RAFAEL SCHEFFER VARGAS

SISTEMAS EMBARCADOS: ACOPLAMENTO DO SOFT-CORE PLASMA AO BARRAMENTO OPB DE UM POWERPC 405

Monografia apresentada ao Curso de Ciências da Computação como requisito parcial à obtenção do grau de Bacharel em Ciências da Computação.

Universidade Federal de Santa Catarina.

Orientador: Prof. Dr. Antônio Augusto Fröhlich.

FLORIANÓPOLIS, 2007

RAFAEL SCHEFFER VARGAS

SISTEMAS EMBARCADOS:
ACOPLAMENTO DO SOFT-CORE PLASMA AO
BARRAMENTO OPB DE UM POWERPC 405

Esta monografia foi julgada adequada à obtenção do grau de Bacharel em Ciências da Computação e aprovada em sua forma final pelo Curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Florianópolis – SC, fevereiro de 2007.

_____

Prof. Dr. Antônio Augusto Fröhlich.
Universidade Federal de Santa Catarina

_____

BsC. Hugo Marcondes
Universidade Federal de Santa Catarina

_____

MsC. Rafael Luiz Cancian
Universidade Federal de Santa Catarina

Aos meus pais.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais por confiarem em mim, no meu talento e que sempre me incentivaram quando fraquejei e por me mandar dormir cedo quando eu tinha aula no dia seguinte.

Aos amigos do Lisha, pelas dicas e conselhos que muito úteis foram ao longo deste trabalho. Aos Khompanheiros, pela compreensão, experiência conhecimento que compartilharam comigo.

À minha namorada por compreender e aceitar minhas ausências e nervosismos, e pela grande ajuda na revisão bibliográfica deste trabalho.

Ao meu orientador, pelas idéias que deram rumo a este trabalho, e por compreender meus atrasos.

Não posso esquecer de agradecer também ao Google, por ter todas as respostas.

“Sábio o homem que inventou a cerveja”.
Platão

RESUMO

O crescente avanço das *Field Programmable Gate Arrays* (FPGAs) tornou possível a criação e utilização de *soft-cores*, processadores projetados e implementados em linguagem de descrição de hardware. Porém, a comunicação destes dispositivos com outros componentes reusáveis de hardware (*Intellectual Propertys*, IPs)é limitada quando este não possui uma conexão com nenhum barramento padronizado. O Plasma é um *soft-core* de código aberto, porém, não possui conexão com nenhum barramento, impossibilitando seu uso como componente num sistema-em-chip (SoC). Este trabalho traz grandes vantagens a projetistas de sistemas embarcados, tendo realizado modificações no código do *soft-core* plasma, de forma a transformá-lo num componente reutilizável de hardware, permitindo que o mesmo seja facilmente conectado a um barramento OPB um padrão de barramento em chip da família IBM CoreConnect, para compor um SoC. Os resultados da integração foram validados tanto por simulação quanto pela implementação física do sistema-em-chip numa FPGA, e demonstram a aplicabilidade deste trabalho.

Palavras-chave: Barramento; System-on-a-Chip; CoreConnect; FPGA; VHDL.

ABSTRACT

With the constant advance of Field Programmable Game Array (FPGAs) became possible to create and use soft-cores, hardware description language (HDL) designed and implemented processors. However, when these processors doesn't have a connection to a standardized bus their communication with other reusable hardware blocks (Intellectual Property, IPs) become very restricted. Plasma is an open-source soft-core, however, it doesn't have any connection with a bus. Blocking it's use as a component in a System-on-Chip (SoC). This work brings great advantages to embedded system designers by making modifications in the Plasma soft-core code, being able to make a reusable hardware component from it, allowing the soft-core the be easily connected to an OPB bus, a standard on-chip bus from the IBM CoreConnect Bus Family to build an SoC. The integration results were validated by simulation and by the System-on-chip physical implementation on FPGA, showing this work applicability.

Keywords: Bus; System-on-a-Chip; CoreConnect; FPGA; VHDL.

Sumário

LISTA DE ABREVIATURAS

ALU: Arithmetic and Logic Unit

ASIC: Application Specific Integrated Circuit

BRAM: Block Random Access Memory

CPU: Central Processing Unit

DRAM: Dynamic Random Access Memory

EDK: Embedded Development Kit

FPGA: Field Programmable Gate Array

GPIO: General Purpose Input Output

HDL: Hardware Description Language

IP: Intellectual Property

IRQ: Interrupt Request

LED: Light Emitting Diode

OPB: On-chip Periferal Bus

PC: Program Counter

PPC: PowerPC

RAM: Random Access Memory

RISC: Reduced Instruction Set Computer

SoC: System-on-a-Chip

UART: Universal Asynchronous Receiver-Transmitter

VHDL: VHSIC Hardware Description Language

VHSIC: Very-High-Speed Integrated Circuit

LISTA DE FIGURAS

LISTA DE TABELAS

1 Introdução

Os sistemas embarcados, cada vez mais comuns na sociedade, estão ficando complexos e poderosos graças aos avanços da tecnologia. Estes avanços possibilitam que estes sistemas possuam uma maior capacidade de processamento, memória e adaptação a diferentes necessidades, podendo ser reconfigurados de acordo com a aplicação a qual se destinam.

O aumento desta flexibilidade e complexidade impacta diretamente no esforço de desenvolvimento destes sistemas. Uma das tendências tecnológicas é o desenvolvimento de todo sistema em um único chip. Os SoCs (do inglês, *System-on-a-Chip*), como são chamados, consistem no hardware e também no software que o compõe.

Em sua maioria, SoCs são criados a partir de componentes de hardware pré-validados, agrupados com a utilização de ferramentas CAD, e de drivers que controlam sua operação, desenvolvidos em um ambiente de desenvolvimento de software.

A chave no projeto de SoCs é a simulação, por reduzir os custos totais de desenvolvimento, e é feita utilizando ferramentas de software. Utilizando essas ferramentas pode-se chegar à validação comportamental do sistema sem a necessidade de implementar o mesmo, reduzindo assim, o tempo de projeto e testes, e conseqüentemente, o custo desse sistema. Posteriormente, deve-se validar o sistema sintetizando-o em FPGAs (do inglês, *Field Programmable Gate Arrays*), onde é possível comprovar seu funcionamento numa plataforma física.

SoCs são composições de componentes reutilizáveis de hardware e software. Para a definição dos componentes de hardware, geralmente utilizam-se linguagens de descrição de Hardware (HDL, do inglês *Hardware Description Language*). As mais comuns são VHDL e Verilog, e este trabalho utilizará VHDL, por ser a linguagem no qual o processador *soft-core* escolhido foi implementado.

Processadores *soft-core* são processadores desenvolvidos utilizando linguagens de descrição de hardware para que sejam sintetizados em FPGA. A flexibilidade das FPGAs permite que o projetista do processador possa adaptá-lo aos requisitos da aplicação.

Para permitir a comunicação entre vários componentes de um SoC, utilizam-se barramentos. Barramentos são conjuntos de fios interligando os componentes do sistema promovendo a troca de dados. Alguns barramentos são comumente utilizados na construção de SoCs, como o OPB [3] e o PLB, da arquitetura CoreConnect da IBM [4], e o Wishbone [11], de iniciativa de código aberto.

Este trabalho irá utilizar um processador *soft-core* de código aberto, o Plasma (arquitetura MIPS I), e irá desenvolver a lógica digital necessária para acopla-lo, como mestre e como

escravo, ao barramento OPB, cuja especificação também é livre. Assim, permite-se desenvolver SoCs usando o Plasma como um simples componente reutilizável de hardware, capaz de ser facilmente integrado ao projeto e de comunicar-se com quaisquer outros dispositivos que sejam também compatíveis com este barramento, como memórias, outros processadores e dispositivos de entrada e saída.

1.1 Motivação

Os sistemas embarcados têm importância tecnológica, econômica e social. São utilizados em controladores de vôo, telefones, equipamentos de redes de computadores, impressoras, *drives*, calculadores e eletrodomésticos. As pesquisas nessa área são pertinentes no contexto atual, onde há competição mercadológica, principalmente no ramo de produtos eletrônicos.

Um dos diferenciais dos equipamentos eletrônicos é a sua capacidade de processamento de informação. Quanto mais rápido e de modo mais preciso se obtém um dado, mais fácil é a ação e a gestão do conhecimento. Por isso, as empresas têm investido com freqüência em pesquisas tecnológicas que visem o aumento da capacidade de processamento dos seus sistemas. Em 2006, a Intel anunciou que investira aproximadamente US$ 6 bilhões em pesquisa e desenvolvimento. Segundo dados da IT Web, em 2007, a concorrente Samsung Electronics, uma das maiores fabricantes de processadores, vai investir US$ 1,9 bilhões em processadores.

Os circuitos integrados têm evoluído muito nos últimos 30 anos em 1970 os processadores possuíam poucos milhares de transistores em um chip. Em 2005, este número aumentou para dezenas a poucas centenas de milhões de transistores e, em 2010 a expectativa é que se tenha um total de mais de um bilhão de transistores em um chip.

Esta evolução dos chips permitiu a criação das FPGAs, que são circuitos integrados compostos por portas lógicas que podem se reconfigurar para formar diversos dispositivos diferentes, como processadores, controladores e outros dispositivos que possam ser descritos por lógica.

Esta flexibilidade oferecida pelas FPGAs, tornou possível a implementação de processadores *soft-core* bastante completos. Estes processadores são geralmente construídos utilizando-se blocos de hardware previamente construídos e testados, como memórias, decodificadores e blocos de entrada e saída, os quais são ligados entre si utilizando uma lógica de cola. Estes blocos são comumente chamados de IPs (do inglês *Intellectual Property*)

Um dos *soft-cores* disponíveis no mercado é o Plasma, um processador de código aberto criado em linguagem VHDL e que implementa o conjunto de instruções MIPS I. Esse fato torna possível utilizar o *GNU Compiler Collection* para gerar códigos para o Plasma, o que facilita o desenvolvimento do software.

Este processador, porém, possui uma capacidade limitada de comunicação com o meio externo por não possuir conexão com um barramento padrão. Ele possui duas portas para GPIO (do inglês, *General Purpose Input/Output*),uma para entrada e uma para saída, e um dispositivo de comunicação serial integrado. O que dificulta sua interligação com outros dispositivos.

Torna-se então, interessante do ponto de vista técnico e científico, a existência de um processador de código aberto, que possa ser ligado a um barramento padronizado, conhecido, amplamente utilizado e de especificação aberta, incentivando assim o desenvolvimento de bibliotecas de software para esta arquitetura, bem como a criação de mais dispositivos para esta arquitetura.

Este é o ponto-chave deste trabalho: como utilizar uma tecnologia livre, o Plasma, ligando-o ao barramento OPB, para criar uma alternativa de processamento que incentive futuras pesquisas e contribua para o desenvolvimento tecnológico dos sistemas embutidos.

1.2 Objetivos

1.2.1 Objetivo geral

Este trabalho tem como objetivo transformar o Plasma em um IP reusável, consistindo do processador e um bloco de lógica para a conexão deste ao barramento *On-chip Periferal Bus* (OPB). O Plasma deve poder ser utilizado como processador secundário, conectando-se ao barramento como escravo, ou mesmo servir como processador principal de um sistema, conectando-se ao barramento como mestre.

1.2.2 Objetivos específicos

- Criar um IP que possa ser acoplado ao barramento OPB, sendo que este IP funcione como mestre e também como escravo. A proposta é permitir que o processador possa ler a partir do barramento sempre que necessário, e que o barramento também consiga ler a memória do processador sem a necessidade de interrupção.

- Mapear para o barramento, as portas de GPIO do Plasma, bem como seu sinal de *reset*.
- Alterar a estrutura interna de acesso à memória do Plasma para permitir que um barramento externo acesse sua memória sem atrapalhar o funcionamento do processador.
- Obter um SoC cujo processador principal seja o Plasma, não necessitando do PowerPC gerado pela ferramenta de desenvolvimento para sistemas embarcados da Xilinx, o EDK.

2 Fundamentação Teórica

Serão vistos neste capítulo alguns conceitos e tecnologias fundamentais para o entendimento do trabalho, como o conceito de sistema embarcado, FPGA, VHDL e PowerPC.

2.1 Sistemas embarcados

Sistemas embarcados ou embutidos são aqueles que manipulam dados dentro de sistemas ou produtos maiores, e executam funções específicas. Geralmente são projetados para realizar uma função ou uma gama de funções e não para serem programados pelo usuário final, como os computadores pessoais. Eles geralmente interagem com o ambiente em que se encontram e coletam dados de sensores para modificar o ambiente utilizando atuadores.

Os sistemas embarcados apresentam características em comum com os sistemas computacionais, mas não possuem a mesma uniformidade. Cada aplicação apresenta requisitos diferentes de desempenho, consumo de energia e área ocupada. Isso pode acarretar em uma combinação diferente de módulos de hardware e software para atender a esses requisitos [12]. Os sistemas embutidos podem ser classificados segundo quatro critérios: tamanho, conectividade, requisitos de tempo e interação com o usuário [14].

Sistemas embarcados são usados desde a década de sessenta, mas até a década de oitenta ainda necessitavam de uma série de componentes externos para funcionar, provendo memória de armazenamento e memória volátil. Após a década de 80, com a queda do custo do microcontrolador, os sistemas embarcados puderam substituir componentes físicos, como potenciômetros e capacitores variáveis. Hoje estão em quase todo equipamento eletrônico. Calcula-se que, nos Estados Unidos, há uma média de oito dispositivos baseados em microprocessadores por pessoa. Apesar de esse parecer um número muito grande, basta observar a variedade dos equipamentos eletrônicos: aparelhos de TV e de DVD, tocadores de MP3, telefones, sistemas de alarme, relógios e celulares são alguns dos exemplos de aparelhos que utilizam sistemas embarcados.

2.2 FPGA

Na indústria eletrônica, o tempo de desenvolvimento e de produção deve ser cada vez mais reduzido para limitar o risco financeiro presente no desenvolvimento de um novo produto. Para solucionar essa questão, foi criado o FPGA (*Field Programmable Gate Array*), cuja estrutura pode ser configurada pelo usuário final, o que permite uma rápida fabricação e construção de protótipos a custos muito reduzidos. A escolha dos FPGAs para implementar um co-processador deve-se aos requisitos de alto desempenho, baixo custo e facilidade de reconfiguração.

O FPGA, introduzido em 1985 pela empresa californiana Xilinx Inc. é um circuito integrado específico para construção de sistemas digitais implementado como uma matriz de blocos funcionais em uma rede de interconexão e cercado por blocos de entrada e saída (*I/O blocks*). O termo *field* (campo) indica que a programação do FPGA pode ser feita no lugar de aplicação. É o usuário quem especifica, no momento da programação, a função de cada bloco lógico do sistema e as suas conexões. Na atualidade existem muitos tipos de FPGA lançados por empresas como Actel, Altera, AMD, Crosspoint Solutions, Atmel entre outras [2].

Os FPGAs contêm atualmente mais de 2,5 milhões de portas lógicas. Por isso, desenvolver projetos utilizando apenas diagramas esquemáticos pode ser uma tarefa muito difícil. Os projetistas estão adotando cada vez mais o projeto de FPGAs baseado em HDL (*Hardware Description Language*). A utilização dessas ferramentas permite um aumento de produtividade, pois o projeto pode ser feito em um nível de abstração mais alto (RTL, do inglês *register transfer level*), ao invés de nível lógico booleano ou de portas.

2.3 VHDL

VHDL ou VHSIC Hardware Description Language é uma linguagem de descrição de hardware comumente usada como linguagem de entrada para FPGAs na automação do projeto de circuitos eletrônicos. Criada originalmente para o Departamento de Defesa dos EUA para documentar o comportamento dos ASICs que eram incluídos nos equipamentos que comprava, substituía manuais grandes e complexos.

Posteriormente, foram criados simuladores de hardware para a leitura do VHDL. Também surgiram os sintetizadores lógicos que liam VHDL e tinham como saída uma definição física da implementação do circuito.

A sintaxe do VHDL é pesadamente baseada em ADA, porém seus meios de especificar o paralelismo inerente ao projeto de hardware (processos) são diferentes dos meios de ADA (tarefas).

Existem também outras linguagens para descrição de hardware, como Verilog. Entre as principais diferenças encontradas entre as linguagens, estão:

- VHDL é linguagem fortemente tipada, ao contrário de Verilog;
- em Verilog não há o conceito de pacotes, assim como não existem diretivas para a configuração ou replicação de estruturas;
- o recurso de parametrização em Verilog é inferior, o que torna necessário sobrescrever constantes ao longo do processo de síntese para que, deste modo, os componentes sejam pré-processados corretamente.

A linguagem VHDL pode ser empregada para descrever hardware através das abordagens estrutural e comportamental. Geralmente emprega-se uma mistura das abordagens para formar um projeto por diferentes seções, expressas de diferentes modos.

A linguagem VHDL oferece uma ampla variedade de níveis de abstração e possibilita mesclar esses níveis durante a simulação, tornando possível a adoção de um estilo de projeto verdadeiramente *top-down*. Essa linguagem possui também recursos excepcionais para modelagem de temporização em hardware e provê mecanismos para construção de modelos genéricos.

2.4 PowerPC 405

Este trabalho utiliza um processador PowerPC 405 como processador principal do sistema. Nele é executado o programa inicial, que entre outras tarefas, carrega o programa para a memória do Plasma. Todavia, não é necessário que haja este seja o processador principal, ou mesmo que PowerPC esteja presente no sistema. O Plasma pode ser utilizado como processador principal do sistema, O PowerPC é utilizado neste trabalho pois a ferramenta para sintetizar o sistema facilita a inicialização das memórias com programas para este processador.

A arquitetura PowerPC é uma arquitetura de 64 bits que possui um subconjunto de 32 bits. O PowerPC 405 implementa a arquitetura de 32 bits para ambientes embarcados [13]. Os principais atributos deste processador são:

- Uma unidade de execução com ponto fixo de 32 bits, com trinta e dois registradores de uso geral de 32 bits;

- Extensões à arquitetura para ambientes embarcados como operação *little-endian*, gerenciamento de memória flexível e três *timers* programáveis.
- Atributos de performance como predição de *branch*, *pipeline* de cinco estágios, multiplicação e divisão em *hardware*, suporte a carga e armazenamento não alinhado na memória; e
- Suporte avançado a gerenciamento de memória.

De acordo com a Xilinx (2003), os programas que executem no PPC405 podem o fazer em dois modos de execução: privilegiado e usuário. O modo privilegiado permite que as aplicações acessem todos os registros e executem todas as instruções suportadas pelo processador, normalmente o sistema operacional e *drivers* de dispositivo de baixo-nível executam neste modo. O modo de usuário restringe o acesso a alguns registradores e instruções, normalmente as aplicações de usuário executam neste modo.

O PPC405 também suporta dois modos de endereçamento: o real e o virtual. No modo real, os programas acessam a memória física diretamente. Já no modo virtual, os programas acessam a memória virtual, que é traduzida pelo processador para endereços de memória física, este modo permite acessar espaços de endereçamento muito maiores que os implementados pelo sistema.

2.5 Plasma

Rhoads (2007) é o criador do Plasma, um *soft-core*, um processador que pode ser sintetizado a partir de uma linguagem de descrição de hardware (HDL), neste caso VHDL. Sua vantagem é que ele implementa uma arquitetura de baseada na de von Neumann, utilizando apenas uma memória física para armazenar dados e instruções. Suporta praticamente todas as instruções especificadas pelo MIPS I Instruction Set, documentada por Price (1995), com exceção de apenas duas: carga e armazenamento não alinhado na memória não são suportados, pois são patenteados. Também não são suportadas exceções, apenas interrupções.

O compilador GCC para o MIPS não costuma gerar instruções de acesso não alinhado à memória, uma vez este tipo de acesso tem suporte limitado na maioria das CPUs RISC. Para garantir que no programa não exista nenhuma destas instruções, pode-se gerar uma lista com as instruções utilizadas pelo programa compilado, através do aplicativo objdump, presente no *toolkit* do compilador, e procurar nesta lista pelas instruções: LWL, LWR, SWL ou SWR.

Para gerar esta lista utiliza-se o comando:

```
objdump --disassemble prog.exe > prog.lst
```

sendo `prog.exe`, o nome do programa já compilado, e `prog.lst` o nome do arquivo que será gerado.

2.5.1 Funcionamento do microprocessador

O funcionamento do Plasma baseia-se em um *pipeline* de execução muito parecido com o *pipeline* descrito por Patterson e Hennessy (1997). Este *pipeline* possibilita que mais de uma instrução seja executada ao mesmo tempo, cada uma utilizando unidades funcionais separadas e independentes, uma instrução pode acessar a memória enquanto outra faz uma operação aritmética, por exemplo.

O *pipeline* implementado pelo Plasma é configurável com 4 ou 5 estágios, sendo que quando utiliza o modo de 4 estágios, o quarto estágio é integrado ao terceiro. A execução ocorre por meio dos seguintes passos e ações. Cada instrução necessita de três a cinco dos passos descritos a seguir.

1. **Busca da instrução**

Busca a instrução da memória, apontada por PC, e calcula o endereço da próxima instrução somando quatro ao valor atual de PC.

2. **Decodificação da instrução e busca dos registradores**

São lidos os registradores identificados por `rs` e `rt` (e gravados em registradores temporários A e B, respectivamente), supondo que a instrução seja do tipo R, pois, caso não seja, os valores desnecessários podem ser simplesmente descartados. Também é calculado o possível enderço da próxima instrução, caso esta seja um *branch*, caso a instrução não seja um branch, este valor é ignorado.

3. **Execução, computação do endereço de memória, ou término do *branch***

Neste passo a ALU (do inglês *Arithmetic and Logic Unit*) executa uma de quatro funções, dependendo do tipo da instrução, podendo esta função ser um dos itens abaixo.

- Endereço de memória: o resultado será A somado do valor do campo "imediato" com o sinal extendido para 32 bits.

- Cálculo lógico-aritmético: o resultado será A (operação) B, onde (operação) pode ser qualquer operação da ALU.

- *Branch*: subtrai B de A, e caso o valor seja igual a zero (valores iguais) atualiza o valor de PC para o endereço calculado no ciclo anterior.

- *Jump*: concatena os 4 bits de mais alta ordem do PC com os 26 bits de mais baixa ordem da instrução deslocados 2 bits para a esquerda. E o valor resultante é escrito em PC.

4. Acesso à memória ou término de instrução lógico-aritmética

No caso de acesso á memória o endereço é determinado pela saída da ALU no ciclo anterior. Para leitura, o valor lido é salvo para o próximo ciclo, e para gravação, é gravado o dado de B.

Já no caso de instrução lógico-aritmética (tipo R) o registrador identificado por `rd` é escrito com o valor da saída da ALU no ciclo anterior.

5. Término de leitura de memória

Aqui, o valor lido da memória no ciclo anterior é escrito no registrador identificado por `rd`.

A Figura 1, de Rhoads (2007), ilustra os componentes do Plasma e as ligações entre eles.

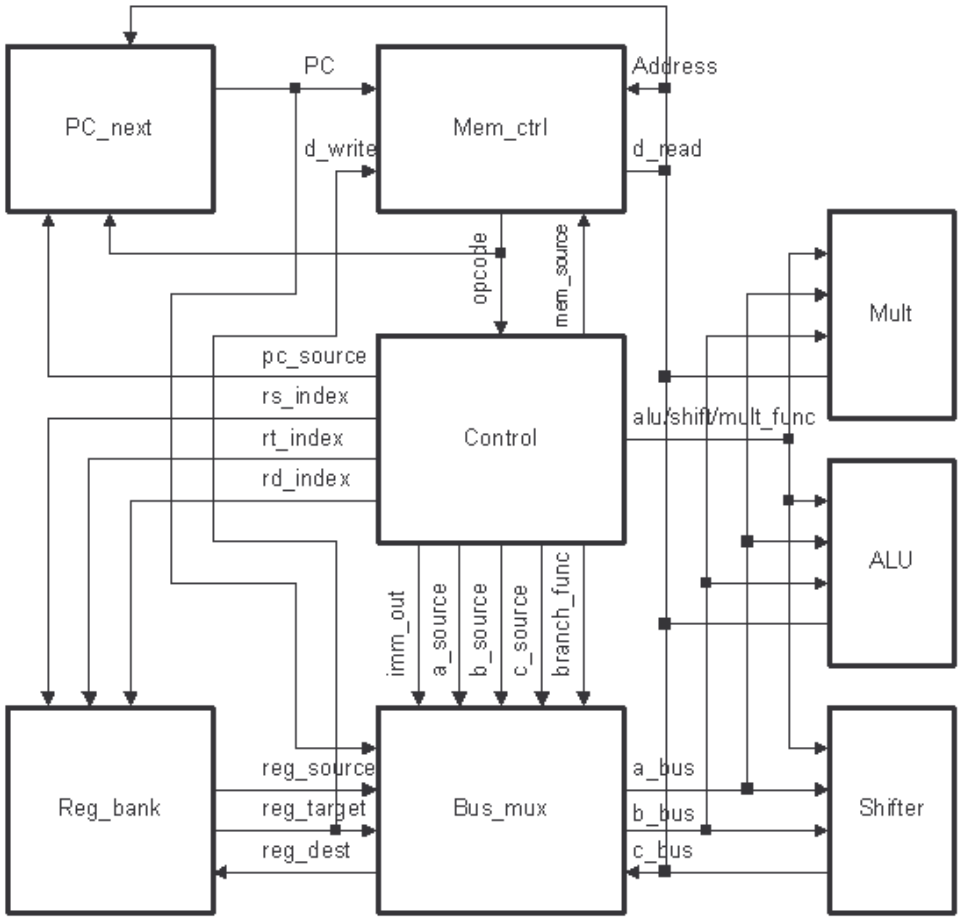

Figura 1: Diagrama de blocos do Plasma.
Fonte: Rhoads, 2007.

2.5.2 Periféricos

Juntamente com o hardware do microprocessador, o Plasma inclui um UART (do inglês *Universal Asynchronous Receiver-Transmitter*), possibilitando a comunicação serial bidirecional. Também agrega um controlador de interrupção e um timer com interrupção.

O acesso a estes periféricos se dá por meio de *programmed I/O*, ou seja, cada periférico possui um endereço-base na memória. Quando se solicita uma leitura ou escrita neste endereço, na realidade, está se solicitando uma leitura no periférico associado a esse endereço. Alguns periféricos, como a memória externa, possuem não apenas um endereço fixo, mas uma área de endereçamento, para que seja possível acessar toda a gama de registradores possíveis do periférico em questão.

A Tabela 1 mostra o mapa de memória com os endereços pré-programados do Plasma, desta forma nota-se a presença dos três periféricos citados e também de duas portas para GPIO, uma de saída e uma de entrada.

Endereço	**Descrição**
0x00000000 – 0x00001FFF	Memória interna (8Kb)
0x10000000 – 0x100FFFFF	Memória externa(1Mb)
0x20000000	UART (escrita e leitura)
0x20000010	Máscara de IRQ
0x20000020	Status do IRQ
0x20000030	GPIO Saída
0x20000050	GPIO Entrada
0x20000060	Contador

Tabela 1: Mapa de memória original do Plasma

Para acessar os registradores de um periférico específico, é necessário especificar o endereço do item desejado (0x20000000, para a UART, por exemplo) e então requisitar uma leitura ou escrita neste endereço. A Figura 2 mostra como exemplo, uma aplicação que verifica o registrador de IRQ para verificar se há dados na UART (*polling*) e, caso haja, estes dados são lidos e então escritos de volta para a saída serial, formando assim um *loopback*.

A UART, tipo de circuito bastante comum, usado nos modems de computadores pessoais, é um módulo geralmente composto de um único circuito integrado que contém, ao

mesmo tempo, os circuitos de transmissão e recepção necessários para as comunicações seriais assíncronas. A UART presente nesta configuração funciona na velocidade de 57600 baud, sendo 8 bits de dados, 0 de paridade e 1 bit de parada.

```
//Endereços de hardware
#define UART                0x20000000
#define IRQ_STATUS          0x20000020

//Bits do IRQ
#define IRQ_UART_READ_AVAILABLE  0x01
#define IRQ_UART_WRITE_AVAILABLE 0x02

#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)

int main()
{
    char c = '0';

    for(;;)
    {
        while(!(MemoryRead(IRQ_STATUS) & IRQ_UART_READ_AVAILABLE))
            ;

        c = (char) MemoryRead(UART);
        MemoryWrite(UART, c);
    }
}
```

Figura 2: Exemplo de programa em C, efetuando *loopback* na UART com *polling*.

O controlador de interrupção presente no Plasma funciona da seguinte maneira, o programa sendo executado escreve no registrador de máscara do IRQ (0x20000010) uma combinação das máscaras de IRQ existentes, descritas na Tabela 2. Quando o conteúdo do registrador de status do IRQ corresponder à máscara definida (o resultado de uma operação “E” com os dois registradores for diferente de zero), o tratador de interrupções é invocado (endereço 0x0000003C). O tratador de interrupções salva todos os registradores temporários e o PC e chama o tratador de interrupções definido pelo sistema operacional. Após o término da execução deste tratador, os registradores salvos são restaurados e o programa que foi interrompido volta a ser executado a partir da instrução que estava sendo executada no momento em que ocorreu a interrupção.

A Figura 3 demonstra o mesmo exemplo da Figura 2, com a diferença que utiliza interrupções. Neste exemplo, é escrito no registrador de máscara do controlador de interrupções, uma máscara com as interrupções que se deseja receber, e então o programa faz um laço infinito, apenas para esperar as interrupções. O tratador de interrupções ilustra o teste do conteú-

do do registrador de *status* de IRQ, identificado pela variável `status`, pela presença da máscara requerida. Este procedimento é importante, pois se pode registrar mais de uma interrupção e todas chamarão a mesma função. Após a confirmação da máscara, o dado é lido da UART e então escrito novamente para ser reenviado.

Bits do IRQ	**Descrição**
7	GPIO[31]
6	GPIO[30]
5	not GPIO[31]
4	not GPIO[30]
3	Contador[18]
2	not Contador[18]
1	not UART ocupada para escrita
0	Dados disponíveis na UART

Tabela 2: Descrição dos bits da IRQ.

```
//Endereços de hardware
#define UART                0x20000000
#define IRQ_MASK            0x20000010
#define IRQ_STATUS          0x20000020

//Bit do IRQ
#define IRQ_UART_READ_AVAILABLE  0x01

#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)

void OS_InterruptServiceRoutine(unsigned int status)
{
    char c = '0';
    if( status & IRQ_UART_READ_AVAILABLE)
    {
        c = (char) MemoryRead(UART);
        MemoryWrite(UART, c);
    }
}

int main()
{
    MemoryWrite(IRQ_MASK, IRQ_UART_READ_AVAILABLE);

    for(;;)
        ;
}
```

Figura 3: Exemplo de programa em C, efetuando *loopback* na UART com interrupção

2.6 O barramento OPB

Barramentos são um conjunto de sinais que permite a comunicação entre componentes de um sistema computacional, como memória, processadores e demais periféricos. Eles conectam as diferentes partes para que possa ocorrer transferência de dados entre eles. A IBM criou um padrão de barramentos para SoCs, o CoreConnect, que é segmentado em três partes interconectadas:

- PLB - *Processor Local Bus*;
- OPB - *On-chip Periferal Bus*; e
- DCR - *Device Control Register*.

De acordo com a IBM (1999), o objetivo principal do PLB é prover conexões de baixa latência e alta velocidade. O do OPB é prover conectividade flexível a periféricos de diferentes tamanhos de palavra e tempos de acesso a dados, com um mínimo de impacto no PLB. O DCR é um mecanismo de transferência de dados entre os registradores de diferentes componentes.

Este trabalho utilizará o barramento OPB. Este barramento foi escolhido entre os demais, pois sua interface e funcionamento são simples, é também bastante flexível. Fato que possibilitaria uma implementação eficiente e que tivesse baixo custo em relação a espaço na FPGA. Há também uma grande gama de periféricos compatíveis com este barramento, fato que aumenta ainda mais a diversidade das aplicações possíveis.

O barramento de periféricos *on-chip* (OPB) foi projetado para permitir uma conectividade fácil entre dispositivos periféricos em um mesmo chip. Segundo dados da IBM (2001), são as principais características do OPB:

- endereçamento de até 64 bits;
- barramento de dados de 32 ou 64 bits;
- totalmente síncrono;
- suporte a escravos de 8, 16, 32 ou 64 bits;
- suporte a mestres de 32 ou 64 bits;
- definição de tamanho dinâmico, com transferências de *byte, halfword, fullword* e *doubleword*;
- suporte a protocolo de endereços seqüenciais;
- *timeout* de 16 ciclos provido pelo árbitro de barramento; e
- supressão de *timeout* pelo escravo.

A quantidade máxima de mestres e de escravos deste barramento depende da implementação, a especificação recomenda que, caso a aplicação necessite de muitos mestres, sejam criados vários segmentos do barramento. Não existem limites lógicos para a quantidade de escravos. O principal limite seria a carga elétrica. Novamente, múltiplos segmentos podem resolver o problema.

A Tabela 3 ilustra os principais sinais utilizados pelos componentes conectados ao OPB.

Sinal	**Interface**	**Direção**	**Descrição**
Mn_request	Mestre	Saída	Requisição do barramento
Mn_grant	Mestre	Entrada	Resposta de requisição
Mn_select	Mestre	Saída	Indicação de utilização do barramento do mestre
Mn_ABus	Mestre	Saída	Barramento de endereço do mestre
Mn_DBus	Mestre	Saída	Barramento de dados do mestre
OPB_xferAck	M/E	Entrada	Transferência completa
OPB_select	Escravo	Entrada	Indicação de utilização do barramento
OPB_ABus	Escravo	Entrada	Barramento de endereço
OPB_DBus	M/E	Entrada	Barramento de dados
OPB_RNW	Escravo	Entrada	*Read not write*
Sln_DBus	Escravo	Saída	Barramento de dados do escravo
Sln_DBusEn	Escravo	Saída	Habilita o barramento de dados do escravo
Sln_toutSup	Escravo	Saída	Supressão de *timeout*
Sln_xferAck	Escravo	Saída	Transferência completa do escravo

Tabela 3: Principais sinais do OPB
Fonte: IBM, 2001

2.6.1 Operações no barramento

As operações no OPB foram projetadas para serem simples. Um mestre que deseja efetuar uma transferência de dados solicita esta através de seu sinal *request* (passando seu valor para ‘1’), e pode iniciar a transferência assim que receber a confirmação do árbitro pelo seu sinal *grant*. A transferência se dá pelo envio do sinal *select* e conclui-se ao receber o sinal

xferAck. O mestre pode permanecer com o sinal de request ligado e continuar a fazer transferências enquanto o valor de *grant* se mantiver em '1'.

Quando um escravo percebe o sinal de *select* do barramento e identifica que o endereço (sinal *ABus*) corresponde ao seu *address space*, ele habilita o barramento de dados (sinal *DBus*) através do sinal *DBusEn*, e quando os dados estiverem prontos envia o sinal *xferAck*.

Um escravo cuja latência para acessar os dados requeridos seja maior que 16 ciclos, deve sinalizar tal condição utilizando o sinal de supressão de *timeout*, (*toutSup*), evitando que sua demora para responder seja tratada como erro.

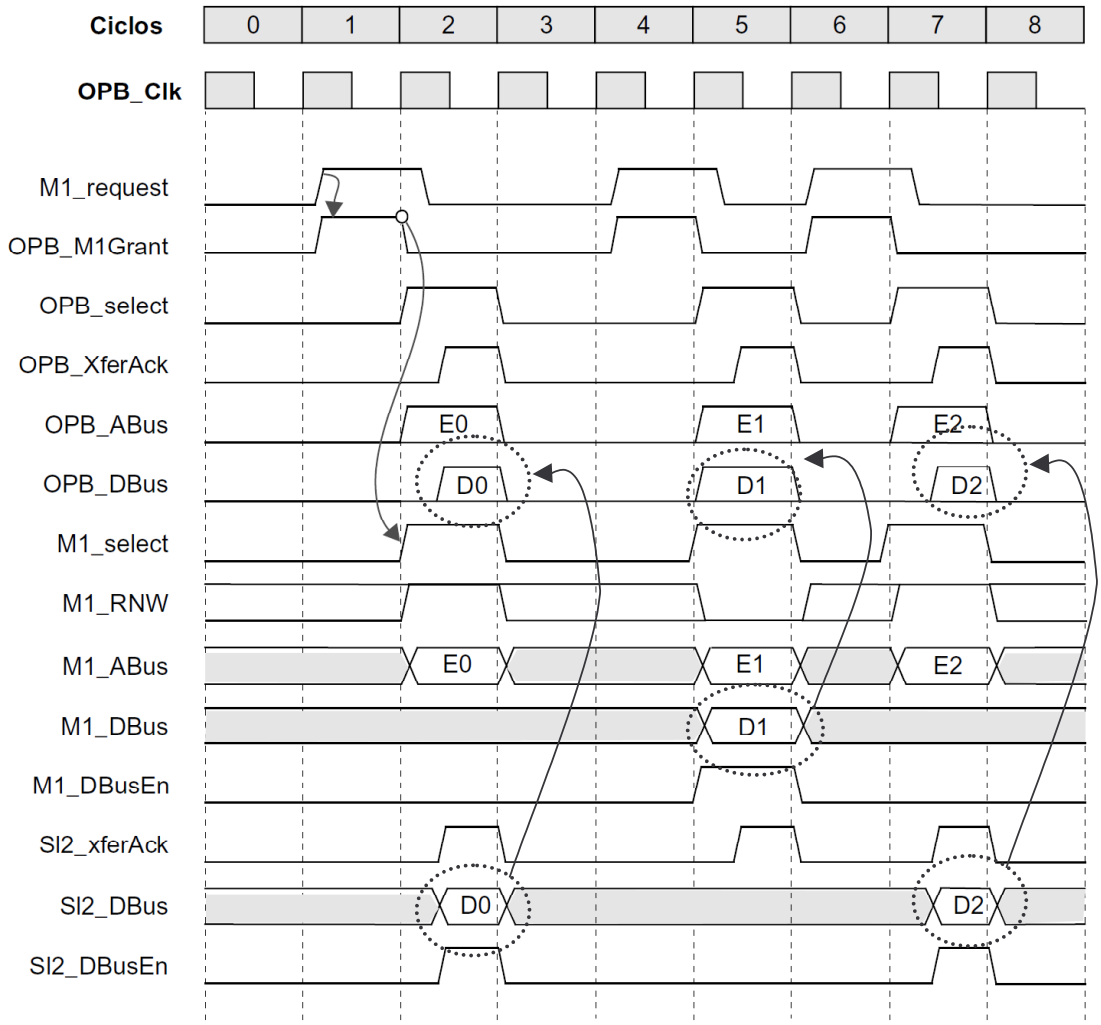

Figura 4: Transferência de dados no barramento OPB
Fonte: IBM, 2001.

A Figura 4 ilustra a transferência de dados utilizando o barramento OPB, com um mestre (M1) e um escravo (Sl2) realizando três transferências, duas leituras uma escrita. Afi-

gura demonstra a multiplexação do sinal *OPB_Dbus* entre os participantes das transferências. É importante notar também na figura a utilização do sinal *M1_RNW* para descrever quando o acesso é uma leitura (sinal alto) ou escrita (sinal baixo). E também utilização dos barramentos de endereços e dos barramentos de dados e a importância dos sinais que habilitam o barramento de dados. (*DBusEn*).

3 Projeto do IP

Para o projeto do IP, adotou-se uma metodologia *top-down*. Consistindo então na elaboração de requisitos de alto nível para o sistema e a partir desta versão efetuar sucessivos refinamentos nestes, até que se tenha obtido um IP sintetizável.

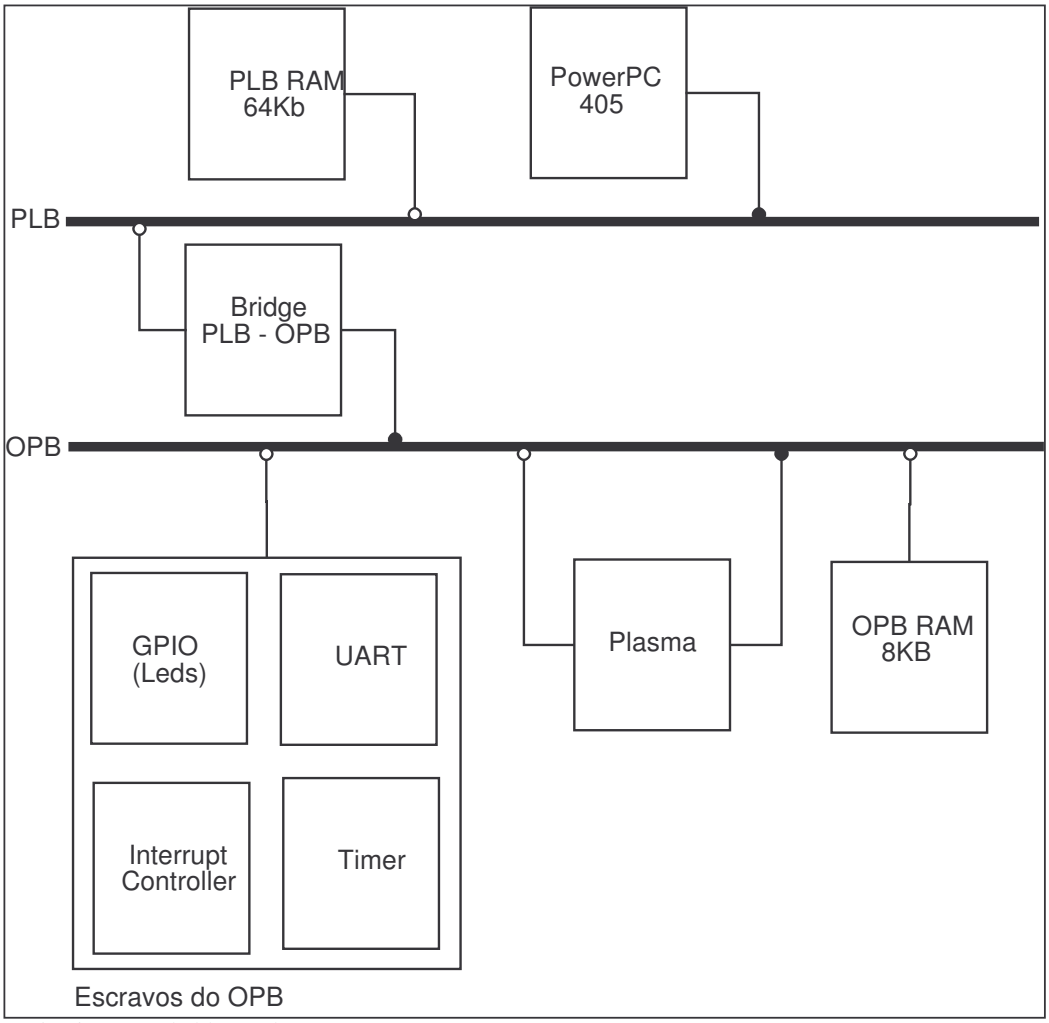

Figura 5: Diagrama de blocos do SoC

A Figura 5 ilustra o diagrama de blocos do System-on-a-chip a ser criado. Este será composto utilizando a ferramenta de projeto de sistemas embarcados da Xilinx, o EDK. Neste sistema, foi criada a seguinte configuração:

- um processador PowerPC;
- um barramento PLB;

- 64Kb de memória BRAM (do inglês *Block Random Access Memory*, bloco de memória interno à FPGA);
- 8Kb de memória BRAM.
- um barramento OPB;
- uma *brigde* PLB-OPB para possibilitar a comunicação de mestres no PLB com escravos no OPB;
- um Timer;
- um dispositivo de comunicação serial (UART);
- um controlador de interrupção;
- uma porta de GPIO, ligada aos Leds do kit; e
- o Plasma.

Na figura, as ligações com o barramento são demonstradas pelas linhas ligando os blocos ao barramento. As esferas na conexão com o barramento denotam o tipo da ligação com o mesmo. As esferas preenchidas denotam uma conexão do tipo mestre e as esferas vazias, uma conexão do tipo escravo.

Neste projeto, foi utilizado o PowerPC com a freqüência de 100MHz, o clock tanto do barramento PLB quanto to barramento OPB foram definidos para 100MHz. O software de análise de desempenho da Xilinx (executado durante a síntese), verificou que a freqüência máxima do *clock* para o Plasma deveria ser de aproximadamente 65MHz, optou-se então por inserir um nível de divisão de *clock* no IP. O valor pelo qual o *clock* deverá ser dividido é um parâmetro de instanciação do IP (*generic*).

3.1 Escolha dos componentes

Para desenvolver o projeto, inicialmente foram escolhidos os elementos necessários na construção do sistema com co-processador, os quais eram:

- circuito integrado (FPGA);
- dispositivo de comunicação com a FPGA; e
- software para interpretação da linguagem e síntese do hardware.

A seguir são detalhados os requisitos de cada um desses elementos bem como informados quais componentes foram escolhidos.

3.1.1 Circuito integrado (FPGA)

Para a realização do projeto é necessária a utilização de um FPGA que comporte o sistema completo.

Este *chip* é uma matriz de portas lógicas, cujo número é limitado, de acordo com o modelo. A princípio, foi cogitado o uso da Xilinx Spartan-3, contida no kit de desenvolvimento *Spartan-3 starter kit*. Porém, após alguns testes, verificou-se que só o Plasma ocupava 90% da capacidade. Desta forma, optou-se pelo uso de uma FPGA com maior capacidade.

O modelo escolhido foi a Xilinx Virtex II Pro, contida no kit de desenvolvimento ML 310. Neste segundo modelo, o Plasma ocupou 11% da capacidade e, então, observou-se que com este modelo seria possível sintetizar o sistema com o PowerPC, o barramento e o Plasma.

3.1.2 Dispositivo de comunicação com a FPGA

São necessários dois tipos de comunicação: um para verificar os resultados através da saída do sistema e outro para programar a FPGA.

Para observar os resultados, foi utilizado o cabo serial RS-232, pois é um modelo facilmente encontrado no mercado e que permite uma comunicação simples, eficiente e bilateral. Sua implementação não é complexa.

Para programar a FPGA normalmente utiliza-se um cabo JTAG. De acordo com o manual do kit de desenvolvimento da Xilinx ML 310, observou-se que o único modelo compatível é o JTAG4. No entanto, não foi possível obter este cabo e, na ausência dele, foi decidido que a programação dar-se-ia através do cartão *compact flash*, presente no kit.

3.1.3 Software para interpretação da linguagem e síntese do hardware

Para programar a FPGA utiliza-se uma linguagem de descrição de hardware, geralmente VHDL ou Verilog. Neste projeto, optou-se pela VHDL, por ser a linguagem em que o Plasma estava implementado e devido ao conhecimento prévio do autor do projeto.

A transformação da VHDL no formato de entrada da FPGA é feita pelo software de síntese. Neste projeto foi escolhido o pacote de desenvolvimento da Xilinx, o ISE (do inglês, *Integrated Software Environment*), por ser compatível com a FPGA escolhida e por estar disponível gratuitamente na Web.

Além desse software, foi utilizado também o EDK (do inglês *Embeded Development Kit*), um kit de desenvolvimento da Xilinx para sistemas embarcados, no qual foi possível gerar o esqueleto do sistema descrito anteriormente.

3.2 Modificação do Plasma

Após a definição dos elementos necessários, o Plasma foi sintetizado na FPGA Virtex II Pro, presente no Kit de Desenvolvimento ML-310, deste kit foi utilizado, além da FPGA, a interface de memória *compact flash* que foi utilizado para programar a FPGA, oito leds de uso geral e o dispositivo de comunicação serial RS-232. O Kit possui também um controlador PCI, no qual se ligam diversos outros periféricos, que não foram utilizados por este trabalho.

3.2.1 Acesso à memória interna

Para que fosse possível acessar a memória interna do Plasma, foi cogitada a implementação de um bloco de lógica que promovesse o duplo acesso à memória existente. Procedeu-se da seguinte forma, dividiu-se o *clock* do Plasma por dois, enquanto a memória continuava com o *clock* original, alternando os acessos do Plasma com os acessos do barramento externo. Este processo, porém, acarretou em uma grande perda de desempenho por parte do processador.

A alternativa encontrada foi a de modificar o componente de RAM do processador, trocando a instanciação de quatro blocos de memória de 2048 bytes com uma porta de acesso (componente `RAMB16_S9`) pela instanciação de quatro blocos de memória de 2048 bytes com duas portas de acesso (componente `RAMB16_S9_S9`) e a externalização dos sinais da porta B de acesso à memória. A Figura 6 demonstra a interface final da memória interna do Plasma.

```
entity ram is
   generic(memory_type : string := "DEFAULT");
   port(clk                 : in  std_logic;
        clk_b               : in  std_logic;
        enable              : in  std_logic;
        enable_b            : in  std_logic;
        write_byte_enable   : in  std_logic_vector(3 downto 0);
        write_byte_enable_b : in  std_logic_vector(3 downto 0);
        address             : in  std_logic_vector(31 downto 2);
        address_b           : in  std_logic_vector(31 downto 2);
        data_write          : in  std_logic_vector(31 downto 0);
        data_write_b        : in  std_logic_vector(31 downto 0);
        data_read           : out std_logic_vector(31 downto 0);
        data_read_b         : out std_logic_vector(31 downto 0));
end;
```

Figura 6: Interface do componente de memória do Plasma.

Constata-se a existência das duas portas de acesso á memória, verificando que todas as portas estão duplicadas (com o sufixo "_b"). Nota-se também, que a memória pode ser acessada por dois dispositivos completamente diferentes mesmo que estes tenham fontes diferentes de *clock*, pois existem duas entradas para o sinal de sincronismo, cada uma correspondente à sua porta.

Por fim, adicionou-se à interface externa do Plasma mais seis sinais, correspondentes aos sinais adicionais criados no componente de memória, são eles:

- **clk_bus**: entrada, o sinal de *clock* do barramento;
- **enable_bus**: entrada, pedido para acesso à memória;
- **write_byte_enable_bus**: entrada, autorização do para escrita na memória;
- **address_bus**: entrada, endereço que deve ser lido ou escrito;
- **data_write_bus**: entrada, dado que deve ser escrito, se for o caso; e
- **data_read_bus**: saída, dado que foi lido.

3.2.2 Interface de memória externa

Além de acessar a memória interna, os programas que estiverem executando no Plasma, podem acessar também endereços de memória que não estejam dentro desta área de memória. Como foi descrito anteriormente, na Tabela 1, o Plasma mapeia a memória externa para os endereços cujo byte mais significativo seja igual a 0x10. Porém seria necessário permitir aos aplicativos que acessem dispositivos que estivessem em outros mapeamentos de memória, como no intervalo 0x60000000 – 0x6000FFFF, por exemplo.

Para permitir este acesso, foi modificada a lógica de controle de memória do Plasma para que este possua o mapa de memória descrito na Tabela 4:

Endereço	**Descrição**
0x00000000 – 0x00001FFF	Memória interna (8Kb)
0x10000000 – 0x10FFFFFF	Memória externa
0x20000000	UART (escrita e leitura)
0x20000010	Máscara de IRQ
0x20000020	Status do IRQ
0x20000030	GPIO Saída
0x20000050	GPIO Entrada
0x20000060	Contador
0x30000000 – 0xFFFFFFFF	Memória externa

Tabela 4: Mapa modificado de memória do Plasma

A Figura 7 ilustra a implementação deste mapa de memória, no qual três dos quatro bits mais significativos do endereço são testados e de acordo com o resultado, o dado lido provém de uma fonte diferente, demonstra também, a decodificação do endereço dos registradores de controle do Plasma.

3.3 Lógica de acoplamento ao barramento OPB

O Plasma ao ser acoplado ao barramento irá realizar operações leitura e escrita no OPB. Na sua arquitetura, o Plasma possui uma memória interna de 8Kb e uma interface de memória externa. É nesta interface de memória externa que o Plasma solicitará leituras e escritas no barramento. E em sua memória interna o barramento poderá solicitar leitura e escrita, possibilitando assim que os programas que devem ser executados sejam passados à memória do microprocessador.

```
misc_proc: process(clk, reset, mem_address, address_reg, enable_misc,
   data_read_ram, data_read, data_read_uart, mem_pause,
   irq_mask_reg, irq_status, gpio0_reg, write_enable,
   gpioA_in, counter_reg, mem_data_write, data_write_reg)
begin
   case address_reg(30 downto 28) is
   when "000" =>      --internal RAM
      mem_data_read <= data_read_ram;
   when "010" =>      --misc
      case address_reg(6 downto 4) is
      when "000" =>      --uart
         mem_data_read <= ZERO(31 downto 8) & data_read_uart;
      when "001" =>      --irq_mask
         mem_data_read <= ZERO(31 downto 8) & irq_mask_reg;
      when "010" =>      --irq_status
         mem_data_read <= ZERO(31 downto 8) & irq_status;
      when "011" =>      --gpio0
         mem_data_read <= gpio0_reg;
      when "101" =>      --gpioA
         mem_data_read <= gpioA_in;
      when "110" =>      --counter
         mem_data_read <= counter_reg;
      when others =>
         mem_data_read <= gpioA_in;
      end case;
   when others =>    -- external RAM
      mem_data_read <= data_read;
   end case;

   -- ...

end process;
```

Figura 7: Implementação do novo mapa de memória do Plasma

Devido à natureza das operações que o processador realiza, é necessário que este atue no barramento OPB, tanto como mestre, nas operações nas quais ele requisita leitura ou escrita nos periféricos conectados a este barramento através da interface de memória externa; quanto como escravo, nas operações nas quais algum mestre do barramento solicita a leitura ou escrita na memória interna do Plasma. Foi então necessário implementar um módulo que pudesse:

- acoplar-se ao barramento OPB como mestre e como escravo;
- conectar-se à interface de memória externa do Plasma;
- acessar a memória interna do Plasma, tanto para leitura quanto para a escrita; e
- mapear para o barramento, as portas de GPIO do Plasma e o sinal de *reset*.

3.3.1 Implementação

Duas foram as abordagens utilizadas para implementar a lógica de acoplamento ao barramento. Utilizar o assistente de criação de periféricos do EDK, e a implementação "*from scratch*" da lógica.

A primeira abordagem foi utilizar o assistente de criação de periféricos da ferramenta da Xilinx. Esta ferramenta cria um esqueleto de um periférico a ser ligado a um dos barramentos suportados pela Ferramenta (OPB ou PLB).

Este esqueleto gerado pela ferramenta utilizando os requisitos deste sistema gerou um dispositivo relativamente simples, possuindo apenas uma interface que conecta-se ao barramento como mestre/escravo. Possuindo dois arquivos, esta implementação consiste de uma interface primária, que recebe todos os sinais e os envia a um IP de controle da Xilinx, que traduz os sinais do barramento para os sinais de controle de mais dois componentes, chamados de "conexão mestre" e "conexão escravo". Os sinais de saída destes componentes eram então mapeados nos sinais do segundo arquivo, chamado de "lógica do usuário".

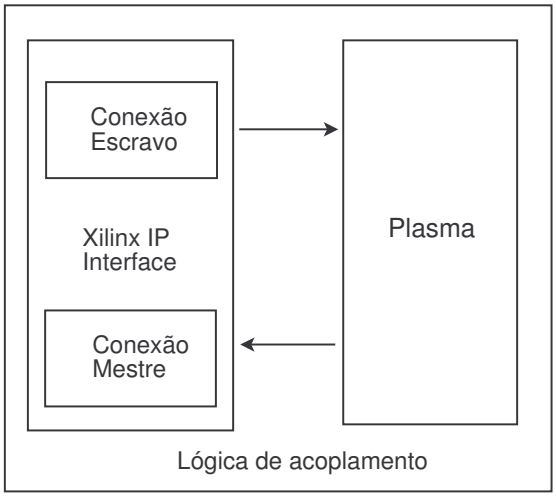

Figura 8: Diagrama de blocos da primeira abordagem ao IP

Esta arquitetura, demonstrada pela Figura 8, fez com que o IP executasse corretamente como escravo no sistema. A conexão mestre, acessa somente os mesmos registradores da conexão escrava. Para executar as funções de mestre, é necessário que outro mestre do barramento escreva nos registradores de controle do IP os seguintes dados:

- Tipo de operação: leitura ou escrita;
- Endereço na conexão escrava;
- Endereço no barramento; e

- Tamanho da operação, em bytes.

Deste modo, quando executada a operação, ocorre que, ao requisitar uma escrita no barramento, é executado uma leitura na conexão escrava e então uma escrita no barramento; e no caso de uma leitura no barramento, é feita a leitura e então é efetuada uma escrita na conexão escrava.

Após a síntese, o IP, seguindo esta primeira abordagem, contendo somente a lógica de acoplamento e o Plasma, ocupou 48% da capacidade da FPGA.

A segunda abordagem, implementação "*from scratch*" da lógica de acoplamento, ilustrada pela Figura 9, foi implementada como a entidade de mais alto nível (*top level entity*) do IP, tornando-se assim a interface do Plasma com qualquer elemento exterior. Composta por duas interfaces separadas, a interface mestre e a interface escrava, sendo que as duas compartilham os sinais de *clock* e *reset*.

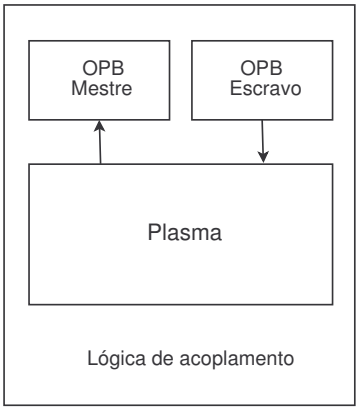

Figura 9: Diagrama de blocos da segunda abordagem ao IP

A Figura 10 ilustra a interface da lógica de acoplamento, onde nota-se a externalização dos sinais de entrada e saída da UART do Plasma, bem como os sinais das interfaces mestre e escravo do barramento. Nota-se também que é possível configurar alguns aspectos do funcionamento da lógica, como, por exemplo, a quantidade de vezes que deve tentar enviar um dado caso seja necessário, assim como os endereços inicial e final do espaço de endereçamento.

Foi estabelecido então, que haveria três espaços de endereçamento dentro da interface escrava do barramento, uma para a memória, uma para as portas de GPIO, e uma para o sinal de *reset* do Plasma. A Tabela 5 ilustra este mapa de memória, levando

para o sinal de *reset* do Plasma. A Tabela 5 ilustra este mapa de memória, levando em consideração somente os 16 bits menos significativos.

Espaço de endereçamento	**Descrição**
0x0000 – 0x1FFF	Memória interna (8Kb)
0x2000 – 0x2FFF	GPIO
0x3000 – 0x3FFFF	Reset

Tabela 5: Mapeamento das interfaces do Plasma no barramento

Deste modo, quando algum mestre do barramento solicita uma leitura a este componente, o endereço recebido é traduzido, escolhendo uma das três portas, sendo então repassado este endereço ao controlador de memória, no caso da primeira porta ou descartado no caso das outras. O dado lido é então repassado ao barramento. Para escrita o processo é similar, com a diferença que após o repasse do endereço o dado proveniente do barramento é escrito na memória.

A interface escrava do barramento possui um atraso de três ciclos, ou seja, o sinal de confirmação (*Sl_xferAck*) de escrita ou leitura só é enviado no terceiro ciclo após o recebimento do sinal de habilitação do barramento (*OPB_select*).

O lado mestre do barramento, conectado à interface de memória externa do *soft-core*, fora implementado de modo que, quando o Plasma solicitar uma leitura na memória, enviando um valor válido para o endereço, será feita uma solicitação de leitura no barramento e, da mesma forma, quando, além de um valor válido para o endereço, for enviado um valor diferente de ‘0’ para os sinais de *write_enable*, é feita uma solicitação de escrita no barramento.

Durante o processo de leitura ou escrita no barramento por parte da interface mestre, o funcionamento do Plasma é paralisado, só sendo este liberado após o término da transferência de dados.

Após a síntese, o IP seguindo esta segunda abordagem, contendo somente a lógica de acoplamento e o Plasma, ocupou 13% da capacidade da FPGA

```
entity opb_plasma is
  generic(
    C_PLASMA_CLK_DIVISOR    : integer              := 2;
    C_MAX_RETRY             : integer              := 5;
    C_BASEADDR              : std_logic_vector     := X"70000000";
    C_HIGHADDR              : std_logic_vector     := X"7000FFFF";
    C_OPB_AWIDTH            : integer              := 32;
    C_OPB_DWIDTH            : integer              := 32;
    C_USER_ID_CODE          : integer              := 3;
    C_FAMILY                : string               := "virtex2p"
  );
  port(
    Plasma_uart_TX      : out std_logic;
    Plasma_uart_RX      : in  std_logic;

    OPB_Clk             : in  std_logic;
    OPB_Rst             : in  std_logic;
    --portas do acoplador escravo
    Sl_DBus             : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sl_errAck           : out std_logic;
    Sl_retry            : out std_logic;
    Sl_toutSup          : out std_logic;
    Sl_xferAck          : out std_logic;
    OPB_ABus            : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE              : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    sOPB_DBus           : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW             : in  std_logic;
    OPB_select          : in  std_logic;
    OPB_seqAddr         : in  std_logic;
    -- portas do acoplador mestre
    M_ABus              : out std_logic_vector(0 to C_OPB_AWIDTH-1);
    M_BE                : out std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    M_busLock           : out std_logic;
    M_DBus              : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    M_request           : out std_logic;
    M_RNW               : out std_logic;
    M_select            : out std_logic;
    M_seqAddr           : out std_logic;
    mOPB_DBus           : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_errAck          : in  std_logic;
    OPB_MGrant          : in  std_logic;
    OPB_retry           : in  std_logic;
    OPB_timeout         : in  std_logic;
    OPB_xferAck         : in  std_logic);
end entity opb_plasma;
```

Figura 10: Interface do IP que acopla o Plasma ao barramento OPB

3.3.2 Validação

A validação da lógica de acoplamento foi feita em duas etapas, sendo a primeira através de simulação por software, e a segunda foi sintetizando o IP e ligando-o a um barramento real.

Para a primeira etapa, foi criado um cenário de testes, no qual testou-se tanto a interface mestre, quanto a interface escrava. Este cenário de testes foi criado ligando os sinais da interface mestre aos sinais da interface escrava. Validando deste modo o funcionamento de ambas as interfaces em uma só simulação. Para que a interface mestre gerasse os pedidos de leitura e escrita, foi escrito um programa, em C, cujo código compilado foi executado no Plasma, de modo que houvesse as leituras e escritas no barramento. Este programa está descrito na Figura 11.

```
int main()
{
    volatile unsigned int i,*ptr;

    ptr = (unsigned int*)0x70000000;

    for(i=0; i<30; i++)
    {
        ptr[i+0x400] = ptr[i];
    }
    return 0;
}
```

Figura 11: Programa para validação por simulação do IP

A segunda etapa de validação do IP necessitou que todo o sistema fosse sintetizado e carregado para a FPGA. Para esta validação, outro programa foi escrito. Este programa deveria escrever dados na BRAM presente no barramento OPB, ler os valores que foram escritos e confirmar a validade dos dados lidos. Caso tenha confirmado, escreve um valor de confirmação nos leds, caso contrário, escreve um valor de desaprovação. Após isto, deve executar um contador e, de tempos em tempos, escrever um valor nos leds. O código-fonte está no Anexo A deste trabalho. Foi escrito também uma rotina, a ser executada pelo PowerPC para escrever um valor no endereço correspondente ao sinal de *reset* do Plasma, esta rotina encontra-se disponível no Anexo B.

Após a compilação dos programas, e sintetização do sistema. O mesmo foi programado na FPGA, que depois de iniciada executou as operações exatamente conforme o previsto.

4 Conclusão

Este trabalho se enquadra na área de sistemas embarcados e envolveu diversos conceitos relacionados às ciências da computação, como circuitos digitais, arquitetura de computadores, sistemas operacionais e síntese de hardware. A proposta deste trabalho era sintetizar, em um único chip, uma configuração composta por dois processadores, sendo um deles o soft-core Plasma, ligado ao barramento OPB, e o outro um PowerPC, ligado ao barramento PLB. A validação do IP, descrita no item 3.2.2 comprova, não só a viabilidade do projeto, mas também o correto funcionamento da implementação realizada, tendo cumprido os objetivos previstos.

Os testes realizados mostraram como o Plasma foi adequadamente acoplado ao barramento como escravo, servindo de co-processador a um processador PowerPC, e também mostram a integração funcional do Plasma como mestre do barramento. Esta configuração permite que esse soft-core seja usado como processador principal de um SoC e que o SoC possa ter, inclusive, várias instâncias do Plasma, implementando um multiprocessador em chip.

Em relação à geração do componente de hardware (IP), existem vantagens e desvantagens em utilizar qualquer uma das abordagens de implementação do IP sugeridas: usar o assistente de criação de periféricos da ferramenta EDK da Xilinx, ou implementar o periférico *"from scratch"*.

A primeira tem como vantagem principal a facilidade e a rapidez para a implementação de periféricos simples, mas peca quando é necessária a criação de IPs mais complexos, por poluir a implementação com muitos sinais e por exigir muito da capacidade da FPGA. A segunda abordagem à construção do IP pareceu afinal mais vantajosa e foi adotada, pois, apesar da dificuldade inicial de implementar todo o protocolo OPB, revela-se mais prática, simples e confiável, além de exigir muito menos da capacidade da FPGA (pouco mais que 25% da exigência da primeira abordagem).

Ainda em relação à geração da interface do IP, havia duas opções de projeto. Uma única interface permitindo conectar o plasma como mestre e escravo, ou interfaces separadas. A opção por construir duas interfaces separadas ao invés de apenas uma interface unificada para as duas funções (mestre e escravo) também se revelou bastante promissora, pois propiciou que o teste de validação pela simulação fosse feito de maneira simples, apenas ligando os sinais de uma interface a outra.

4.1 Trabalhos Futuros

Partindo do ponto em que há uma solução completamente aberta e sendo simples a interligação de periféricos e processadores *soft-core* em SoCs usando-se IPs, abre-se caminho para um número muito grande de possíveis aplicações e trabalhos futuros.

Uma sugestão para trabalhos futuros é a implementação de blocos funcionais que possam ser integrados ao Plasma, como uma Unidade de Ponto Flutuante, uma Unidade de gerenciamento de Memória, ou memórias Cache. Esses blocos poderiam ser adicionados ao sistema conforme sua necessidade ou não, de forma a não implicar aumento desnecessário de área da FPGA.

Outros desenvolvimentos futuros nessa incluem a customização do próprio componente Plasma e de suas unidades funcionais (ULA, banco de registradores, memória interna, etc) conforme a aplicação. Ou seja, poderia ser possível remover algumas dessas unidades funcionais internas do processador quando não forem necessárias (quando não houver instruções no código executável que as utilize).

Diversas aplicações ou avaliações de aplicações poderiam ainda ser realizadas. No caso da implementação de múltiplos processadores num SoC, ou mesmo de um processador com muitos periféricos, poderia-se avaliar o impacto da perda de desempenho causada por bloqueios no barramentos, e alternativas de interconexão entre esses componentes, como redes-em-chip.

5 Bibliografia

[1] DANGUI, Sandro. *Modelagem e simulação de barramentos com SystemC*. 2006. 121f. Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Campinas.

[2] GIORGINI, André Linhares. *Implementação de um Controlador PID Digital para Robótica baseado em Computação Reconfigurável*. 2001. 108 f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Universidade de São Paulo.

[3] IBM (International Business Machines Corporation). *On-Chip Peripheral Bus: Architecture Specifications*. Versão 2.1. Carolina do Norte, 2001.

[4] IBM (International Business Machines Corporation). *CoreConnect Bus Architecture*. Carolina do Norte, 1999.

[5] NAVABI, Zainalabedin. *VHDL : Analysis and modeling of digital systems*. 2° Edição. McGraw Hill: Nova Iorque, 1998.

[6] OLIVEIRA, Tadeu F. Desenvolvimento de aplicações para sistemas embutidos: um estudo da plataforma J2ME. 2006. 71 f. Dissertação (Bacharelado em Ciências da Computação) - Universidade Tiradentes.

[7] PATTERSON, David; HENNESSY, John. *Computer Organization and Design: The Hardware/Software Interface*. 2° edição. Califórnia: Morgan Kaufmann, 1997.

[8] POLPETA, Fauze V. *Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação*. 2006. 115 f. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina.

[9] PRICE, Charles. *MIPS IV Instruction Set*. Setembro, 1995.

[10] RHOADS, Steve. *Plasma CPU Core*. Disponível em: <http://www.open-cores.org>. Acesso em: 30 jan. 2007.

[11] WISHBONE, SoC Arquitecture Specification. 2002.

[12] WOLF, Wayne. *Computers as Components*. Nova Iorque: McGraw-Hill, 2001.

[13] XILINX, *PowerPC Processor Reference Guide*. Versão 1.1. Califórnia, 2003.

[14] YAGHMOUR, Karim. *Building Embedded Linux Systems*. Califórnia: O'Reilly, 2003.

ANEXOS

Anexo A:

Aplicação a ser executada no IP, para validação juntamente com um barramento real

Arquivo: vlidacao.c

```
#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)


#define LED_BASEADDR      0x40000000
#define LED_MSB           0x00000080

#define RAM_BASEADDR      0x60000000

unsigned int led_data = 0;

typedef unsigned char  uint8;

void led_changer() {

    if (led_data & LED_MSB)
        led_data = led_data << 1;
    else
        led_data = (led_data << 1) + 1;

    MemoryWrite(LED_BASEADDR, led_data);
}


void Test()
{
    volatile int* ptr = (int*)RAM_BASEADDR;
    volatile int local[50];
    int i;

    for( i=0; i<50; i++)
    {
        ptr[i] = i;
    }

    for( i=0; i<50; i++)
    {
        local[i] = ptr[i];
        if( local[i] != i )
        {
            MemoryWrite(LED_BASEADDR, 0xF3);
            return;
        }

    }
    MemoryWrite(LED_BASEADDR, 0xFC);

    for(i=0; i<0x20000000; i++)
        ;

}
```

```
int main()
{
    unsigned int counter;
    Test();

    counter = 0;
    while(1)
    {
        counter++;
        if(counter == 0x1000000 ) //1 segundo aprox.
        {
            counter = 0;
            led_changer();
        }
    }

    return 0;
}
```

Anexo B:

Aplicação a ser executada no PowerPC para a validação do IP juntamente com um barramento real.

Arquivo: TestInterrupt.c

```
// Located in: ppc405_0/include/xparameters.h
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "xgpio_l.h" /* general-purpose I/O peripheral control functions */
#include "xtmrctr_l.h" /* timer/counter peripheral control functions */
#include "xuartlite_l.h" /* uartlite peripheral control functions */
#include "xintc_l.h" /* interrupt controller control functions */
#include "xexception_l.h" /* PPC exception handler control functions */

#define LED_MSB 0x00000080 /* mask for position of left-most LED */

/* Global variables */
volatile unsigned int exit_command = 0; /* flag from UART ISR to exit */
volatile unsigned int dump_command = 0;

/* UART interrupt service routine */
void uart_int_handler(void *baseaddr_p) {
   char c;

   volatile unsigned int *magic_value;
   magic_value = (unsigned int*)0x70801100;
   while (!XUartLite_mIsReceiveEmpty(XPAR_RS232_UART_BASEADDR)) {
      /* Read a character */
      c = XUartLite_RecvByte(XPAR_RS232_UART_BASEADDR);
      switch (c) {
         case 'x': /* EXIT command */
            exit_command = 1;
            break;
         case 'r': /* reset */
            *((volatile unsigned int*)(0x70803000)) = 0xFF;
            break;
         case 'd': /* dump memory */
            dump_command = 1;
            break;
      }
   }
}

/* Timer interrupt service routine */
/* Note: This ISR is registered statically in the Software Platform Set-
tings dialog */
void timer_int_handler(void * baseaddr_p) {
   unsigned int csr;
   /* Read timer 0 CSR to see if it requested the interrupt */
   csr = XTmrCtr_mGetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0);
   /* Clear the timer interrupt */
   XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0, csr);
}

/* Interrupt test routine */
void InterruptTest(void) {
    const int addr = 0x70800000;
```

```
      int offset = 0;
      volatile int *ptr;
   print("-- Entering InterruptTest() --\r\n");
   /* Initialize exception handling */
   XExc_Init();
   /* Register external interrupt handler */
   XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
      (XExceptionHandler)XIntc_DeviceInterruptHandler,
      (void *)XPAR_OPB_INTC_0_DEVICE_ID);
   /* Register the UART interrupt handler in the vector table */
   XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
            XPAR_OPB_INTC_0_RS232_UART_INTERRUPT_INTR,
            (XInterruptHandler)uart_int_handler,
            (void *)XPAR_RS232_UART_BASEADDR);
   /* Start the interrupt controller */
   XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);
   /* Set the gpio for LEDs as output */
   XGpio_mSetDataDirection(XPAR_LEDS_8BIT_BASEADDR, 1, 0x00000000);
   /* Set the number of cycles the timer counts before interrupting */
   XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 0, timer_count);
   /* Reset the timers, and clear interrupts */
   XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
      XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );
   /* Enable timer and uart interrupt in the interrupt controller */
   XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR,
      XPAR_OPB_TIMER_1_INTERRUPT_MASK | XPAR_RS232_UART_INTERRUPT_MASK);
   /* Start the timers */
   XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
      XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
      XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);
   /* Enable PPC non-critical interrupts */
   XExc_mEnableExceptions(XEXC_NON_CRITICAL);
   /* Enable UART interrupts */
   XUartLite_mEnableIntr(XPAR_RS232_UART_BASEADDR);
    print("Test Periferal Start!!!\r\n");
    while(!dump_command)
        ;
    ptr = (int*)addr;
    for(offset=0; offset < 0x7ff; offset++ )
    {
        volatile unsigned int addr = (unsigned int)ptr;
        volatile unsigned int value = (unsigned int)*ptr;
        xil_printf("%08x = %08x\r\n", addr, value );
        ptr ++;
    }

    print("Test Periferal end!\r\n");

   /* Wait for interrupts to occur until exit_command is asserted */
   while (!exit_command)
   ;
   /* Disable PPC non-critical interrupts */
   XExc_mDisableExceptions(XEXC_NON_CRITICAL);
   print("-- Exiting InterruptTest() --\r\n");
}

int main (void) {
   print("-- Entering main() --\r\n");
   /*
    * MemoryTest routine will not be run for the memory at
    * 0xffff0000 (plb_bram_if_cntlr_1)
```

```
    * because it is being used to hold a part of this application    */
   /* Testing BRAM Memory (opb_bram_if_cntlr_1)*/
   {
      XStatus status;

      print("Starting MemoryTest for opb_bram_if_cntlr_1:\r\n");
      print("  Running 32-bit test...");
      status = XUtil_MemoryTest32( (Xuint32*)
               XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR ,
               512, 0xAAAA5555, XUT_ALLMEMTESTS);
      if (status == XST_SUCCESS) {
         print("PASSED!\r\n");
      }
      else {
         print("FAILED!\r\n");
      }
      print("  Running 16-bit test...");
      status = XUtil_MemoryTest16((Xuint16*)
               XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR,
               1024, 0xAA55, XUT_ALLMEMTESTS);
      if (status == XST_SUCCESS) {
         print("PASSED!\r\n");
      }
      else {
         print("FAILED!\r\n");
      }
      print("  Running 8-bit test...");
      status = XUtil_MemoryTest8((Xuint8*)
               XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR,
               2048, 0xA5, XUT_ALLMEMTESTS);
      if (status == XST_SUCCESS) {
         print("PASSED!\r\n");
      }
      else {
         print("FAILED!\r\n");
      }
   }

   /* Run user-supplied interrupt test routine */
   InterruptTest();
   print("-- Exiting main() --\r\n");
   return 0;
}
```