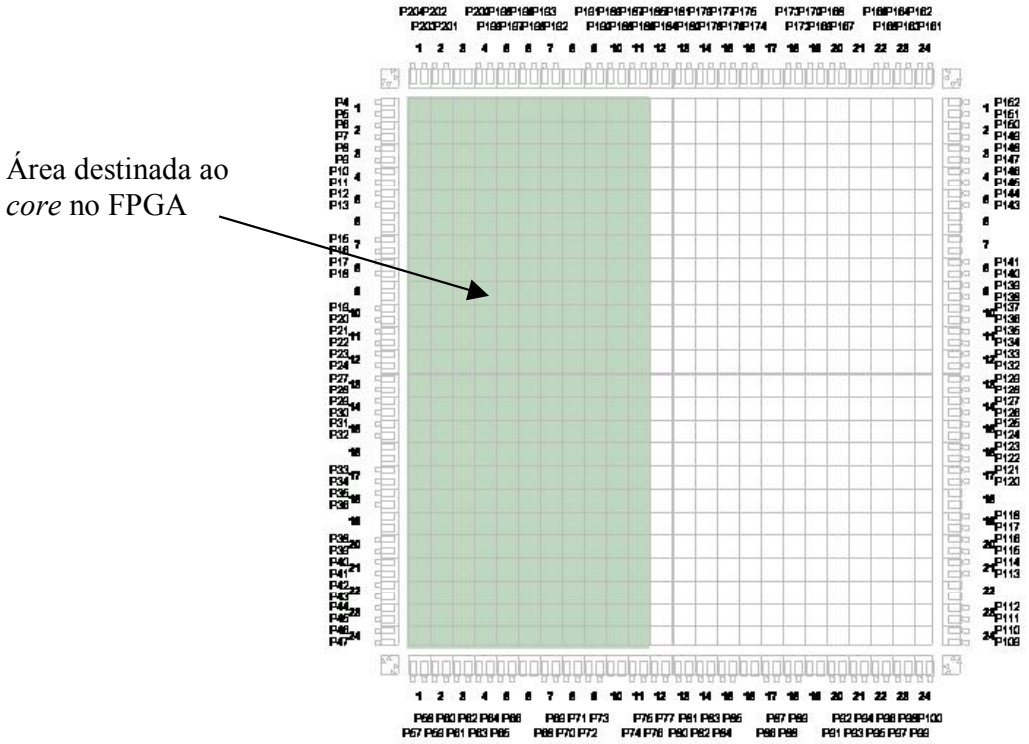

Figura 34 – Restrição de área para a síntese do *core*.

Com esta restrição preliminar, a freqüência de operação passou de 16 MHz para 24 MHz. (41.832 ns de período). Esta freqüência ainda é insuficiente para o padrão PCI, e mostra claramente a dificuldade em implementar este tipo de norma em FPGAs, apesar de que a especificação 32 bits / 33 MHz permite operar entre 0 e 33 MHz. Para que a freqüência passe para 33 MHz, serão necessárias mais iterações com a ferramenta de *floorplanning*, impondo também restrições na síntese lógica. Cabe ressaltar que esta implementação preliminar não fixou a posição dos pinos de entrada e saída. Uma fixação cuidadosa destes pinos contribuirá também para aumentar a freqüência de funcionamento.

Os resultados apresentados na Tabela 10 foram obtidos após a imposição de restrições de *timing*. Com estas restrições, a freqüência de operação do *core* chegou a 40,783 MHz, acima dos 33 MHz esperados.

Tabela 10 - Resultados da síntese com imposições temporais.

```
Timing summary:
-----------------------
Timing errors: 0  Score: 0

Constraints cover 1601 paths, 0 nets, and 812 connections (42.1% coverage)

Design statistics:
   Minimum period:  24.520ns (Maximum frequency:  40.783MHz)
   Maximum path delay from/to any node:  18.939ns

Analysis completed Fri Jun 29 12:44:15 2001
```

As Figura 35 e Figura 36 mostram, respectivamente, os resultados da distribuição dos blocos lógicos no FPGA e a distribuição das linhas de *clock* para que a restrição temporal fosse alcançada.

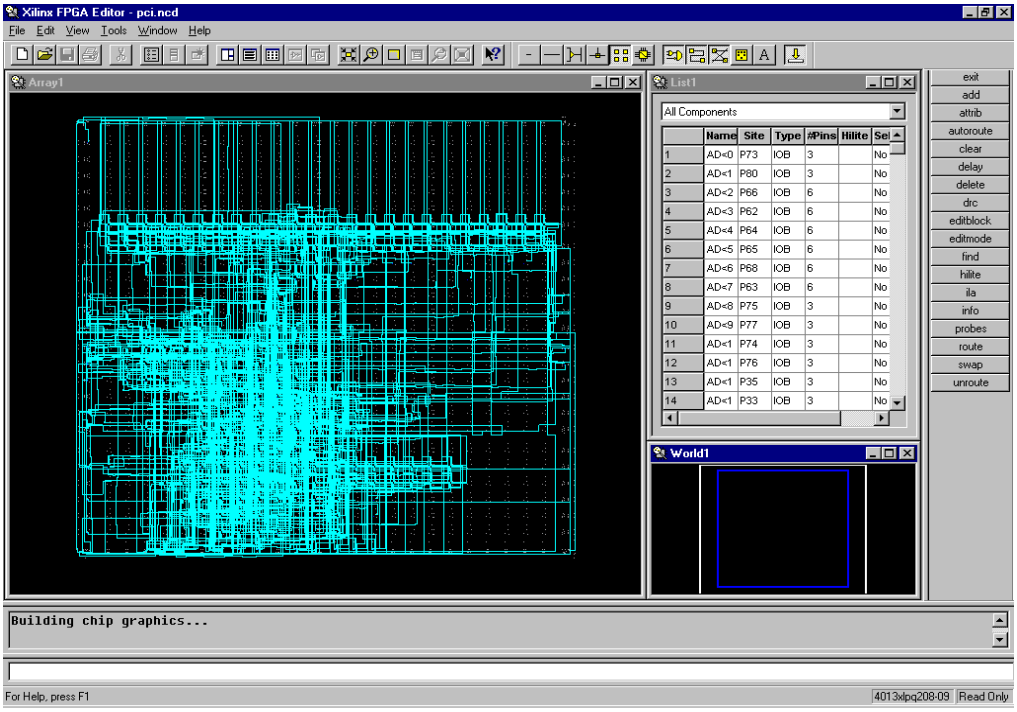

Figura 35 - Roteamento dos blocos lógicos.

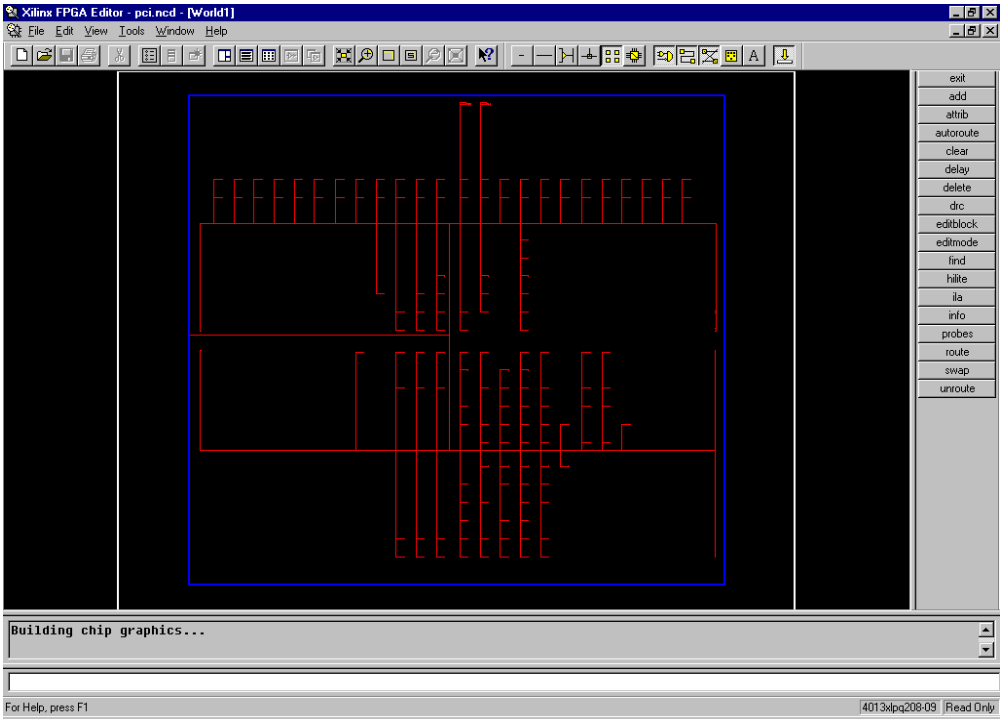

Figura 36 - Distribuição do sinal de *clock*.

Mesmo que haja escorregamento do sinal de *clock* ao longo da estrutura, os componentes dispostos em uma mesma região vertical receberão o sinal de *clock* ao mesmo tempo, isto é, com o mesmo atraso. Esta técnica de distribuição aumenta o desempenho em velocidade do circuito.

4 AMBIENTE DE PROTOTIPAÇÃO

Os Capítulos anteriores detalharam o padrão de barramento PCI e a implementação de um *core* para operar conforme as normas deste padrão. Este Capítulo tem por objetivo mostrar o desenvolvimento de aplicações em FPGAs comunicando-se com um computador *host*, utilizando para tal fim um *core* PCI. O *core* utilizado para este fim é o *core* da empresa XILINX - Xilinx PCI32 LogiCORE [31], do tipo *firm core*, descrito em formato EDIF. O *core* desenvolvido não é utilizado neste Capítulo devido ao fato da placa empregada não permitir em seu uso outro *core* que não seja o pertencente ao sistema de *hardware/software* desenvolvido pelo fabricante.

4.1 Arquitetura da Hardware Object Technology – H.O.T. II-XL

A placa de prototipação que é empregada para a implementação de sistemas computacionais compostos por parte em *hardware* e parte em *software* é a HOT II-XL [26]. A comunicação entre a parte *hardware* e a parte *software* é feita pelo barramento PCI (32 bits, 33 MHz, ou seja, até 132 MB/s). A comunicação entre a parte *software* e o barramento PCI é provida por um *driver* de comunicação e a comunicação entre a parte *hardware* e o barramento PCI é feito por um *core*, disponibilizado juntamente com a plataforma de prototipação. A Figura 37 ilustra a arquitetura da placa de prototipação HOT II-XL.

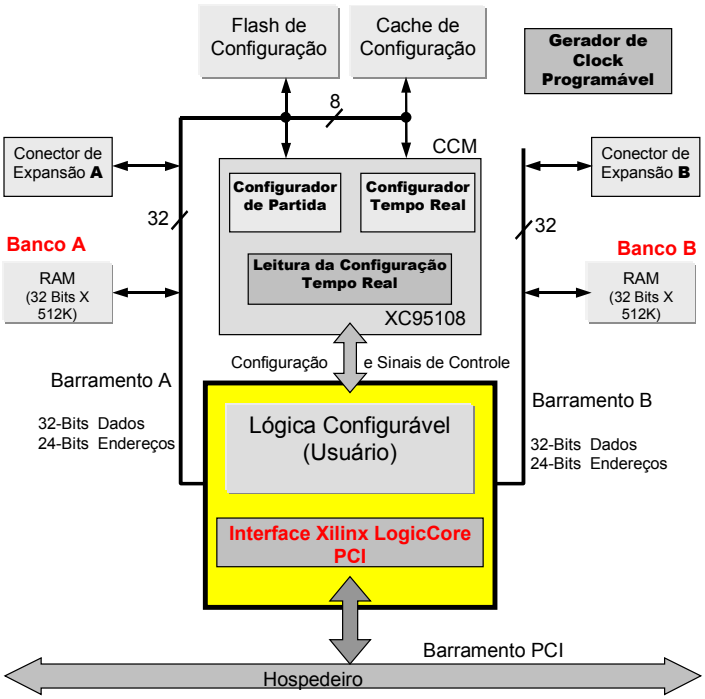

Figura 37 - Arquitetura da placa de prototipação HOT II-XL.

Os principais componentes desta placa de prototipação são:

- ***Configuration Flash***. Guarda de forma não volátil configurações para o FPGA.
- ***Configuration Cache***. Armazena configurações para o FPGA em memória SRAM. Logo, se o sistema for reinicializado perde-se as configurações lá armazenadas.
- ***Configuration Manager***. Gerencia a configuração do FPGA. A reconfiguração pode ser feita partindo de uma configuração armazenada na *flash*, ou na *cache* de configuração. O CPLD XC95108 é responsável por este gerenciamento
- ***RAM Bank A and Bank B***. Dois bancos de memória para serem usados pela aplicação do usuário.
- ***FPGA***. dispositivo XC4062XL. Neste dispositivo é carregada a aplicação do usuário (parte *hardware*) e o *core* PCI. A interface da aplicação do usuário com o *core* é descrita posteriormente.
- ***Clock Programável***. Freqüência programável pelo usuário no intervalo entre 360 kHz e 100MHz.

A comunicação entre o *host* (PC) e a placa HOT II-XL é feita através de uma *Application Program Interface* – API. Esta API é uma biblioteca de objetos (e métodos) de comunicação, permitindo reconfigurar a placa e comunicar-se com o circuito implementado na mesma.

A comunicação entre a placa HOT II-XL e o *host* (PC) é feita através de uma biblioteca de *hardware*. Esta biblioteca é o *core* propriamente dito, contendo o *netlist* do *core* em formato EDIF e o símbolo deste para ser utilizado nos projetos no sistema de CAD Foundation. Esta biblioteca realiza, também, a comunicação da aplicação do usuário (dispositivo *back-end*) com o barramento PCI e as funções da placa (basicamente bancos de RAM).

Uma importante característica deste ambiente de prototipação é a possibilidade de armazenar diferentes circuitos de configuração em uma memória *flash*. Até três diferentes configurações são armazenadas simultaneamente, permitindo carregar circuitos com funções diferentes durante a execução de uma dada aplicação.

A Figura 38 mostra uma foto da placa de prototipação utilizada.

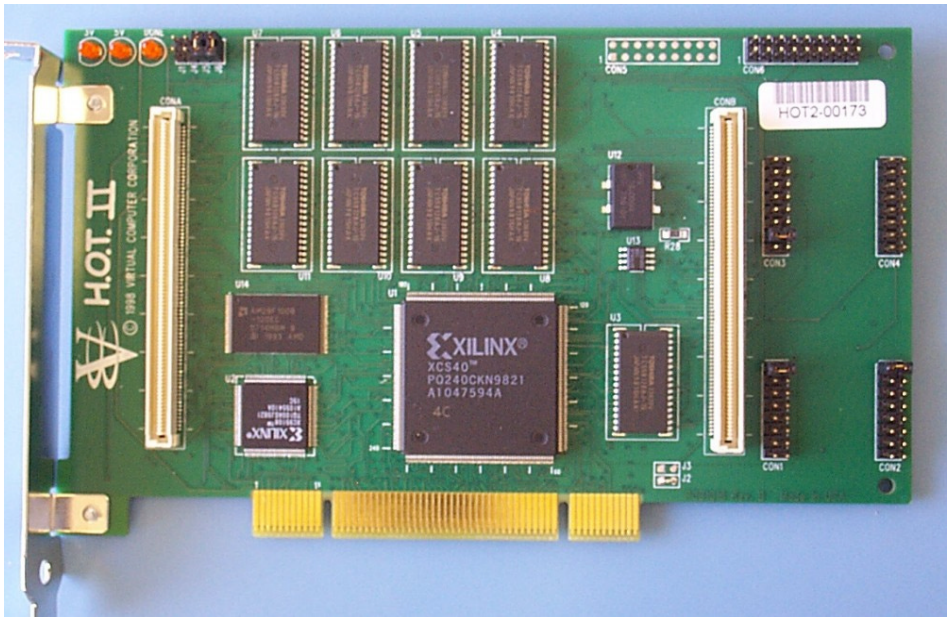

Figura 38 - Placa de prototipação HOT II.

4.1.1 Modelo de execução Hardware/Software

A Figura 39 ilustra o modelo de execução *hardware/software*. Neste modelo, a execução da parte do sistema que é crítica em termos de desempenho, é implementada em *hardware* (FPGA), e a parte não crítica, ou que requer uma maior quantidade de entrada/saída, é implementada em *software* (processador do *host*). O maior gargalo deste ambiente de prototipação é o tempo gasto para transmitir dados entre o processador e o dispositivo, o qual pode ser superior ao ganho de desempenho estimado [30].

Na parte *software* utiliza-se programas escritos em linguagem C++, com funções para acessar os recursos da placa de prototipação (FPGA, memória RAM, *cache* de configurações, *flash*). Essas funções pertencem a uma API proprietária da plataforma HOT II-XL. A parte *hardware* é composta pelo *hardware* do usuário (*back-end*) e um módulo de comunicação com o barramento - biblioteca de *hardware*.

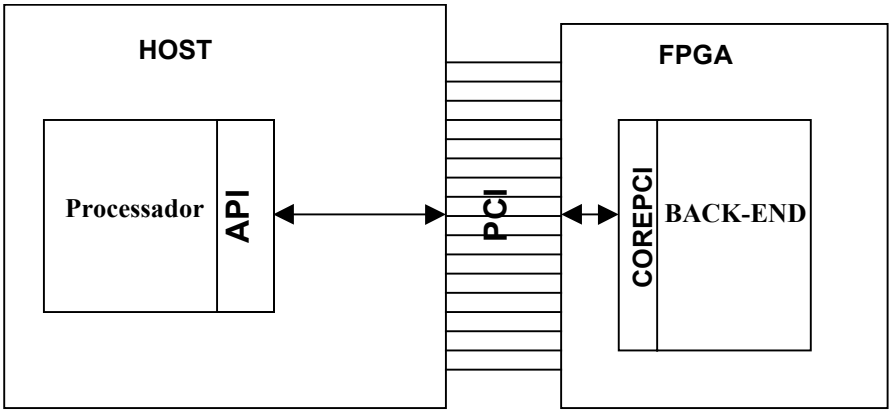

Figura 39 - Modelo de execução *Hardware*/*Software*.

4.1.2 Operação da API da HOT II-XL

A HOT II-XL possibilita a reconfiguração do FPGA (PCI *core* + *back-end*) através de funções da API, escritas em C++. Esta API permite comunicação, através de uma interface de alto nível, com o *core* implementado no FPGA. Desta forma é possível fazer com que o *hardware* comunique-se com o *software* sem a necessidade de se programar um driver de dispositivo.

Para carregar o *bitstream* contendo a configuração do FPGA a partir de um programa executável, é necessário que o *bitstream* seja convertido para um formato denominado de *run-time* (que gera um arquivo com extensão .hot). Este formato é chamado de *Hardware Object*. Desta forma, ao gerarmos o executável, este conterá o código C++ compilado e o *bitstream* em um único arquivo.

Toda interface com a placa é feita através de uma classe C++ denominada **Hot2**. O construtor do objeto é responsável pela comunicação com o driver da placa. Para verificar se o objeto foi corretamente inicializado utiliza-se o método *CheckBoard*, da classe HOT II. Se todos os recursos foram inicializados corretamente, é retornada a constante *EHOT2_SUCCESS*.

Os principais métodos da classe Hot2 são:

- Reset(int n=0): Inicializa a placa para a configuração n. Se n for omitido, o *default* é 0.
- LoadConfig(char* fileName, s_HOTConfig* config): Carrega a configuração de um arquivo para a memória do *host*.
- LoadCache(s_HOTConfig* config, word location): Carrega uma configuração (contida na estrutura config), para a posição especificada da *cache* de configuração.
- RtrCache(word n=0): Reconfigura o FPGA da placa com a configuração contida no *slot* n da *cache*.
- Write(word addr, word data): Escreve um dado na placa em um dado endereço (ver na Tabela 11 os endereços dos periféricos da placa).
- Write(word addr, word* data, int dataSize): Escreve um vetor de dados na placa em um dado endereço (ver na Tabela 11 os endereços dos periféricos da placa).
- Read(word addr) e Read(word addr, word* data, int dataSize): Lê dados da placa.

Os métodos Read e Write operam com endereços absolutos dentro da placa. Estes endereços estão listados na Tabela 11. Pode-se acionar uma escrita ou leitura em *burst* utilizando os métodos Write(word addr, word* data, int dataSize) e Read(word addr, word* data, int dataSize).

Tabela 11 - Endereços absolutos dos componentes da placa.

Endereço inicial	Nome	Modelo de acesso	Descrição
0x000000	RamAOffSet	R/W	Banco de Memória A
0x400000	RamBOffSet	R/W	Banco de Memória B
0x800000	CcmOffSet	R	Gerenciador da Cache de Configuração
0xA00000	UserOffSet	R/W	Espaço do usuário no FPGA
0xB00000	SytemOffSet	Reserved	Área do Sistema no FPGA
0xC00000	CcMemOffSet	Reserved	Área do Sistema da Cache de Configuração

4.1.3 Operação da biblioteca de *hardware*

A comunicação entre o dispositivo *back-end* e o barramento PCI é feita através do *core* PCI. O acesso a este *core* é feito a partir de um símbolo no editor de esquemáticos. Não há forma direta de se trabalhar com este *core* a partir de VHDL. A Tabela 12 apresenta os sinais aos quais o usuário tem acesso.

Tabela 12 - Sinais do *core* PCI disponíveis para a aplicação do usuário.

Interface HOT II-XL		Descrição
Do barramento PCI	D_IN[31::0]	Barramento de dados de entrada
	PCI_ADR[31::0]	Barramento de endereços
Controles do Banco de Memória A	MAD_IN[31::0]	Barramento de entrada da memória do Banco A
	MAD_OUT[31::0]	Barramento de saída do Banco A
	MAD_ADR[23::0]	Barramento de endereços de memória do Banco A
	MA_RW	Lê/Escreve dados no Banco A (1 = Leitura)
	MA_ON	Habilita Banco A
	MA_BSY	*Host* acessando Banco A
Controles do Banco de Memória B	MBD_IN[31::0]	Barramento de entrada da memória do Banco B
	MBD_OUT[31::0]	Barramento de saída do Banco B
	MBD_ADR[23::0]	Barramento de endereços de memória do Banco B
	MB_RW	Lê/Escreve dados no Banco B (1 = Leitura)
	MB_ON	Habilita Banco B
	MB_BSY	*Host* acessando Banco B
Controles do *Target*	SLV_DCD[15::0]	Decodificação do *target*
	DATA_VLD	Indica que dados foram transferidos para o FPGA
	SRC_EM	Pulso que inicia uma transferência para o FPGA
	URDY	*User Ready*
	USER_INT	*User Interrupt*
	USER_LED	Diodo emissor de luz, disponível ao usuário
Ao barramento PCI	D0_OUT[31::0]	Barramento de leitura D0
	D1_OUT[31::0]	Barramento de leitura D1
	D2_OUT[31::0]	Barramento de leitura D2
	D3_OUT[31::0]	Barramento de leitura D3
	FCLK	*Clock* programável
	PCI_FLK	*Clock* do barramento PCI

4.1.4 Operação dos bancos de memória

A HOT II-XL contém 4 MB de memória SRAM, organizado em dois bancos de memória independentes: Banco A e Banco B, cada um com 2 MB (com largura de 32 *bits*).

O acesso a estes bancos de memória se dá da seguinte maneira:

- O sinal MA_ON habilita a memória. Deve estar em '1' para que se possa realizar operações com a memória. Esse sinal interfere na comunicação com o *host*. Se MA_ON estiver em '0' o driver da HOT vai colocar em um *buffer* os valores a serem escritos na memória (eles somente serão escritos quando MA_ON for a '1').

- Os sinais MA_OE e MA_WE definem se a operação com a memória será de leitura ou escrita. Quando MA_OE for '0' e MA_WE for '1' os dados que estiverem em MAD_OUT serão escritos no endereço selecionado por MAD_ADR. Quando MA_OE for '1' e MA_WE for '0' MAD_IN receberá da memória o valor armazenado no endereço selecionado por MAD_ADR.

4.2 Fluxo de Projeto Utilizando o Core PCI

O primeiro passo para implementar uma aplicação na plataforma de prototipação HOT II-XL é a criação da aplicação do usuário (dispositivo *back-end*). Para isso há no sistema de desenvolvimento um projeto padrão, que é utilizado como modelo. Este projeto padrão contém a interface externa que da aplicação do usuário deve possuir. A Figura 40 ilustra o código VHDL que é utilizado como modelo para o desenvolvimento da aplicação do usuário.

O conjunto de pinos externos deve corresponder ao conjunto de sinais descritos na Tabela 12. O projeto contendo o VHDL da aplicação do usuário não deve ter a síntese física realizada (posicionamento e roteamento). A única etapa realizada é a síntese lógica. O resultado desta síntese é um *netlist* em formato EDIF, que é posteriormente utilizado, juntamente com o *core* PCI para a síntese física.

Comparando-se este fluxo de projeto com o fluxo apresentado na Seção 3.2, observa-se que o ambiente de prototipação não dispõe de um modelo funcional do *core* para simulação, o que dificulta a validação das aplicações desenvolvidas. Tipicamente, o usuário descreve sua aplicação sem preocupar-se com o *core* PCI, validando-a por simulação. Uma vez sua aplicação desenvolvida, é feita a integração aos pinos do *core*.

Uma vez obtido o *netlist* EDIF, deve-se copiar este *netlist* para o local onde se encontra o *netlist* do *core*, o esquemático que une os *netlists* (usuário e *core*) e o arquivo contento as restrições

de temporização e posicionamento do *core* no FPGA.

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

 entity user_app is
  port       (
      -- Data in and out
D_IN                  : in   std_logic_vector(31 downto 0);
PCI_ADR               : in   std_logic_vector(31 downto 0);
D0_OUT                : out  std_logic_vector(31 downto 0);
D1_OUT                : out  std_logic_vector(31 downto 0);
D2_OUT                : out  std_logic_vector(31 downto 0);
D3_OUT                : out  std_logic_vector(31 downto 0);

      -- Memory Bank A
      MAD_IN          : in  std_logic_vector(31 downto 0);
      MAD_OUT         : out std_logic_vector(31 downto 0);
      MA_ADR          : out std_logic_vector(23 downto 0);
      MA_WE,
      MA_OE,
      MA_ON           : out std_logic;
      MA_BSY          : in  std_logic;

      -- Memory Bank B
      MBD_IN          : in  std_logic_vector(31 downto 0);
      MBD_OUT         : out std_logic_vector(31 downto 0);
      MB_ADR          : out std_logic_vector(23 downto 0);
      MB_WE,
      MB_OE,
      MB_ON           : out std_logic;
      MB_BSY          : in  std_logic;

      -- Control signals
      SLV_DCD         : in std_logic_vector(15 downto 0);
      BASE_HIT0       : in std_logic;
      S_DATA          : in std_logic;
      SRC_EM          : in std_logic;
      S_WRDN          : in std_logic;
      S_DATA_VLD      : in std_logic;

      -- Configuration signals that need to be set
      INTONBOOT       : out std_logic;
      PRT             : out  std_logic_vector(3 downto 0);
      USER_INT        : out std_logic;
      USER_LED        : out std_logic;
      URDY            : out std_logic;
      HOLDOFF         : out std_logic;
      FCLK            : in std_logic;
      PCI_CLK         : in std_logic
      );
 end user_app;

architecture rtl of user_app is

                          -- Aqui são inseridos os sinais da aplicação do usuário --
begin
                              -- Aqui fica a descrição da aplicação do usuário --
end rtl;
```

Figura 40 - Modelo de descrição VHDL para integrar a aplicação do usuário ao *core* PCI.

Observa-se que não há um ambiente integrado de desenvolvimento, sendo necessária a execução de diversas etapas manuais. Também não há a possibilidade de desenvolvimento puramente a partir de VHDL. Deve-se ter, no nível mais alto de hierarquia, um esquemático. A Figura 41 ilustra o esquemático que une os dois *netlist.*

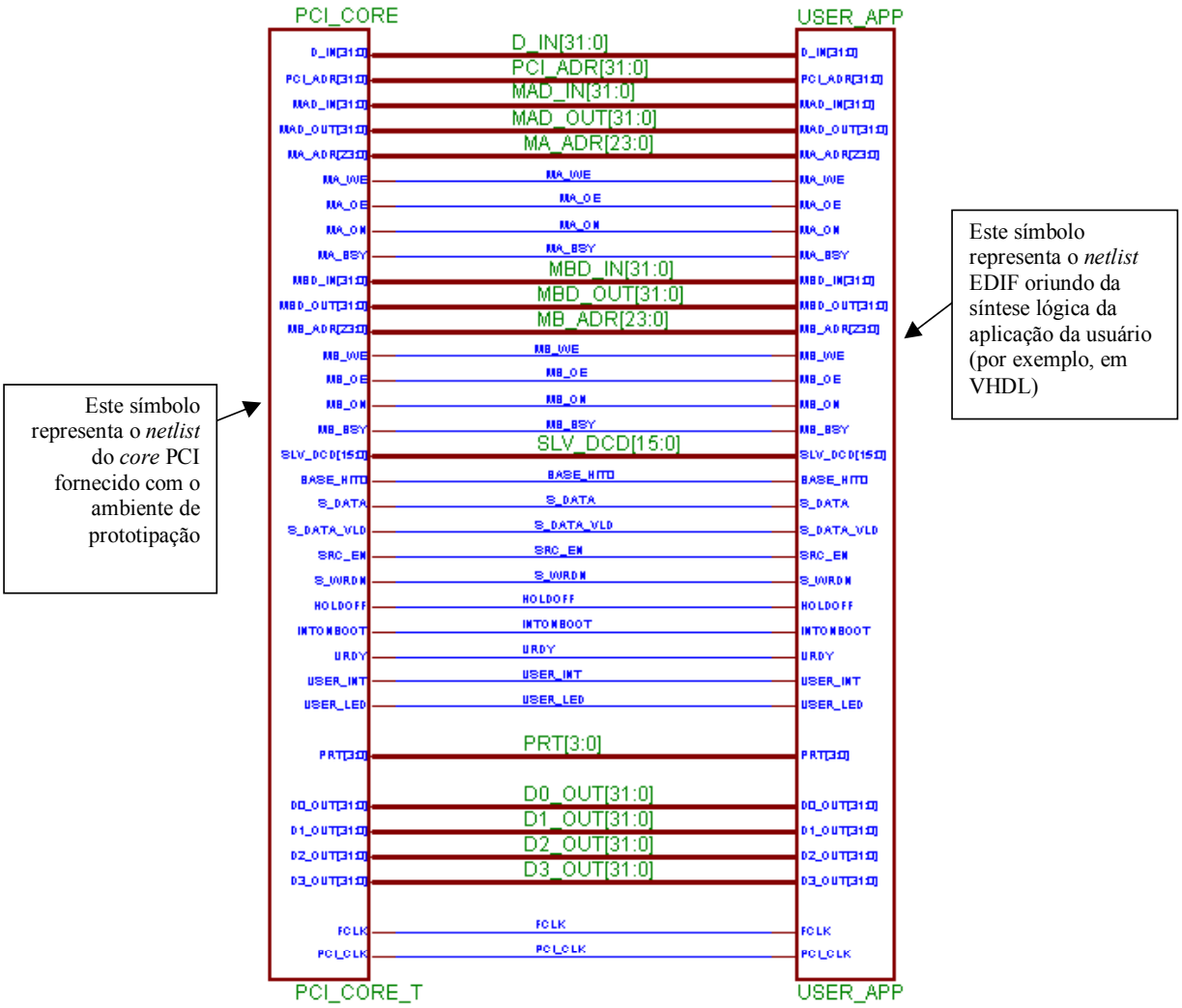

Figura 41 - Esquemático utilizado para união dos *netlist.*

Uma vez obtido um diretório de projeto contendo os dois *netlists* e o arquivo com as restrições, procede-se à ultima etapa no fluxo da síntese de *hardware*, que é a síntese física. É importante comentar que o arquivo de restrições contém três partes principais, (*i*) localização dos pinos de entrada e saída na periferia do FPGA, (*ii*) restrições de temporização e (*iii*) posicionamento dos elementos do *core* no interior do FPGA. Este posicionamento é importante para que o circuito alcance o desempenho esperado, que é a freqüência de operação em 33 MHz.

O arquivo resultante da síntese física, em formato binário padrão XILINX (arquivo com extensão .bit), é convertido para um formato denominado *run-time*. Uma vez que o arquivo de configuração do FPGA está em formato *run-time*, é possível utilizá-lo em um programa (parte de *software*). O ambiente Microsoft Visual C++ 6.0 é exigido pela plataforma HOT II-XL.

A Figura 42 apresenta o fluxo de projeto de *hardware* e *software* da aplicação do usuário.

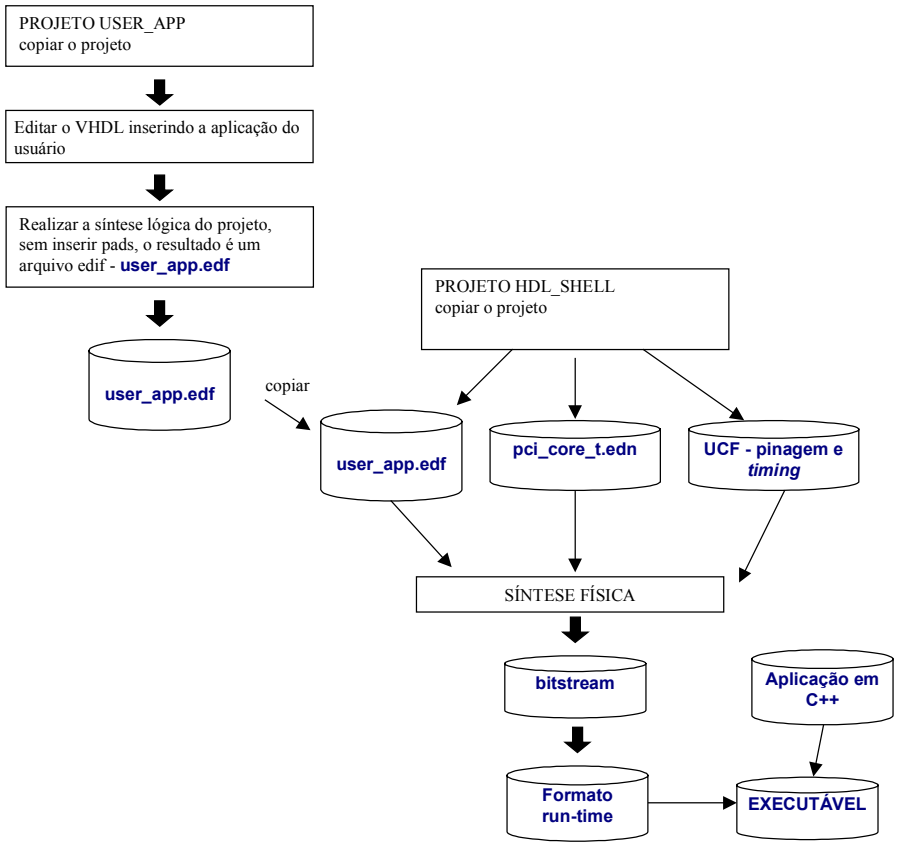

Figura 42 - Fluxo de projeto da parte *hardware* e *software.*

O código C++ apresentado a seguir demonstra a interação *hardware/software* com a placa HOT II-XL. A aplicação reconfigura o FPGA da HOT II-XL, e a seguir faz algumas operações de leitura e escrita, tanto nos bancos de memória quanto no espaço de endereçamento do usuário.

```
void main()
{

 //  Declara variavel. Uma vez declarada, a conexão com a placa se estabelece e pode ser acessada.
 Hot2* Board=new Hot2;

 // Teste rapido... Placa detectada?   O driver esta rodando?
 if (Board->CheckBoard()) {
   exitOnError(Board);
 }

 // Após os testes leva a placa a um estado inicial
 Board->Reset();

 //  A configuracao depende do dispositivo escolhido
 //  Carga do bitstream (.bit) na memória principal coloca na cache na posicao 0.

  switch (Board->GetCompType()) {
  case SPARTAN40:
              …
  case Xc4062XLT:
  case Xc4062XLA:
              resOK=LoadIntoCache(Board, "hdlshell.hot", 0);
              if (!resOK) exitOnError(Board);
  }
   break;
 }

 // Reconfigures the board FPGA with data from cache position 0
 if (Board->RtrCache(0)==-1) {
   exitOnError(Board);
 }
```

```
  Board->Write(0xA00000,0);    // comando para o FPGA- inicializa a máquina de estados

   for(int i=0;i<80;i+=4) {      // escreve as 20 posicoes iniciais da RAM
       Board->m_ramBankA->Write(i,i);
  }

  // le um dado do inicio do espaco de enderecamento do usuario e escreve na tela
  printf("Dado lido da hot %d\n", Board->Read(0xA00000));

  Board->Write(0xA00004,0);     // escreve um dado (0) na segunda posicao do espaco do usuario

// le as primeiras 20 posicoes do banco de memoria A da hot
  for(i=0;i<80;i+=4) {
       printf("Posicao %d= %d\n",i,Board->m_ramBankA->Read(i));
  }

  delete Board;

  exit(1);
}
```

Uma vez o código C++ corretamente descrito, compila-se este, gerando-se um executável. Conforme visto no exemplo acima, o executável carrega o *bitstream.* Há um segundo método de configuração, que é compilar o fonte C++ juntamente com o *bitstream* (como um *header*), gerando-se um arquivo executável único.

4.3 Resultados de síntese

A Figura 43 apresenta os relatórios de utilização de área e freqüência de operação para uma aplicação do usuário utilizando o *core* Xilinx (teste de acesso à memória RAM).

```
Target Device  : x4062xl
Target Package : hq240
Target Speed   : -08

   Number of CLBs:          471 out of  2304   20%
     CLB Flip Flops:     709
     CLB Latches:          0
     4 input LUTs:       552 (5 used as route-throughs)
     3 input LUTs:       358 (217 used as route-throughs)
   Number of bonded IOBs:     173 out of   193   89%
   Number of clock IOB pads:    2 out of    12   16%
   Number of TBUFs:           357 out of  4800    7%
   Number of BUFGLSs:           2 out of     8   25%
   Number of readclk:           1 out of     1  100%
   Number of readback:          1 out of     1  100%
   Total equivalent gate count for design: 9815

Timing summary:
---------------

Timing errors: 3  Score: 3548

Constraints cover 20482 paths, 0 nets, and 6166 connections (92.0% coverage)

Design statistics:
   Minimum period:  30.195ns (Maximum frequency:  33.118MHz)
```

Figura 43 - Relatório de ocupação de área e *timing*.

O consumo total de área foi de 471 CLBs, mostrando que este core tem um pequeno consumo de área, deixando a maior parte da área de lógica programável para a aplicação do usuário. Outro ponto importante é a freqüência de operação que foi atendida, 33 MHz. Para que esta restrição de freqüência seja atendida, a síntese deve ter esforço máximo em todas as etapas, podendo consumir até 20 minutos de processamento (Pentium III 550 MHz).

5 CONCLUSÃO

Este trabalho detalhou o funcionamento e as operações do barramento PCI, assim como mostrou a importância da utilização de *cores* em sistemas digitais complexos. A principal contribuição deste trabalho foi o desenvolvimento de um *soft core* VHDL para possibilitar aos projetistas de *hardware* um meio de acesso ao barramento PCI a partir de aplicações *target*.

Este trabalho foi um excelente meio de aprimorar o uso das técnicas de projeto de sistemas digitais com dispositivos programáveis, implicando no uso de tecnologia no estado da arte para projeto, validação funcional, validação de temporização e validação de protótipo.

O s*oft core* desenvolvido está funcionalmente de acordo com o padrão estabelecido pela norma do barramento PCI e resultados preliminares indicam um baixo consumo de área, sendo possível através da inserção de restrições de *timing* alcançar 33 MHz (obteve-se 24 MHz apenas restringindo a posição dos blocos, sem impor freqüência de operação). Os resultados obtidos após a imposição de restrições de *timing*, a freqüência de operação do *core* chegou a 40,783 MHz, acima dos 33 MHz esperados.

No decorrer do trabalho também foi utilizado um ambiente de prototipação comercial (HOT II-XL), contendo um *firm core*. A partir do uso deste ambiente, foram observadas carências existentes em termos de *software* para projeto de sistemas digitais que utilizam *cores*. O ambiente utilizado, um dos poucos disponíveis comercialmente, não tem um ambiente integrado de *software* nem o modelo funcional de simulação. O projeto deve ser integrado ao barramento PCI, supondo-se correta a descrição fornecida pelo fabricante. Outra lição oriunda da utilização deste ambiente é a dificuldade em se obter o atendimento das restrições temporais (o tempo de síntese é muito grande).

Como o *core* desenvolvido é um *soft core*, não há a necessidade de um modelo funcional de simulação. A aplicação do usuário pode ser simulada diretamente com o *core*, sendo necessário apenas desenvolver o *test-bench*.

A seguir relacionam-se os trabalhos que podem ser feitos como continuidade desta Dissertação:

- A validação e a implementação do *core* e a aplicação do usuário em uma plataforma de prototipação. Esta etapa necessita de uma plataforma flexível, que permita experimentos com diversos *cores*, o que não é o caso da HOT-II. É necessário adquirir, ou desenvolver, tal

ferramenta.

- Ampliar a máquina de estados para que o *core* possa operar como *master* do barramento.

- Desenvolver uma interface simples entre o *core* e a aplicação *back-end*, disponibilizando ao projetista os sinais necessários à conexão da sua aplicação ao *core*. Isto corresponde a definir uma interface semelhante à Figura 41. O ambiente de desenvolvimento deve ser todo em VHDL, sem utilização de esquemáticos, para uma maior facilidade de uso por parte do usuário, já que linguagens de descrição de *hardware* permitem aumentar o nível de abstração de projeto.

- Desenvolver *drivers* em *software* para acesso ao dispositivo PCI. Este trabalho já foi iniciado e será desenvolvido no sistema operacional Microsoft Windows NT 4.0, devido a sua disponibilidade e ampla utilização, além de uma certa facilidade de obtenção de documentação sobre o mesmo. Para a realização desta tarefa está sendo utilizado o sistema denominado *Windows NT Driver Development Kit* – DDK, que possui as ferramentas necessárias para a criação de *drivers*, contendo desde bibliotecas até ferramentas de montagem de *drivers*, depuradores e gerenciadores de *kernel*.

6 REFERÊNCIAS

[1] Algotronix Ltd, "**CAL 1024 Data Sheet**". Edinburgh, Scotland, 1989.

[2] ALTERA, "**Excalibur Development Kit Data Sheet**". Disponível por ftp em www.altera.com/literature, V1.0, Junho 2000.

[3] ALTERA, "Altera Flex10K". Disponível por ftp em www.altera.com/literature/ds/dsf10k.pdf, Junho 2000.

[4] Athanas P. M. and Silverman H. F., "**Processor Reconfiguration Through Instruction-Set Metamorphosis**". IEEE Computer, New York, pp. 11-18, Março 1993.

[5] Bergamaschi R. A. and Lee W. R., "**Designing Systems-on-Chip Using Cores**". Design Automation Conference - DAC'2000, Los Angeles, California, USA, pp 420-425, 2000.

[6] Cappelatti E. A., Moraes F. G., Calazans N. L. V. e Oliveira L. A., "**Barramento de Alto Desempenho para Interação Hardware/Software**". VII Workshop IBERCHIP, Montevidéu, Uruguai, Março 2001.

[7] Case J. et al., "**Design Methodologies for Core-Based FPGA Designs**". XILINX Publications, http://www.xilinx.com, Abril 1997.

[8] Concurrent Logic, "**CFA 6006 Field Programmable Gate Array Data Sheet".** Sunnyvale, California, 1991.

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

[10] Dehon A., "**DPGA – Coupled Microprocessors: Commodity ICs for the Early 21st Century**". IEEE Workshop on FPGA Custom Computing Machines – FCCM ´93, D. A . Buell and K. L. Pocek, Napa, CA, pp. 202-211, Abril 1993.

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

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

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

[14] Kendall G. W., "**Inside the PCI Local Bus**". Byte; pp. 177-180; Fevereiro 1994.

[15] Kuusilinna K. et al., "**Field Programable Gate array-based PCI Interface for a coprocessor system**". Microprocessor and Microsystems, vol. 22, pp. 373-388, Janeiro 1999.

[16] Lewis D. M. et al., "**The Transmogrifier-2: A 1 Million Gate Rapid-Prototyping System**". IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6, No. 2, pp. 188-198, Junho 1998.

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

[18] Muroga H. et al., "**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 1991.

[19] PCISIG, "**PCI Local Bus Specification – Revision 2.2**". Dezembro 1998.

[20] Plessey Semiconductors, "**ERA 60100 Preliminary Data Sheet**". Swindon, England, 1989.

[21] Radunovic B., "**An Overview of Advances in Reconfigurable Computing Systems**". Proceedings of the 32nd Hawaii International Conference on System Sciences, pp. 1-10, 1999.

[22] Rose J. et al., "**Architecture of Field-Programmable Gate Arrays**". Proceedings of the IEEE, Vol. 81, No. 7, pp. 1013-1028, Julho 1993.

[23] Sanchez E. S. et al., "**Static and Dynamic Configurable Systems**". IEEE Transactions on Computers, vol. 48, No. 6, pp. 556-566, Junho 1999.

[24] Shan N., "**The Challenges of doing PCI Designs in FPGAs**". XILINX Publications, http://www.xilinx.com, Abril 1998.

[25] Solari E. and Willse G., "**PCI – Hardware and Software, Architecture & Design**". Annabooks, Fourth Edition, 1998.

[26] VCC – Virtual Computer Corporation, "**HOT II Architecture – Technical Bulletin TCN_HOT2_Arch**". Virtual Computer Corporation, Outubro 1998.

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

[28] Vuillemin J. et al., "**Programmable Active Memories: Reconfigurable Systems Come of Age**". IEEE Transactions on VLSI Systems, vol. 4, pp. 56-69, Março 1996.

[29] Wirthlin J. M. and Hutchings B. L., "**DISC: The Dynamic Instruction Set Computer**". John Schewel Editor, Proceedings of the SPIE 2607, pp. 92-103, 1995.

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

[31] XILINX, "**The Real-PCI**". Xilinx PCI Data Book, XILINX, San Jose, CA, Março 1999.

[32] XILINX, "**XC4000E and XC4000X Series Field Programmable Gate Arrays Data Sheet**". Version 1.4, Novembro 1997.

[33] XILINX, "**Virtex Series Configuration Architecture User Guide"**. Disponível por ftp em http://www.xilinx.com/xapp/xapp151.pdf, Setembro 2000.

[34] XILINX, **"Floorplanner Guide"**. Version 3.1i, Xilinx, 2000.

[35] XILINX, **"Libraries Guide"**. Version 2.1, Xilinx, 2000.

ANEXO I – DESCRIÇÕES VHDL COMPLEMENTARES

Código VHDL do *package* com os componentes utilizados no projeto.

```
-- package com os componentes
library IEEE;
use IEEE.std_logic_1164.all;


package PCI_PACK is

    type FSM_TARGET_TYPE is (IDLE, BUSY, READ1, READ_DATA, WRITE_DATA);

    -- Alguns comandos de barramento. Para lista completa referir a pag 175 do livro PCI
    constant C_INTERRUPT_ACK        :STD_LOGIC_VECTOR(3 downto 0) := "0000";
    constant C_SPECIAL_CYCLE        :STD_LOGIC_VECTOR(3 downto 0) := "0001";
    constant C_IO_READ              :STD_LOGIC_VECTOR(3 downto 0) := "0010";
    constant C_IO_WRITE             :STD_LOGIC_VECTOR(3 downto 0) := "0011";
    constant C_MEMORY_READ          :STD_LOGIC_VECTOR(3 downto 0) := "0110";
    constant C_MEMORY_WRITE         :STD_LOGIC_VECTOR(3 downto 0) := "0111";
    constant C_CONFIGURATION_READ   :STD_LOGIC_VECTOR(3 downto 0) := "1010";
    constant C_CONFIGURATION_WRITE  :STD_LOGIC_VECTOR(3 downto 0) := "1011";

    component TARGET is
        port(AD:  in STD_LOGIC_VECTOR (31 downto 0);
                  INT_ADDR:     in STD_LOGIC_VECTOR (31 downto 0);
                  C_BE:         in STD_LOGIC_VECTOR (3 downto 0);
                  CLK:          in STD_LOGIC;
                  FRAME:        in STD_LOGIC;
                  IDSEL:        in STD_LOGIC;
                  IRDY:         in STD_LOGIC;
                  RST:          in STD_LOGIC;
                  HIT:          in STD_LOGIC;
                  ESTADO:       out FSM_TARGET_TYPE
                  );
    end component;

    component PCI is
        port(AD:  inout STD_LOGIC_VECTOR (31 downto 0);
                  INT_ADDR:     in STD_LOGIC_VECTOR (31 downto 0);
                  C_BE:         in STD_LOGIC_VECTOR (3 downto 0);
                  CLK:          in STD_LOGIC;
                  FRAME:        in STD_LOGIC;
                  IRDY:         in STD_LOGIC;
                  IDSEL:        in STD_LOGIC;
                  RST:          in STD_LOGIC;
                  DEVSEL:       out STD_LOGIC;
                  TRDY:         out STD_LOGIC;
                  PAR:          inout STD_LOGIC;
                  SERR:         inout STD_LOGIC;
                  PERR:         inout STD_LOGIC
                  );
    end component;

    component PARIDADE is
        port (
                  CLK:    in STD_LOGIC;                          -- clock do sistema
                  AD:     in STD_LOGIC_VECTOR (31 downto 0);     -- barramento enderecos/dados
                  PAR:    inout STD_LOGIC;                       -- sinal paridade
                  PERR:   inout STD_LOGIC;                       -- sinal parity error
                  SERR:   inout STD_LOGIC;                       -- sinal system error
                  C_BE:   in STD_LOGIC_VECTOR (3 downto 0);      -- command/byte enable
                  ESTADO: in FSM_TARGET_TYPE;                    -- estado da fsm
                  FRAME:  in STD_LOGIC;                          -- sinal inicio ciclo
                  IRDY:   in STD_LOGIC;                          -- sinal initiator ready
                  TRDY:   in STD_LOGIC                           -- sinal target ready
```

```
                );
    end component;

    component BANCO_REG is
        port (
                RST:            in STD_LOGIC;
                CLK:            in STD_LOGIC;
                CBE:            in STD_LOGIC_VECTOR (3 downto 0);
                ESTADO:         in FSM_TARGET_TYPE;
                AD_IN:          in STD_LOGIC_VECTOR (31 downto 0);
                AD_OUT:         out STD_LOGIC_VECTOR (31 downto 0);
                HIT:            out STD_LOGIC;
                END_DEC:        out STD_LOGIC_VECTOR (5 downto 0)
                );
    end component;

    component BANCO_REG_TESTE is                                        -- inicialmente para teste
        port (                                                          -- ja' que estarao no backend
                RST:            in STD_LOGIC;
                CLK:            in STD_LOGIC;
                CBE:            in STD_LOGIC_VECTOR (3 downto 0);
                ESTADO:         in FSM_TARGET_TYPE;
                DECOD:          in STD_LOGIC_VECTOR (5 downto 0);    -- 5 bits para decodificar 32 regs, 1 para os BARs
                DADO_IN:        in STD_LOGIC_VECTOR (31 downto 0);  -- informacao escrita nos regs
                DADO_OUT:       out STD_LOGIC_VECTOR (31 downto 0) -- informacao lida dos regs
                );
    end component;

end PCI_PACK;
```

Código VHDL do par entidade/arquitetura do bloco principal do *core* desenvolvido.

```
------------------------------------------------------------
--          CONTROLADOR PCI                --
------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.PCI_PACK.all;

entity PCI is
    port(C_BE:                  in STD_LOGIC_VECTOR (3 downto 0);
        CLK:                    in STD_LOGIC;
        FRAME:                  in STD_LOGIC;
        INT_ADDR:               in STD_LOGIC_VECTOR (31 downto 0);
        IRDY:                   in STD_LOGIC;
        IDSEL:                  in STD_LOGIC;
        RST:                    in STD_LOGIC;
        DEVSEL:                 out STD_LOGIC;
        TRDY:                   out STD_LOGIC;
        AD:                     inout STD_LOGIC_VECTOR (31 downto 0);
        PAR:                    inout STD_LOGIC;
        SERR:                   inout STD_LOGIC;
        PERR:                   inout STD_LOGIC
        );
end;

architecture arch_1 of PCI is

    signal ESTADO: FSM_TARGET_TYPE;
    signal TRDY_PCI: STD_LOGIC;
    signal HIT: std_logic;
    signal AD_OUT: std_logic_vector (31 downto 0);
    signal AD_OUT_REG_TESTE: std_logic_vector (31 downto 0);
    signal XX : std_logic_vector(7 downto 0);
    signal comando_ant,c1: std_logic_vector(3 downto 0);
    signal END_DEC: STD_LOGIC_VECTOR (5 downto 0);

begin

    U1: TARGET port map(
        C_BE    =>      C_BE,
        CLK     =>      CLK,
        FRAME   =>      FRAME,
        INT_ADDR        =>      INT_ADDR,
```

```
        IRDY    =>      IRDY,
        RST     =>      RST,
        IDSEL   =>      IDSEL,
        AD      =>      AD,
        HIT                     =>  HIT,
        ESTADO          =>      ESTADO);

P1: PARIDADE port map(
        CLK             =>      CLK,
        AD              =>      AD,
        PAR             =>      PAR,
        PERR    =>      PERR,
        SERR    =>      SERR,
        C_BE    =>      C_BE,
        ESTADO  =>      ESTADO,
        FRAME   =>      FRAME,
        IRDY    =>      IRDY,
        TRDY    =>      TRDY_PCI
        );

B1: BANCO_REG port map(
        RST             => RST,
        CLK             => CLK,
        CBE             => C_BE,
        ESTADO  => ESTADO,
        AD_IN   => AD,
        AD_OUT  => AD_OUT,
        HIT             => HIT,
        END_DEC         => END_DEC
        );

B2: BANCO_REG_TESTE port map(
        RST             =>      RST,
        CLK             =>      CLK,
        CBE             =>      C_BE,
        ESTADO  =>      ESTADO,
        DECOD   =>      END_DEC,
        DADO_IN=>       AD,
    DADO_OUT=>          AD_OUT_REG_TESTE
        );



-- geracao de dados por parte da aplicacao do usuario, quando se esta' em
-- modo de leitura
AD <= AD_OUT_REG_TESTE when TRDY_PCI='0' and ESTADO = READ_DATA and comando_ant=C_IO_READ
        else AD_OUT when comando_ant=C_CONFIGURATION_READ and ESTADO=READ_DATA
        else (others=>'Z');

-- geracao dos dados posteriores ao primeiro dado, em leitura BURST
process(RST, CLK )
        begin
        if RST='0' then
                xx <= (others=>'0');
        elsif CLK'event and CLK='0' then
                if  TRDY_PCI='0' and ESTADO = READ_DATA then
                        XX <= XX + 1;
                end if;
        end if;
end process;


-- Quando a FSM estiver nos estados write_data ou read_data o target ready
-- vai a zero (asserted), isto e', o target esta pronto para R/W;
-- Se o master pode subir o sinal IRDY se necessitar de wait states, logo
TRDY_PCI <=  '0' when (ESTADO=WRITE_DATA or ESTADO=READ_DATA) and IRDY='0'
        else '1' when (ESTADO=WRITE_DATA or ESTADO=READ_DATA) and IRDY='1'
        else 'Z';

TRDY<=TRDY_PCI;

-- quando o endereco presente no barramento AD for igual ao endereco interno
-- do target, o devsel vai a zero (asserted), indicando que e' com ele;
DEVSEL <= '0' when ESTADO /= IDLE and ESTADO /= BUSY else 'Z';


process(CLK)
begin
        if(CLK'event and CLK='1') then
```

```
                        if(ESTADO=IDLE) then
                                c1 <= C_BE;
                        end if;
        end if;
    end process;
    process(CLK)
    begin
        if(CLK'event and CLK='0') then
                        comando_ant <= c1;
        end if;
    end process;

end arch_1;
```

Código VHDL do *test-bench*.

```
--------------------------------------------------------------------
--  PPGCC PUCRS - GAPH arquivo: test bench
--  core PCI
--------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.pci_pack.all;

entity TB is
end TB;

architecture arch_tb of TB is

signal CLK, RST, FRAME, IRDY, TRDY, DEVSEL, IDSEL, PAR, PERR, SERR: STD_LOGIC;
signal C_BE: STD_LOGIC_VECTOR (3 downto 0);
signal AD, INT_ADDR: STD_LOGIC_VECTOR (31 downto 0);


begin
   V1: PCI port map (
                        C_BE            =>C_BE,
                        CLK             =>CLK,
                        FRAME           =>FRAME,
                        INT_ADDR        =>INT_ADDR,
                        IRDY            =>IRDY,
                        IDSEL           =>IDSEL,
                        RST             =>RST,
                        DEVSEL          =>DEVSEL,
                        TRDY            =>TRDY,
                        AD              =>AD,
                        PAR             =>PAR,
                        SERR            =>SERR,
                        PERR            =>PERR
                        );

process
begin
    -- clock: periodo 30 ns -> freq 33MHz;
    CLK <= '1', '0' after 15 ns;
    wait for 30 ns;
end process;

    PAR             <= 'H';
    PERR            <= 'H';
    SERR            <= 'H';

    -- reset: ativo baixo;
    RST             <= '0','1'   after 25 ns;

    -- frame: sinal comandado pelo master e indica inicio de transacao no barramento;
    FRAME           <=          '1','0' after  45 ns, '1' after   75 ns,
                                '0' after 135 ns, '1' after  165 ns,
                                '0' after 285 ns, '1' after  585 ns,
                                '0' after 705 ns, '1' after 1015 ns,
                                '0' after 1065ns, '1' after 1095ns;

    IDSEL <=        'Z', '1' after 45 ns, 'Z' after 75 ns, '1' after 135ns, 'Z' after 165ns, '1' after 1065ns, 'Z' after 1095ns;
```

```
    -- command: tipo de comando valido;  byte enable: habilita 1,2,3 ou 4 bytes;
    C_BE<=          x"0",       x"B" after  45 ns, x"0" after  75 ns,
                                x"B" after 135 ns, x"0" after 165 ns,
                                x"3" after 285 ns, x"0" after 315 ns,
                                x"3" after 705 ns, x"0" after 735 ns,
                                x"2" after 1065ns, x"0" after 1095ns;

    -- barramento multiplexado para enderecos (A) e dados (D);
    -- nos cilco de leitura ha' inversao do barramento, pois o master coloca o endereco
    -- e o target coloca o dado a ser lido;

    AD              <=  (others => 'H'),
                    x"40000010" after   45 ns, x"0CCCCCCC" after  75 ns,
                    x"40000014" after  135 ns, x"86868686" after 165 ns, (others => 'H') after 195 ns,
                    x"84868687" after  285 ns, x"ABCDEF01" after 315 ns,
                    x"848686A7" after  705 ns, x"87878787" after 735 ns,
                    x"88888888" after  765 ns, x"89898989" after 795 ns,
                    x"90909090" after 825 ns, x"91919191" after 855 ns,
                    x"92929292" after 915 ns, x"93939393" after 945 ns,
                    x"94949494" after 975 ns, x"95959595" after 1005 ns,(others => 'H') after 1035 ns,
                    x"84868687" after 1065ns, (others=>'H') after 1095ns;

    -- este sinal por enquanto nao esta sendo utilizado. muito importante
    -- coloca-lo na maquina de estados, pois indicara' os wait states por
    -- parte do host;
    -- se o FRAME vai a 1 (deasserted) simultaneamente com a descida
    -- do IRDY (asserted) no primeiro acesso, este acesso sera do tipo single;
    -- se o FRAME permanece em '0' (asserted) apos a descida
    -- do IRDY (asserted) no primeiro acesso, entao este acesso sera' do tipo burst.

    IRDY <=         '0'after  75 ns, '1' after  105 ns,
                    '0' after 165 ns, '1' after 195 ns,
                    '1' after 225 ns, '1' after  255 ns,
                    '0' after 315 ns, '1' after  615 ns,
                    '0' after 945 ns, '1' after 1035 ns,
                    '0' after 1095ns, '1' after 1155ns;

    INT_ADDR        <= x"AAAAAAAA";

    DEVSEL          <= 'H';

    PAR  <=         'Z','1' after  75ns, 'Z' after 105 ns,
                    '0' after 225ns, '1' after 255ns, 'Z' after 285ns,
                    '0' after 315ns, 'Z' after 345ns;

end arch_tb;
```
