Barramento de alto desempenho para interação software/hardware

Ewerton Artur Cappelatti[1], Fernando Gehm Moraes[2], Ney Laert Vilar Calazans[3], Leandro Augusto de Oliveira[4]

[1] Mestre em Ciência da Computação (PUCRS), Engenheiro Eletrônico (PUCRS, 1992). E-mail: ewertonac@feevale.br; [2] Doutor em Informática, opção Microeletrônica (LIRMM, França, 1994), Engenheiro Eletrônico (UFRGS, 1987), Professor Adjunto da Faculdade de Informática/PUCRS. E-mail: moraes@inf.pucrs.br; [3] Doutor em Ciências Aplicadas, opção Microeletrônica (UCL, Bélgica, 1993), Engenheiro Eletrônico (UFRGS, 1985), Professor Titular da Faculdade de Informática/PUCRS. E-mail: calazans@inf.pucrs.br; [4] Bacharel em Informática da PUCRS. E-mail: laolivei@inf.pucrs.br.

Resumo

Este trabalho apresenta o projeto de um módulo de hardware reutilizável, *soft core*, para a implementação do padrão PCI, 32 bits - 33 MHz. A motivação para o desenvolvimento deste *soft core* é prover aos projetistas de hardware um módulo que aumente a largura de banda na iteração *hardware/software*. Apresenta-se 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. A etapa seguinte deste trabalho é a validação do *core* em um ambiente de prototipação, composto de FPGA e barramento PCI.

Palavras-chave

Cores; PCI; FPGAs; prototipação.

Abstract

This paper presents the design of a soft core, for the PCI interface, 32 bits 33 MHz. Our goal is to provide hardware developers with a standard functional block to be used in peripheral boards designs, specially in the context of hardware/software codesign, minimizing the communication bottleneck between hardware and software parts. The paper begins presenting the general characteristics of the PCI interface, followed by the definition, of the core architecture. The core is 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. Current work involves the core validation in a prototyping environment base on FPGAs.

Key words

Cores; PCI; FPGAs; fast prototyping.

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 dedicados (ASICs) têm como características a ocupação mínima de área de silício, altíssimo 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 FPGAs (*Field-Programable Gate Arrays*). 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. Uma classe significativa de FPGAs possui uma estrutura lógica interna que pode ser modificada sempre que necessário. Um FPGA é configurado 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 [1].

O advento dos FPGAs possibilitou retomar o trabalho proposto no início da década de 60 por Gerald Estrin [2]. Estrin propôs um “computador com estrutura fixa e variável”, no qual o *hardware* era dedicado tanto a uma 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 [3]. Infelizmente, os conceitos arquiteturais de Estrin estavam bem à frente da tecnologia à disposição para a sua época, o que só lhe permitiu uma prototipação parcial de sua idéia [4]. 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 foram propostas e desenvolvidas. Entre os exemplos mais significativos estão [5, 6, 7, 8, 9, 10, 11].

CORES

A complexidade atual dos circuitos digitais induz os projetistas de hardware a utilizarem módulos pré-projetados, denominados *cores* [12]. 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. Os *cores* podem ser classificados em 3 categorias:

- *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.
- *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* é o mais difundido hoje, pois representa um bom compromisso entre proteção da propriedade intelectual e desempenho [12].
- *Soft cores* são descritos com o emprego de linguagens de descrição de *hardware*, como VHDL ou Verilog, oferecendo 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 de forma estrita.

COMUNICAÇÃO HARDWARE/SOFTWARE

A maior causa de baixo desempenho em computadores é o tempo gasto em ler/escrever dados em memória e em periféricos. Uma largura de banda de comunicação de dados pequena reduz o desempenho no processamento dos dados. Os sistemas operacionais multitarefas atuais e suas aplicações sofisticadas não requerem apenas processadores rápidos, exigem também uma maior vazão de dados para periféricos, tais como discos rígidos e *hardware* de vídeo. Sendo assim, o tradicional barramento ISA, usado em computadores pessoais, tornou-se um gargalo que se opõe ao aumento do desempenho geral do sistema [14]. Neste contexto, foi desenvolvido o barramento *Peripheral Component Interconnect* - PCI, o qual aumenta da largura de banda provendo uma via para os dados, capaz de transmitir até 528 MByte/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 [13].

OBJETIVOS DO TRABALHO

O trabalho aqui proposto visa o desenvolvimento de um *soft core* PCI para a interação entre *software/hardware*. 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 poderão agregar à sua estrutura estes *cores* 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, por exemplo, a empresa XILINX comercializa um *firm core* PCI por cerca de US$ 10.000,00.

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 uso de tecnologia no estado da arte para projeto, validação funcional, validação de temporização e validação de protótipo.

1. Arquitetura PCI

O PCI é um barramento de alto desempenho, empregado para conectar componentes de uma placa mãe, bem como componentes destas a placas de extensão de um computador. 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*. 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). O termo PNP refere-se à capacidade do sistema computacional de determinar automaticamente os recursos requeridos por 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 as placas de portas serial/paralela e as placas conectadas aos *slots* ISA.

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 *PCI-to-host bridge*. 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.

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, que continua até que o *master* ou o *target* solicitem o final da transferência. O término de uma transação no modo *burst* também pode ser solicitado quando um dispositivo com maior prioridade requisitar o barramento.

1.1 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. Toda operação PCI ocorre entre dois dispositivos distintos: o *master* e o *target*. O *master* é o dispositivo que inicializa 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. 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 49 pinos (todos os pinos obrigatórios).

1.2 DESCRIÇÃO DOS SINAIS

A Figura 1 ilustra os sinais obrigatórios (para *master* e *target*), bem como os sinais para operação em barramento de *64 bits* [13]. 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*.

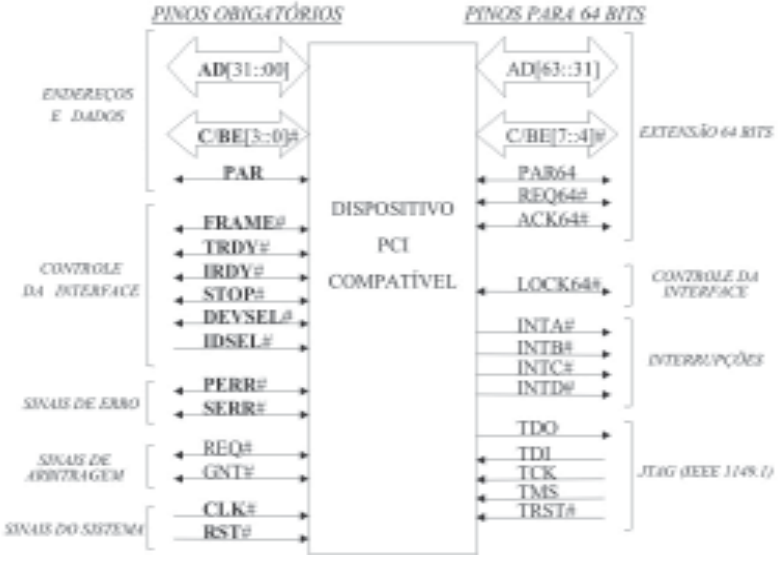

Figura 1 - Sinais presentes em dispositivos compatíveis com PCI, 32/64 bits.

– **CLK**: *Clock* do sistema (33/66 MHz). Os dispositivos PCI são sensíveis à borda de subida do pulso de *clock*.

– **RST#**: Inicializa todos os registradores de configuração. Sinal assíncrono em relação ao *clock* do sistema.

– **AD[31::0]**: 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 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.

– **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 *m*aster do barramento aciona este sinal indicando que um dado válido está disponível no barramento. Durante uma transação de leitura, o *master* do barramento aciona este sinal indicando que o *master* está pronto para receber um dado do *target*. Durante transações de leitura e escrita *wait states* podem ser acrescentados em IRDY# e TRDY#, enquanto estão ativos.

– **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 em IRDY# e TRDY#, enquanto estão ativos.

– **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 exclusivo 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 4.

2. Diagrama de blocos do *core* PCI

A Figura 2 apresenta, em forma de diagrama de blocos, a composição básica de um *core* PCI, representado pelos conteúdos do retângulo pontilhado.

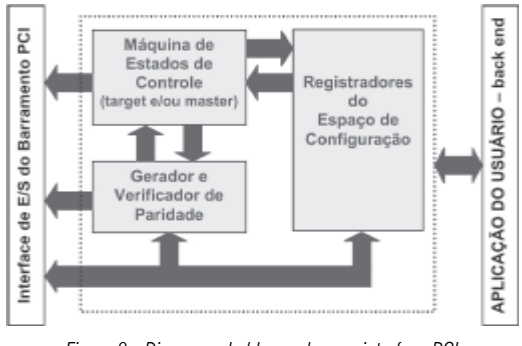

Figura 2 - Diagrama de blocos de uma interface PCI.

Descrição dos módulos:

• *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 6 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*.

Como referência de complexidade de um *core* PCI, pode-se citar o *firm core* fornecido pela empresa XILINX [15]. Este *firm core* consome, para um FPGA Xilinx, da família XC4000E, 152 blocos lógicos e 51/53 pinos de E/S.

3. Implementação dos módulos do *core*

O *core* que está sendo desenvolvido possui é compatível com a especificação PCI para um dispositivo funcionar como *target*. Além destes três módulos, o *core* disponibiliza sinais para aplicações de usuário (dispositivo *back-end*). Este *core* provê, inicialmente, uma *interface* de 32 *bits* a 33 MHz.

3.1 MÁQUINA DE ESTADOS DE CONTROLE

A máquina de estados de controle do *core* operando como *target* está ilustrada, de forma simplificada, na Figura 3. 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 3:

• *IDLE* (S1): O *target* captura os sinais do barramento para determinar quando o *master* está requerendo um acesso à configuração ou I/O de dados. Este é o estado padrão. Quando uma transação inicia, o endereço presente no barramento é comparado com o espaço de endereços do dispositivo PCI. Se este endereço estiver contido neste espaço (HIT='1'), a *interface* 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.

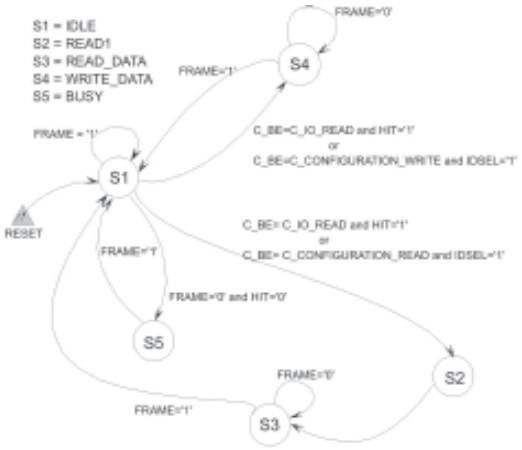

Figura 3 - Máquina de estados de um dispositivo PCI target *compatível.*

3.2 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.

3.3 ESPAÇO DE CONFIGURAÇÃO

Este bloco provê os registradores do *Configuration Space Header* (CSH). A Figura 4 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.

Bytes	Bits			
	31	16	15	0
00h	Device ID		Vendor ID	
04h	Status		Command	
08h	Class Code			Revision ID
0Ch	Built-in Self-Test (BIST)	Header Type	Latency Timer	Cache Line Size
10h	Base Address Register (BAR)			
14h				
18h				
1Ch				
20h				
24h				
28h	Reserved			
2Ch	Reserved			
30h	Expansion ROM Base Address			
34h	Reserved			
38h	Reserved			
3Ch	MAX_LAT	MIN_GNT	Interrupt Pin	Interrupt Line

Figura 4 - Espaço dos registradores de configuração onde se encontra o Configuration Space Header.

3.4 APLICAÇÃO DO USUÁRIO (BACK-END)

O *core* disponibilizará sinais para o acesso de aplicações de usuários ao barramento PCI. Como o objetivo aqui não é implementar uma aplicação do usuário, esta será implementada apenas a título de demonstração. Para este projeto, a aplicação *back-end* será a de ler o estado de chaves e acessar memória da plataforma de prototipação.

3.5 CRITÉRIOS TEMPORAIS

Implementar um controlador PCI sem utilizar um *core* pré definido (*hard core*) é uma tarefa difícil. Além da compatibilidade elétrica, o *soft core* implementado em FPGA deve alcançar todas as especificações temporais para o protocolo PCI. A especificação PCI permite aos componentes operar em freqüências de 0Hz até 33MHz (período do *clock* é de 30 ns) ou até 66MHz [16].

A especificação do barramento PCI sugere uma distribuição dos pinos do componente que implementa 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 [17]. 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.

4. Ciclos básicos de operação PCI em 32 bits

O barramento PCI, de acordo com sua especificação, suporta transferências de dados em modo *burst* (rajada), que consiste 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* 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) [13].

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

4.1 CICLO DE ACESSO DE LEITURA SIMPLES

A Figura 5 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 dipositivo 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[1], 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,

[1] 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.

ao final do ciclo de leitura, é calculada a paridade (sinal PAR).

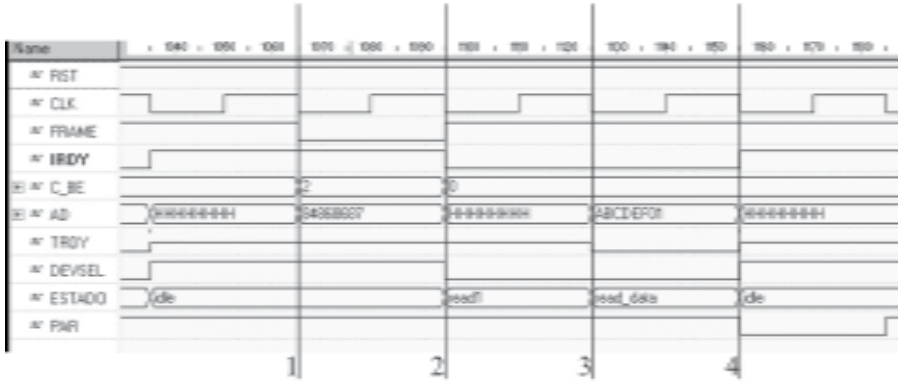

Figura 5 - Sinais do ciclo de acesso de leitura no modo simples.

4.2 CICLO DE ACESSO DE ESCRITA SIMPLES

Os sinais deste ciclo são mostrados na Figura 6. 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.

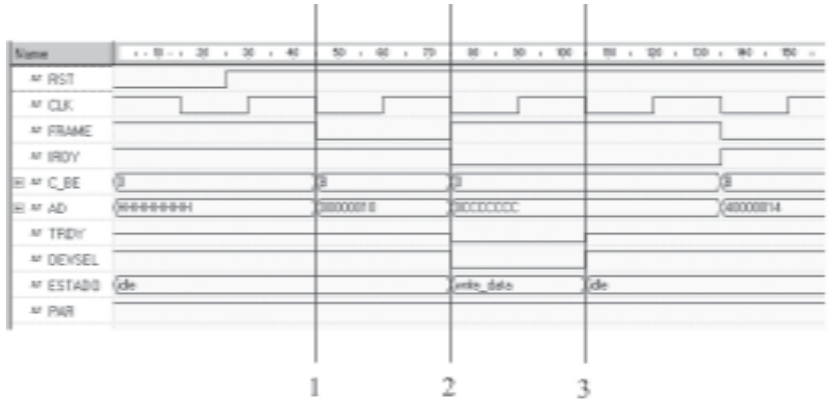

Figura 6 - Sinais do ciclo de acesso de escrita no modo simples.

4.3 CICLO DE ACESSO DE ESCRITA E LEITURA EM MODO BURST

As Figuras 7 e 8 mostram os sinais para um ciclo de leitura em modo *burst* e um ciclo de escrita em modo *burst*, respectivamente. 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 estado da fase de endereçamento permanece durante todo o modo *burst*, e 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 o *byte* a ser acessado em C/BE[3::0]#, e coloca os dados em AD[31::0]. No caso de um ciclo de leitura, coloca as linhas AD[31::0] em alta impedância. Os ciclos de leitura/escrita têm 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.

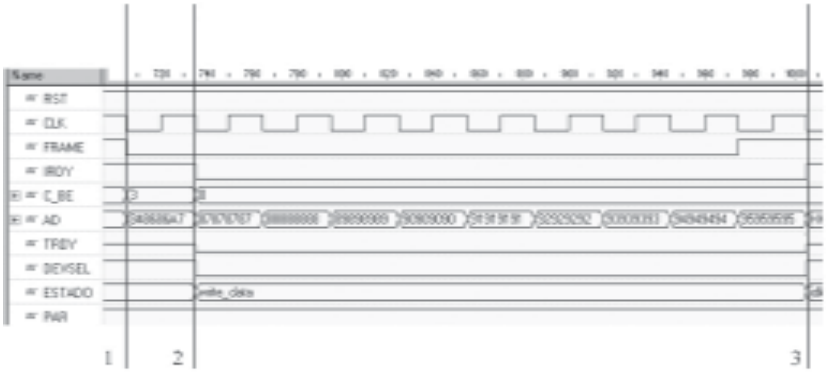

Figura 7 - Sinais do ciclo de acesso de escrita no modo burst. O sinal FRAME# permanece ativo (nível lógico 0) durante toda a fase de escrita.

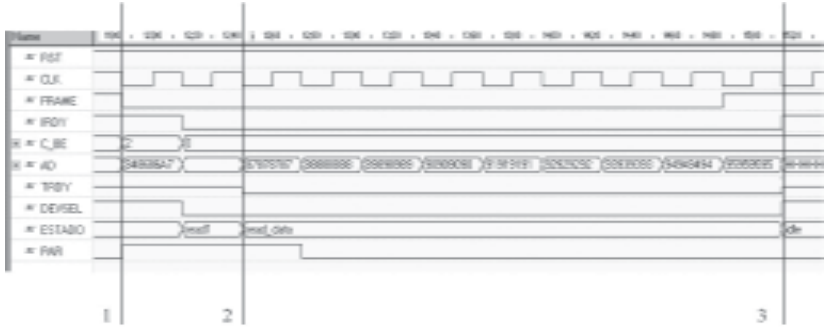

Figura 8 - Sinais do ciclo de acesso de leitura no modo burst. O sinal FRAME# permanece ativo (nível lógico 0) durante toda a fase de leitura.

5. Integração do *core* ao *host*

A etapa final do trabalho será a de integração do *core* ao barramento PCI do computador hospedeiro (*host*). Para tanto, será necessário o desenvolvimento de um *software* que gerencie a comunicação entre o *core* e o barramento PCI.

Existem duas formas de enviar ou receber dados de um dispositivo PCI: através de endereços de I/O (entrada/saída), ou através de I/O mapeado em memória. No caso de endereços de I/O, as operações de leitura ou escrita em um periférico são feitas (no caso da arquitetura Intel) pelas instruções *IN* e *OUT*. Para I/O mapeado em memória, estas operações são executadas através de instruções de manipulação de memória (e.g. *MOV*). Entretanto, a *interface* com um periférico costuma ser mais complexa que o simples envio de *bytes* para o mesmo. Em vista disto, faz-se necessária a utilização de uma camada de *software* que torne mais amigável a *interface* com o dispositivo. Esta camada de *software* é chamada *device driver* (controlador de dispositivo). Para sistemas operacionais que operam em modo real (e.g. MSDOS), esta camada de *software* somente implementa um conjunto de funções (geralmente através de interrupções de *software*) que realiza a comunicação com o dispositivo. Em sistemas operacionais que operam em modo protegido, a função do *device driver* é ampliada. Além de prover uma biblioteca de funções que facilitam o acesso às funções do *hardware* (o que nem sempre é necessário; como exemplo um controlador de porta serial), o *driver* representa a única forma de interagir com o dispositivo. Isso se deve ao fato de que as instruções *IN* e *OUT* não estão disponíveis em modo usuário, ou seja, somente o sistema operacional pode executá-las, pois roda em modo supervisor. Como os *drivers* de dispositivo fazem parte do sistema operacional, eles podem executar estas instruções.

Optou-se implementar o *driver* para o dispositivo que está sendo desenvolvido para o sistema operacional Microsoft Windows NT 4.0, devido à sua disponibilidade e facilidade para obtenção de documentação sobre o mesmo. A construção de *drivers* para NT é feita com um *kit* de software denominado Windows NT Driver Development Kit - DDK. Este *kit* contém as ferramentas necessárias à criação de *drivers*, contendo desde bibliotecas até ferramentas de montagem de *drivers*, *kernel debuggers* e *kernel profilers*.

Conclusão e trabalhos futuros

Este trabalho apresentou a implementação de um módulo de *hardware* reutilizável descrito em VHDL, *soft core*, para o padrão PCI, operando em modo *target*, 32 bits, 33 MHz. Este módulo de *hardware* será utilizado em projetos conjuntos de *hardware* e *software*, minimizando o gargalo de comunicação entre estes. A validação funcional está completa e ocorreu através do emprego do *software* Active-HDL 3.5 (*www.aldec.com*). *Cores*, como o desenvolvido neste trabalho, reduzem o tempo de desenvolvimento, o número de erros, o custo de produção e aumentam a flexibilidade dos sistemas, diminuindo assim o *time-to-market*. A etapa posterior a este trabalho consiste na síntese física em FPGAs, usando o *software* Foundation 3.1i (*www.xilinx.com*) de todos os blocos e o teste de todo o conjunto com o emprego de *drivers* de acesso ao *core* e à aplicação *back-end*. Como ambiente de prototipação será utilizada a placa PCI HOT2-XL (*www.vcc.com*), placas extensoras de barramento PCI, osciloscópio e analisador lógico.

Agradecimentos

Os autores Fernando Moraes e Ney Calazans agradecem o suporte do CNPq (projetos integrados número 522939/96-1 e 522939/96-1) e da FAPERGS (projetos número 96/50369-5 e 94/01340-3).

Referências Bibliográficas

[1] J. Rose et al, “Architecture of Field-Programmable Gate Arrays”. Proceedings of the IEEE, Vol.81, NO. 7, July 1993.

[2] G. Estrin, et al., “Parallel Processing in a Restructurable Computer System”, IEEE Transactions on Electronic Computers, pp. 747-755, December 1963.

[3] J. Villasenor and W. H. Mangione-Smith, “Configurable Computing”. Scientific American, June 1997, pp. 54-59.

[4] W. H. Mangione-Smith et al., “Seeking Solutions in Configurable Computing”. Computer, pp. 38-43, December 1997.

[5] P. M. Athanas and H. F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis”, IEEE Computer, New York, pp. 11-18, March 1993.

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

[7] J. Vuillemin et al., “Programmable Active Memories: Reconfigurable Systems Come of Age”. IEEE Transactions on VLSI Systems, vol. 4, pp. 56-69, March 1996.

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

[9] D. M. Lewis 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, June 1998.

[10] A. Dehon, “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, April 1993.

[11] E. S. Sanchez et al., “Static and Dynamic Configurable Systems”, IEEE Transactions on Computers, vol. 48, No. 6, pp. 556-566, June 1999.

[12] J. Case et al, “Design Methodologies for Core-Based FPGA Designs”, XILINX Publications, http://www.xilinx.com, April 1997.
[13] E. Solari and G. Willse, “PCI – Hardware and Software, Architecture & Design”, Annabooks, Fourth Edition, 1998.
[14] G. W. Kendall, “Inside the PCI Local Bus”, Byte; pp. 177-180; February 1994.
[15] XILINX, “The Real-PCI “, Xilinx PCI Data Book, XILINX, San Jose, CA, March 1999.
[16] N. Shan, “The Challenges of doing PCI Designs in FPGAs”, XILINXPublications,http://www.xilinx.com, April 1998.
[17] K. Kuusilinna, et al., “Field Programable Gate array-based PCI Interface for a coprocessor system”, Microprocessor and Microsystems, vol. 22, pp. 373-388, January 1999.