Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Sistemas Eletrônicos - PSI

PSI-2553- Projeto de Sistemas Integrados

Experiência 2: O Processador Plasma (Teoria)

M.S. / W.J.C (17)

Conteúdo:

1. Objetivos

Esta experiência visa a familiarização do estudante com a arquitetura do processador Plasma, que segue a arquitetura MIPS, introduzida nas aulas de teoria. Serão realizados o estudo e a simulação de uma implementação específica do processador que adota a técnica de segmentação (*pipelining*), além da simulação com um programa que realiza o cálculo de máximo divisor comum. Após a experiência, o aluno saberá correlacionar o código em *assembly*, os formatos de instruções, e o fluxo de dados e controle dentro do processador.

O processador adotado nesta experiência é o Plasma, projetado e disponibilizado no Opencores (www.opencores.com) por Steve Rhoads, que tem também realizado a manutenção constante dos códigos e ferramentas de apoio associados. O processador é um RISC de 32 bits que segue de perto a arquitetura MIPS 32 (www.mips.com). Iremos, nesta seção, relembrar alguns aspectos da arquitetura MIPS, relevantes para esta experiência, além de mostrar algumas diferenças arquiteturais entre a versão da implementação prática e do exposto no livro de Robert Britton.

2. O Processador Plasma- Assembly e Registradores MIPS

As instruções do MIPS a serem usadas nos programas desta experiência estão listadas no tabela de referência do arquivo PDF que acompanha este na página do "Moodle". Elas podem apresentrar uma pequena variação de sintaxe em relação ao visto nas aulas de teoria, porém, são plenamente identificáveis. Se tiver dúvidas ao significado das instruções, o(a) aluno(a) deverá se referir ao livro-texto de Robert Britton.

As instruções devem ser utilizadas com registradores ou constantes como parâmetros. Os 32 registradores devem ser descritos por seus nomes ou, equivalentemente, por seus números, correspondência que é indicada na Tabela 1. Por exemplo, as instruções "addi $4,$15, 0x1cf1" e "addi $a0,$t7, 0x1cf1" são equivalentes (obs. 0x1cf1 indica o número 1cf1 em hexadecimal).

A nomenclatura existe para organizar o código *assembly*, principalmente, pelo seu uso em compiladores (conversão de linguagem de programação em alto nível para *assembly*). Desta forma, com os registradores reservados para tarefas específicas, ficam claros, no código *assembly*, os objetivos do código apresentado. A seguir, uma breve explicação sobre alguns registradores é dada; outros, para os quais o uso é intuitivo e já foram discutidos em aulas de teoria, não serão aqui tratados.

- $at é um registrador usado em caso de pseudo-instruções. O conjunto "oficial" de instruções *assembly* da arquitetura MIPS é razoavelmente simplificado, portanto ela é estendida para novas pseudo-instruções, que no processo de montagem são convertidos em duas ou mais instruções. Nestas conversões, o registrador $at é utilizado para armazenar valores temporários.
- $s e $t são variáveis temporárias que são utilizadas igualmente em um programa principal. Quando há chamada de sub-rotinas, os $s passam a ter função especial, pois armazenam valores de estado do chamador, ficando estes "protegidos", para que a sub-rotina não os modifique.
- $k são usados para armazenar dados/endereços de interrupção, seja de hardware ou software.
- $gp, $sp e $fp. Para entender a função destes registradores, é importante saber que o espaço de memória (total de 4GB) é dividido em quatro partes: a) reservado para S.O.; b) instruções de programa; c) dados globais (de uso geral pelo programa); d) dados locais (validade em sub-rotinas). O $gp guarda o endereço de início dos dados globais como referência; $sp guarda o endereço do topo da pilha onde contém os dados locais; finalmente, $fp guarda o endereço da pilha específica de sub-rotinas ativadas.

Tabela 1. Conjunto de registradores MIPS

Nome	Número	Uso
$0	$0	Valor constante 0
$at	$1	Temporário de montagem
$v0-$v1	$2-$3	Valores de retorno de sub-rotinas
$a0-$a3	$4-$7	Argumentos de sub-rotinas
$t0-$t7	$8-$15	Variáveis temporárias
$s0-$s7	$16-$23	Variáveis gravadas
$t8-$t9	$24-$25	Variáveis temporárias
$k0-$k1	$26-$27	Temporários do S.O.
$gp	$28	Ponteiro global
$sp	$29	Ponteiro de pilha (stack)
$fp	$30	Ponteiro de quadro
$ra	$31	Endereço de retorno de sub-rotina

3. Estrutura de Arquivos para Análise e Implementação da Aplicação

Como já mencionado no tutorial do Plasma, há uma série de arquivos que são gerados e utilizados na implementação de um sistema embutido (baseado na arquitetura MIPS). O usuário deve fornecer como entrada o comportamento desejado de sua aplicação em código C ou Assembly. Na nossa experiência, haverá três arquivos, como mostrado na Figura 1:

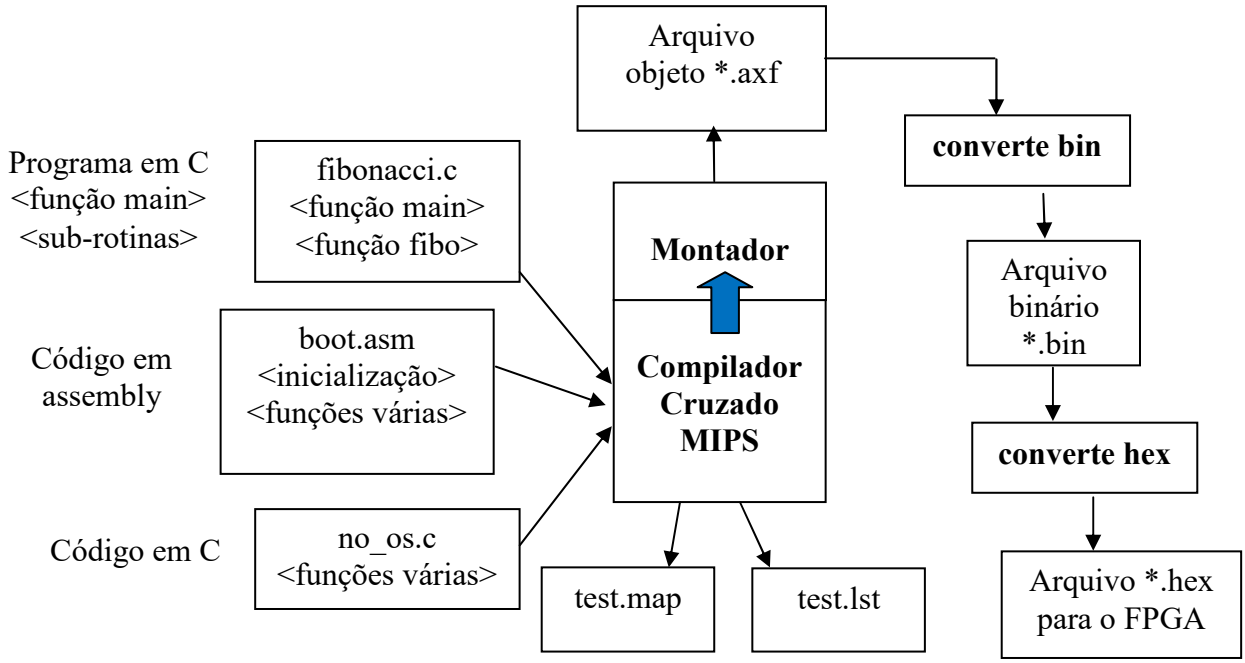

Figura 1. Estrutura de arquivos no Plasma

- *fibonacci.c*: contendo o algoritmo Fibonacci, na linguagem C, dado por uma função "main" que chama uma sub-rotina "fibo".
- *boot.asm*: arquivo em assembly, contendo um código para inicialização (limpeza da memória e chamada do main), além de várias rotinas codificadas em assembly. Observe-se que estas rotinas podem ser chamadas por programas codificadas em C, sendo que o linker se encarrega de juntá-las.
- *no_os.c*: como esta versão do Plasma não há suporte de sistema operacional, este arquivo fornece algumas funções de interface com o meio externo (periféricos).

O compilador cruzado gera alguns arquivos sendo o *test.map* e *test.lst* os mais importantes para a nossa exeperiêcia. O primeiro contém um sumário dos endereçamentos na memória das diversas sub-rotinas sendo que grande utilidade para a compreensão do código completo em assembly MIPS que está em *teste.lst*.

O código assembly é montado em um código de máquina compatível com o Plasma e, em princípio, o arquivo binário é carregado na memória. Este código é convertido em um formato legível *.hex, o qual é realmente é introduzido no FPGA, através do código VHDL. O formato *.hex é apresentado no Apêndice da parte prática da apostila.

4. Implementação do Processador Plasma

4.1. Hierarquia de Módulos do Plasma

O módulo topo, denominado de **plasma** (plasma.vhd), é na realidade, o sistema embutido todo composto de processador, memória e periféricos. Vamos nos deter inicialmente em dois submódulos do topo, como mostrado na Figura 2, para observar os protocolos de comunicação na transferência de dados: **mlite_cpu**, que corresponde ao processador de fato, e **ram** (ram.vhd), que implementa a memória principal. A arquitetura seguida é a Von Neumann, isto é, a mesma memória física ram é usada para dados e para instruções. A memória ram trabalha com palavras de largura de 1 byte. Apesar de receber 30 bits de endereçamento (bits 31 a 2, os mais significativos de uma palavra de 32 bits, já que os dois menos significativos (0-ésimo e 1-ésimo) correspondem aos quatro bytes que compõem um dado de 32 bits), internamente, a RAM é de 8 kbytes, considerando um espaço útil de memória de 2^{13} posições, ou seja 13 bits de endereçamento.

Alguns dos portos da Figura 2 são auto-explicativos, mas muitos merecem uma melhor exposição. São eles:

- *intr_in*: sinal de interrupção;
- *enable*: é um sinal de seleção do módulo ram (determinado pelo espaço de memória do sistema), obtido pela decodificação dos 3 bits mais significativos do endereço *address_next*; quando estes 3 bits forem "000", sabe-se que a destinação é a ram e *enable* é '1'
- *address* (da **ram**): endereço de escrita ou leitura de dados da ram, operação efetivada no ciclo seguinte;
- *write_byte_enable* (da **ram**): sinal de 4 bits de *enable* de escrita para cada byte da palavra separadamente (despendendo se cada um dos bits correspondentes é `1`) e de leitura de dados da RAM (quando todos os bits são `0`), operação efetivada no ciclo seguinte; permite escrita em bytes, half-words, etc.

- *address_next* e *byte_we_next* (do **mlite_cpu**): idem aos dois grupos acima, ou seja, endereço e *write enable* antecipado, ou seja, do dado que será escrito na ou disponibilizado pela Ram, no ciclo seguinte, seja para instrução seja para dados;
- *address* e *byte_we* (do **mlite_cpu**): endereço e *write enable* do ciclo atual para periféricos;
- *mem_pause*: sinal indicativo de situações em que o pipeline ou operações devem ser suspensos por um ou mais ciclos. É o caso de tentativa de escrita em periféricos ocupados ou operações de load ou store que atrasam a operação de *fetch* (problema que não ocorreria na arquitetura Harvard).

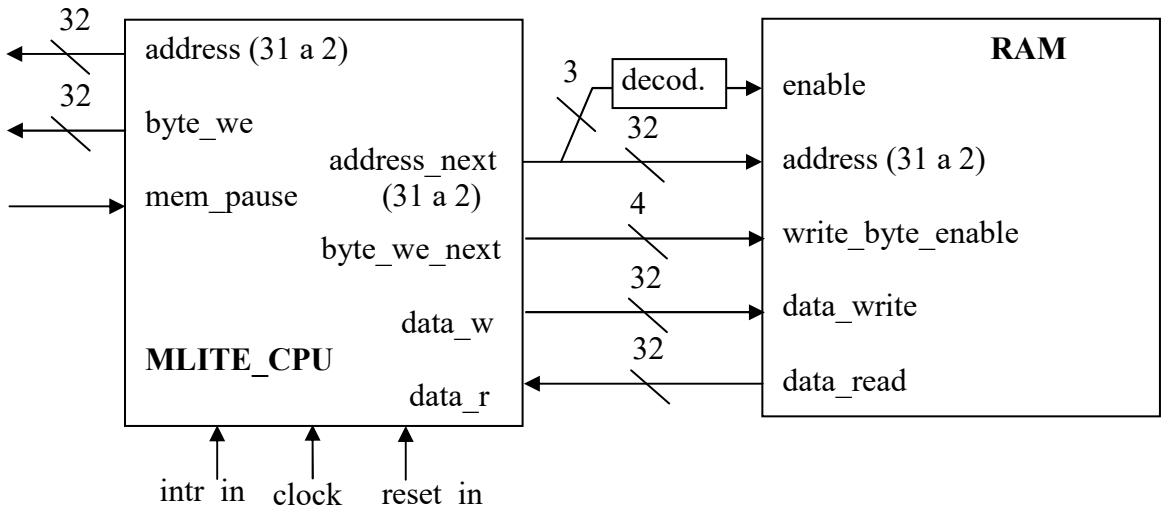

Figura 2. Módulo topo – plasma

A CPU, denominada **mlite_cpu** (mlite_CPU.vhd) é composta dos submódulos apresentados na Figura 3. A grande quantidade de sinais dificulta o entendimento de seu funcionamento, mas em termos gerais os blocos têm as seguintes funcionalidades:

- **pc_next**: responsável pelo cálculo dos próximo valor de **pc** (derivado de jump, branch ou seqüencial);
- **mem_ctrl**: controle de memória, onde os sinais para acesso à memória ram são gerados; os dados lidos e escritos passam também pelo módulo;
- **control**: analisa a instrução, classifica os opcodes e separa os diversos campos.
- **mult**, **alu** e **shifter**: execução de operações aritméticas e lógicas;
- **bus_mux**: caixa de multiplexação, para definir a destinação (à memória ou registrador);
- **reg_bank**: banco de registradores (32 de uso geral, além de específicos para interrupção).

O **mlite_cpu** permite duas implementações de *pipeline*, ou em 2 ou 3 estágios. As duas formas de pipeline são evidenciadas na Figura 4: à esquerda, temos os dois estágios de *pipeline*, enquanto à direita, a forma em três estágios é ilustrada. É importante observar novamente a restrição derivada da memória comum para dado e instrução. As escritas ou leituras de dados em memória, neste caso, levam um ciclo a mais (envolvendo ação com **mem_ctrl**, não mostrada explicitamente nas figuras) e causam a paralisação da tarefa de *fetching*.

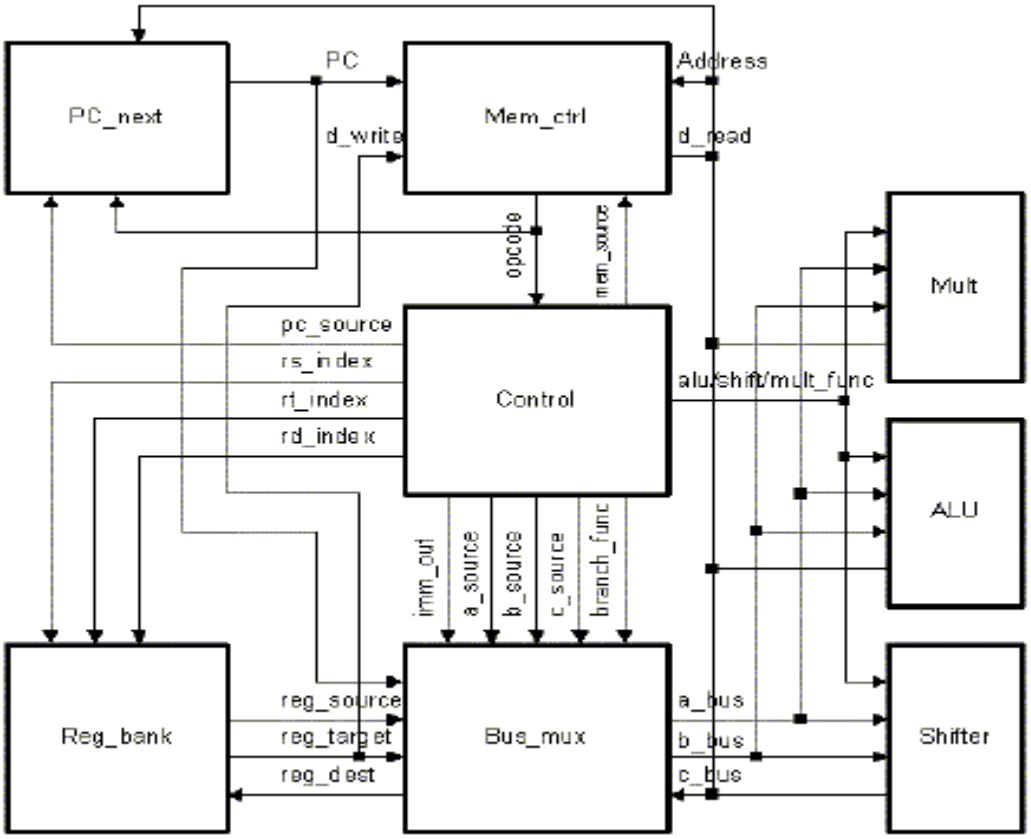

Figura 3. Módulo mlite_cpu

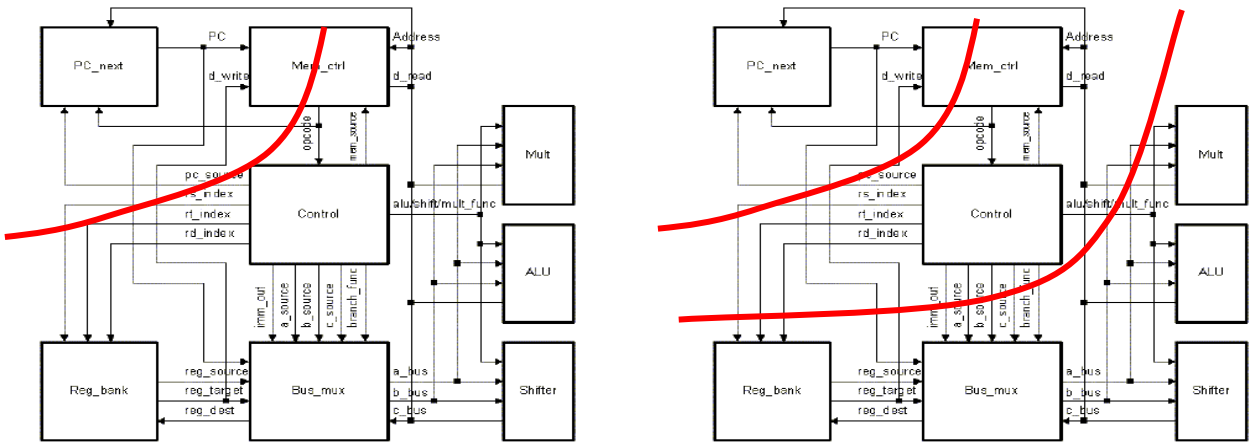

Figura 4. Opções de pipeline: 2 estágios (esquerda) e 3 estágios (direita)

4.2. Memória RAM em FPGAs Altera

O código VHDL do Plasma permite a implementação da memória RAM de diversas formas, através de parâmetros de configuração, entre elas, mapeamento para blocos pré-projetados de memórias para FPGAs da Xilinx ou Altera, ou blocos dedicados, cujo código completo faz parte do arquivo ram.vhd. No caso de dispositivos Altera que utilizaremos na experiência, existe a particularidade que os bancos de memória são de 1 byte e, por esta razão, quatro bancos idênticos são utilizados. Como os quatro bancos juntos compõem as palavras de 32 bits, o esquema utilizado é o da Figura 5. Pode-se notar que o mesmo endereço (com os dois bits menos significativos

excluídos) é utilizado para o acesso em todos os quatro bancos, e os bytes individuais são concatenados.

É importante notar que este esquema vale tanto para dados como para as instruções que ali residirão.

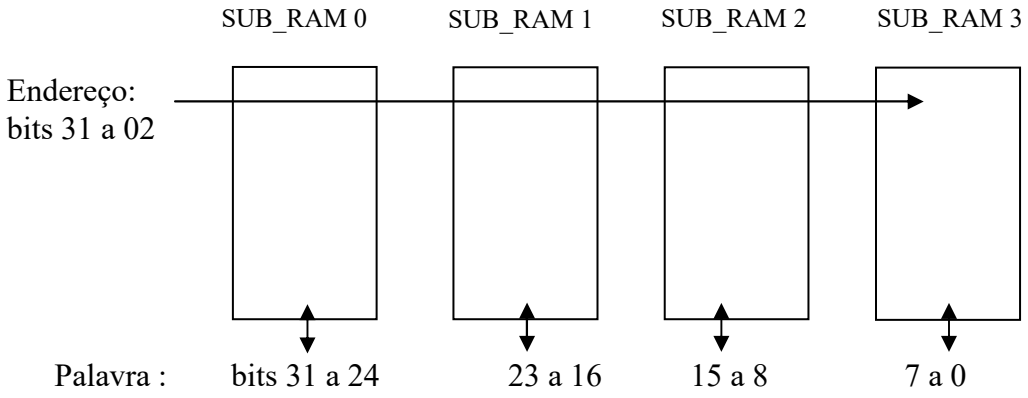

Figura 5. Acesso a uma palavra nas memórias RAM da Altera