

PROCESSADOR MULTI-CICLO – MR4

1 CARACTERÍSTICAS GERAIS DAS ARQUITETURAS MRX

- As arquiteturas MRx são do tipo *load-store*, ou seja, as operações lógicas e aritméticas são executadas exclusivamente entre registradores da arquitetura ou entre constantes imediatas e registradores. As operações de acesso à memória só executam ou uma leitura (*load*) ou uma escrita (*store*).
- Devido à característica *load-store*, o processador deve disponibilizar um conjunto relativamente grande de registradores, para reduzir o número de acessos à memória externa, pois estes representam perda de desempenho em relação a operações entre registradores internos ao processador. Esta característica difere da arquitetura baseada em acumulador, a qual mantém todos os dados em memória, realizando as operações aritméticas entre um conteúdo que está em memória e um, ou poucos registradores especiais, denominados acumuladores. Considere o exemplo do código C a seguir: `for(i=0; i<1000; i++)`. Neste exemplo, caso *i* esteja armazenado em memória, tem-se 2000 acessos à memória, realizando leitura e escrita a cada iteração do laço `for`. Caso se tenha o valor de *i* armazenado em um registrador interno, apenas opera-se sobre este, sem acesso à memória externa durante a maior parte do tempo. Considerando-se que o tempo de acesso a um registrador é normalmente de uma a duas ordens de grandeza (10 a 100 vezes) menor que o tempo de acesso a uma posição de memória, percebe-se o ganho de desempenho que se pode auferir em relação a arquiteturas baseadas em acumulador.
- Dados e endereços nas arquiteturas MRx são de 32 bits. Logo, diz-se que a **palavra** destes processadores é de 32 bits, ou que se trata de processadores de 32 bits. Compara com a Cleópatra, um processador de 8 bits.
- O endereçamento de memória é orientado a byte, ou seja, cada endereço corresponde a um identificador de posição onde residem apenas 8 bits. Uma palavra do processador, ao ser armazenada por este ocupa 4 posições consecutivas de memória.
- O banco de registradores possui 32 registradores de uso geral, de 32 bits cada um, denominados \$0 a \$31. O registrador \$0 não é realmente um registrador, mas a constante 0 representada em 32 bits, disponível para uso em instruções que necessitem usar este valor.
- Há um formato regular para as instruções, todas possuem exatamente o mesmo tamanho, e ocupam 1 palavra em memória, ou seja, 4 posições consecutivas de memória. Deve-se comparar esta situação com a que ocorre na arquitetura Cleópatra. A instrução contém o código da operação e o(s) operando(s), caso exista(m). Acesso a instruções são sempre alinhados em uma fronteira de palavra inteira. Assim, todo endereço válido de instrução possui os dois últimos bits iguais a 0.

Esta família de processadores é derivada da famosa arquitetura MIPS, sendo de fato um subconjunto da mesma. Trata-se praticamente uma máquina RISC faltando, contudo algumas características que existem em qualquer RISC, tal como *pipelines*, assunto que será introduzido ao final desta disciplina e estudado em profundidade maior em Arquitetura de Computadores I.

2 CARACTERÍSTICAS ESPECÍFICAS DO PROCESSADOR MR4

- Não existem biestáveis qualificadores de estado na MR4. A detecção de qualificadores, tais como vai-um, transbordo, geração de números negativos e geração do resultado zero devem ser realizadas em software ou usando instruções específicas de comparação existentes na arquitetura;
- Modos de endereçamento que implicam múltiplos acessos à memória (tais como o modo indireto) não existem. O modo direto ou absoluto é limitado a casos especiais, em que seu uso é inevitável e produz código relocável. Os modos a registrador, relativo e suas combinações são privilegiados na MR4.
- O processador MR4 não possui suporte em hardware para a implementação de pilhas, mas duas instruções (`JAL/JALR`) têm funcionalidade que, associada ao banco de registradores permite simular esta funcionalidade.

3 A RELAÇÃO PROCESSADOR - MEMÓRIAS – AMBIENTE EXTERNO

A Figura 1(a) mostra como deve ser implementada a organização do processador MR4 neste trabalho. Nela, está explicitada a relação entre processador, memórias externas e mundo exterior ao subsistema processador

– memórias de dados e instruções. O mundo externo é o responsável por gerar os sinais de **clock** e **reset**. Uma característica primordial desta organização é que ela deve ser implementada como uma **organização Harvard**, ou seja, o processador deve usar interfaces distintas para as memórias de instruções e de dados. Alguns trabalhos anteriores previam o uso de uma interface de memória unificada para instruções e dados, caracterizando uma organização von Neumann. Segue agora uma breve discussão da organização MR4 a ser implementada. Esta será detalhada em Seções posteriores deste documento.

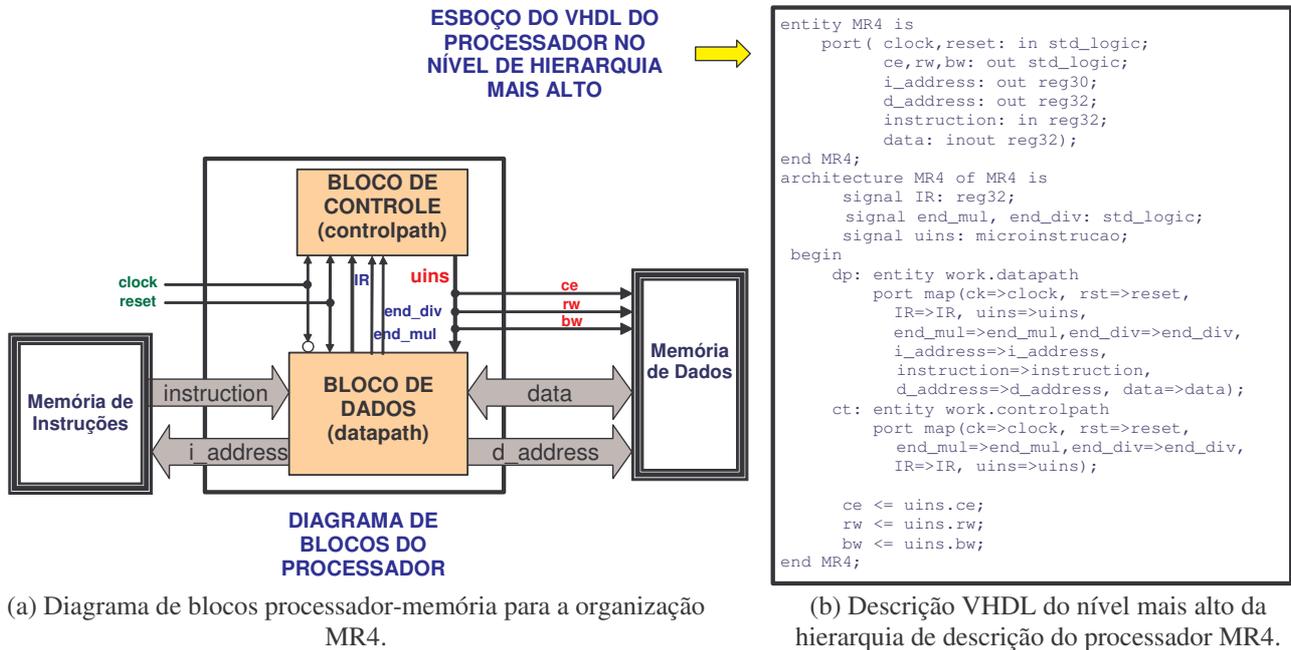


Figura 1 - Relação entre o processador, ambiente externo, memórias externas e esboço de descrição VHDL de topo para a organização MR4 a ser implementada.

O processador MR4 recebe do mundo externo dois sinais de controle. O **clock** sincroniza todos os eventos internos do processador. O sinal **reset** leva o processador a reiniciar a execução de instruções a partir do endereço 00400000H da memória (este endereço é definido por compatibilidade com o endereço assumido pelo simulador SPIM, a ser usado no trabalho). O sinal **reset** pode ser usado para provocar o reinício da execução de instruções.

Os sinais providos pelo processador MR4 para a troca de informações com as memórias são:

- ◆ **i_address** – um barramento unidirecional de 30 bits que define sempre o endereço da posição de memória contendo a instrução a ser buscada a seguir. O motivo de este barramento ser de 30 e não 32 bits (a dimensão total de um endereço de memória) é explicado na Seção 3.1 abaixo, quando se discutir a estrutura de acesso à memória;
- ◆ **instruction** – um barramento unidirecional de 32 bits, apresentando a instrução contida na posição de memória dada por **i_address**;
- ◆ **d_address** – um barramento unidirecional de 32 bits, contendo o endereço da posição de memória a ser acessada para leitura ou escrita de dados, da ou para a memória de dados, respectivamente;
- ◆ **data** - um barramento bidirecional de 32 bits transportando dados do ou para o processador MR4;

O bloco de controle gera a palavra de microinstrução (**uins**) para comandar a execução passo a passo das instruções pelo bloco de dados. A microinstrução é responsável por especificar cada uma das ações unitárias que serão executadas pelo hardware do bloco de dados a cada ciclo de relógio, ou seja, as microoperações. Exemplos destas são cada uma das três seleções de registradores a serem escrito/lidos no/do banco de registradores, a operação que a ALU (do inglês, *Arithmetic Logic Unit*) executará, e os sinais de controle de acesso à memória de dados externa.

O bloco de dados envia para o bloco de controle a instrução corrente (conteúdo do registrador **IR**). O bloco de dados também é responsável pela troca de informações com a memória externa. Os sinais de controle **end_mul** e **end_div** são produzidos no Bloco de Dados pelas operações de multiplicação e divisão, informando ao bloco de controle o final de sua execução. Isto é necessário porque estas operações mais complexas gastam múltiplos ciclos de relógio para executar. O bloco de controle comanda o início da execução destas e aguarda pela informação de seu término para seguir com o fluxo normal de ciclos de controle.

É importante ressaltar que os blocos de dados e de controle operam sempre em fases distintas do sinal **clock**. Na borda de subida de **clock**, o bloco de controle gera a microinstrução, e na borda de descida o bloco de dados executa esta, modificando seus registradores internos. Com isto, sempre se tem dado estável nas transições de relógio em cada um dos blocos¹. A Figura 1(b) representa o nível mais alto da hierarquia do processador, em linguagem VHDL. Nesta Figura, deve-se notar que os blocos de dados e de controle são instanciados e conectados entre si por *sinais*, conforme definido pelos comandos *port map* no processo de instanciação.

3.1 Estrutura de Acesso à Memória

A partir da discussão da Figura 1, percebe-se que ambas as interfaces de memória são assíncronas, ou seja, não dependem de sinais de relógio tais como o sinal **clock**. Cada uma das interfaces de acesso define um **mapa de memória**, conforme ilustrado na Figura 2. Um mapa de memória é uma abstração que permite organizar as informações acessadas pelo processador segundo uma tabela. Cada posição deste mapa possui um endereço associado e armazena um valor. Na arquitetura MR4, ambos os mapas (de instruções e de dados) possuem endereços de 32 bits e conteúdos associados de 8 bits. Assim, diz-se que as memórias de instruções e de dados são **endereçadas a byte** (8 bits) e cada um destes mapas possui 2^{32} posições (4.294.967.296 posições, correspondendo ao número de endereços diferentes de 32 bits, de 00000000H a FFFFFFFFH).

A memória de instruções armazena apenas instruções. Cada instrução na MR4 ocupa exatamente 32 bits. Assim, uma instrução ocupa 4 posições de memória. Como visto na Figura 1, contudo, o barramento de instruções é de 32bits. Logo, uma leitura da memória de instruções busca desta não apenas o byte cujo endereço está no barramento **i_address**, mas este e os três seguintes (esta é a regra ditada pela arquitetura). Uma limitação imposta pela arquitetura para facilitar o acesso à memória de instruções é **alinhar instruções em uma fronteira de palavra**. Esta expressão significa que cada instrução da MR4 só pode começar a partir de um endereço múltiplo de 4, ou seja, aqueles cujos 2 últimos bits são 00. Assim, por exemplo, uma instrução pode ocupar os endereços de **AAA00004** a **AAA00007** na memória de instruções, mas nunca os endereços **FF004677** a **FF00467A** da mesma memória, embora ambas as faixas representem 4 bytes consecutivos na memória. Dada esta situação, o barramento **i_address** não precisa ter 32 bits, apenas 30, pois os 2 últimos são sempre 00.

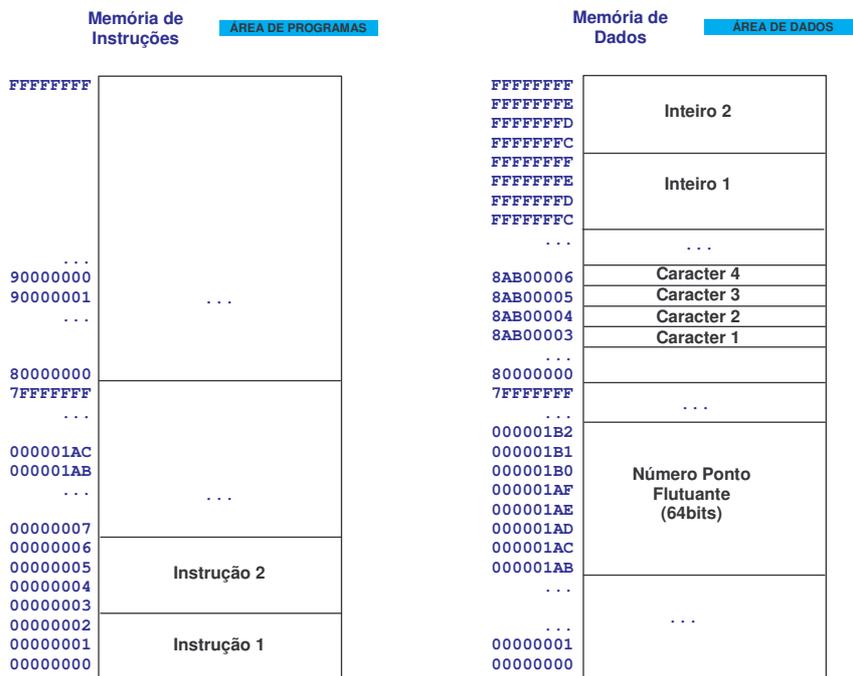


Figura 2 - Ilustração da estrutura dos mapas de memória da MR4. Cada mapa possui 4Gposições de 1 byte. Cada instrução da memória de instruções ocupa exatamente 4 bytes consecutivos, e sempre começa num endereço onde os 2 últimos bits são 00 (alinhamento a palavra).

Não existem sinais de controle para acesso à memória de instruções. Isto não é necessário, pois não há fluxo bidirecional de informação. A memória de instruções é vista pelo MR4 como uma memória de apenas

¹ Deve estar claro que esta última afirmação é verdade apenas se a frequência de relógio for suficientemente baixa para permitir que os sinais estabilizem em no máximo meio período de relógio (entre duas bordas opostas consecutivas do sinal **clock**).

leitura, que fornece informações na sua saída (instruções) a partir do estabelecimento do endereço de memória pelo processador no barramento **i_address**.

A memória de dados é um pouco mais complicada, devido a duas necessidades. Primeiro, os dados podem ter tamanho variável, segundo, deve-se poder não apenas ler, mas também escrever nesta memória. A primeira necessidade faz com que os acessos às informações tenham de ser mais flexíveis. Por exemplo, um inteiro típico ocupa 32 bits, mas um caracter ocupa exatamente 1 byte, e um número em ponto flutuante pode ser de 32 ou de 64 bits. Além disto, estruturas complexas, tais como usadas em programação de alto nível (e.g. **struct** de C/Java ou **record** de Pascal), podem ter qualquer tamanho. Vetores e arranjos multidimensionais de números caracteres e de estruturas complexas complicam ainda mais a situação. Logo, a interface de acesso a dados da memória de dados deve ser flexível para prover desempenho e economia de memória. Ainda assim, o barramento de dados (**data**) é fixo, de 32 bits. Logo, se lê sempre 4 bytes (ou posições) de memória de cada vez. O que é lido pode ser 1 inteiro, 4 caracteres, ou metade de um número em ponto flutuante. Como a leitura é alinhada em qualquer fronteira, é até mesmo possível ler metade de um inteiro e dois caracteres.

O controle de acesso à memória de dados é feito pelo processador através dos sinais **ce**, **rw** e **bw**. O sinal **ce** indica se está em curso uma operação com a memória de dados (quando **ce** = 1) e o sinal **rw** indica se esta operação é de escrita (quando **rw** = 0) ou de leitura (quando **rw** = 1). Obviamente, quando **ce** = 0 o valor do sinal **rw** é irrelevante, a exemplo do que ocorria na Cleópatra. O sinal **bw** serve para indicar se uma operação de escrita na memória deve escrever apenas um byte na posição da memória de dados cujo endereço aparece no barramento **d_address** (**bw** = 0) ou se esta deve escrever nesta posição e nas três seguintes (escrita de palavra, **bw** = 1). Nas operações de leitura este sinal é usado no bloco de dados para controlar a escrita de um byte ou de uma palavra proveniente da memória de dados em um registrador interno (nas instruções **LW** e **LBU**).

3.2 Uma palavra sobre “Endianismo”

Processadores podem numerar os bytes dentro de uma palavra fazendo o byte com menor número sendo ou o byte mais à esquerda ou o byte mais à direita. A convenção usada por uma máquina é a ordem escolhida. Processadores que usam a arquitetura MIPS podem operar com qualquer das ordens. A ordem escolhida é denominada o “endianismo” do processador (ou da organização deste processador) e pode ser “*big-endian*” ou “*little-endian*”. Processadores *big-endian* assumem os 8 bits mais à esquerda de uma palavra (para números a parte mais significativa deste) como sendo o byte 0, os 8 bits imediatamente à direita deste como o byte 1, etc. Processadores *little-endian* assumem os 8 bits mais à direita de uma palavra (para números a parte menos significativa deste) como sendo o byte 0, os 8 bits imediatamente à esquerda deste como o byte 1, etc. O simulador a ser usado (SPIM) opera com ambas as ordens. A ordem específica do SPIM é determinada pela ordem dos bytes da máquina que executa o simulador. Assim, ao rodar o simulador em um PC, SPIM é *little-endian*, enquanto que ao rodar SPIM em uma Sun, SPIM é *big-endian*.

4 CONJUNTO DE INSTRUÇÕES

A Tabela 1 descreve resumidamente cada instrução da arquitetura MR4, usando as convenções a seguir.

Convenções Utilizadas na Tabela 1:

- ◆ **Rd (destination register)** é o registrador usado na maioria das instruções como destino dos dados processados, especificado por um código binário de 5 bits. Ver Seção 5;
- ◆ **Rs (source register)** e **Rt (target register)** são registradores usados em muitas instruções como origem dos operandos para obter os dados. São especificados por códigos binários de 5 bits. Ver Seção 5;
- ◆ Registradores específicos da máquina são indicados por **\$xx**, onde **xx** é o número decimal do registrador. Ver Seção 5;
- ◆ Os registradores **Hi** e **Lo** são registradores especiais (que se encontram fora do banco de registradores de uso geral) usados para armazenar os resultados das operações de multiplicação e divisão (**MULTU** e **DIVU**);
- ◆ O sinal **←** é usado para designar atribuição (escrita) de valores resultantes da avaliação da expressão à direita do sinal ao registrador ou posição de memória identificada à esquerda do sinal;
- ◆ Os identificadores **imed16** e **imed26** representam operandos imediatos de 16 e 26 bits, respectivamente;
- ◆ O identificador **shamt** representa a quantidade de bits a deslocar nas instruções **SLL**, **SRA** e **SRL**. O operador **mod** representa o resto da divisão inteira;
- ◆ O operador **&** representa a concatenação de vetores de bits;
- ◆ A expressão **PMEMD(X)** representa o conteúdo de uma posição de memória de dados cujo endereço é **X** (na leitura) ou a própria posição da memória de dados (na escrita). A quantidade de bits relevantes depende da instrução específica executada. Por exemplo, em **LW** são 32 bits, em **LBU** são 8 bits;

- ◆ Está implícito em todas as instruções o incremento do registrador PC após a busca da instrução. Na organização, isto se reflete no uso de um registrador temporário denominado NPC, que contém o valor do PC incrementado após a busca de uma instrução. Qualquer outra referência à manipulação do PC é parte da semântica da instrução particular;
- ◆ **Extensão de sinal** é a operação que transforma um dado vetor de bits em outro maior, mas cujo valor em complemento de 2 é equivalente. Consiste em copiar o bit de sinal (ou seja, o bit mais significativo do vetor) localizado à esquerda do vetor original tantas vezes quanto seja necessário, para gerar o vetor maior. Por exemplo, na instrução **ADDIU**, se **imed16** for 1111 1111 1111 1111 (-1 em complemento de 2, 16 bits), a extensão de sinal transforma este vetor em 1111 1111 1111 1111 1111 1111 1111 1111 (-1 em complemento de 2, 32 bits). A operação é trivialmente correta também para números positivos, onde o bit de sinal estendido é o bit 0. Quando se menciona extensão de sinal, os valores imediatos representam números em complemento de 2. Caso contrário, os números são representações binárias puras, como ocorre na instrução **J**; No simulador SPIM, a extensão de sinal é feita sobre os 14 bits, e não sobre 16 bits (parece ser um *bug* do simulador, e que afeta a implementação do processador aqui especificado);
- ◆ Os operandos imediatos das instruções de salto têm um tratamento diferente dos operandos das instruções que não se referem a saltos. Como o endereçamento de ambas as memórias (instruções e dados) é feito à byte (ou seja, cada byte de memória possui um endereço distinto), uma instrução ocupa quatro endereços de memória. A CPU manipula operandos de salto multiplicando eles por 4 antes de operar (o que equivale a deslocar o operando dois bits à esquerda, inserindo zeros). Isto vale apenas para as instruções **BEQ**, **BGEZ**, **BLEZ**, **BNE**, **J** e **JAL**.

Tabela 1 – Mnemônicos, codificação e semântica resumida das instruções do processador MR4. As constantes numéricas dos códigos de instrução são valores hexadecimais.

Instrução	FORMATO DA INSTRUÇÃO						AÇÃO
	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0	
ADDU Rd, Rs, Rt	00	Rs	Rt	Rd	00	21	Rd ← Rs + Rt
SUBU Rd, Rs, Rt	00	Rs	Rt	Rd	00	23	Rd ← Rs – Rt
AND Rd, Rs, Rt	00	Rs	Rt	Rd	00	24	Rd ← Rs and Rt
OR Rd, Rs, Rt	00	Rs	Rt	Rd	00	25	Rd ← Rs or Rt
XOR Rd, Rs, Rt	00	Rs	Rt	Rd	00	26	Rd ← Rs xor Rt
NOR Rd, Rs, Rt	00	Rs	Rt	Rd	00	27	Rd ← Rs nor Rt
SLL Rd, Rt, shamt	00	00	Rt	Rd	shamt	00	Rd ← Rt deslocado shamt bits à esquerda (0s à direita)
SLLV Rd, Rt, Rs	00	Rs	Rt	Rd	00	04	Rd ← Rt deslocado (Rs mod 32) bits à esquerda (0s à direita)
SRA Rd, Rt, shamt	00	00	Rt	Rd	shamt	03	Rd ← Rt deslocado shamt bits à direita (mantendo sinal)
SRAV Rd, Rt, Rs	00	Rs	Rt	Rd	00	07	Rd ← Rt deslocado (Rs mod 32) bits à direita (mantendo sinal)
SRL Rd, Rt, shamt	00	00	Rt	Rd	shamt	02	Rd ← Rt deslocado shamt bits à direita (0s à esquerda)
SRLV Rd, Rt, Rs	00	Rs	Rt	Rd	00	06	Rd ← Rt deslocado (Rs mod 32) bits à direita (0s à esquerda)
ADDIU Rt, Rs, lmed16	09	Rs	Rt	lmed16			Rt ← Rs + (lmed16 com sinal estendido)
ANDI Rt, Rs, lmed16	0C	Rs	Rt	lmed16			Rt ← Rs and (0x0000 & (lmed16))
ORI Rt, Rs, lmed16	0D	Rs	Rt	lmed16			Rt ← Rs or (0x0000 & (lmed16))
XORI Rt, Rs, lmed16	0E	Rs	Rt	lmed16			Rt ← Rs xor (0x0000 & (lmed16))
LUI Rt, lmed16	0F	0	Rt	lmed16			Rt ← (lmed16 & 0x0000)
LBU Rt, lmed16(Rs)	24	Rs	Rt	lmed16			Rt ← 0x000000 & PMEMD(lmed16 com sinal estendido+Rs)
LW Rt, lmed16(Rs)	23	Rs	Rt	lmed16			Rt ← PMEMD(lmed16 com sinal estendido+Rs) (4 bytes)
SB Rt, lmed16(Rs)	28	Rs	Rt	lmed16			PMEMD(lmed16 com sinal estendido+Rs) ← Rt [7:0] (1 byte)
SW Rt, lmed16(Rs)	2B	Rs	Rt	lmed16			PMEMD(lmed16 com sinal estendido+Rs) ← Rt (4 bytes)
SLT Rd, Rs, Rt	00	Rs	Rt	Rd	00	2A	Rd ← 1 se Rs menor que Rt (c/sinal), senão Rd ← 0
SLTU Rd, Rs, Rt	00	Rs	Rt	Rd	00	2B	Rd ← 1 se Rs menor que Rt (s/sinal), senão Rd ← 0
SLTI Rt, Rs, lmed16	0A	Rs	Rt	lmed16			Rt ← 1 se Rs menor que lmed16 (c/sinal), senão Rt ← 0
SLTIU Rt, Rs, lmed16	0B	Rs	Rt	lmed16			Rt ← 1 se Rs menor que lmed16 (s/sinal), senão Rt ← 0
BEQ Rs, Rt, rótulo	04	Rs	Rt	lmed16			PC ← PC + (lmed16 & “00” com sinal estendido), se Rs=Rt
BGEZ Rs, rótulo	01	Rs	01	lmed16			PC ← PC + (lmed16 & “00” com sinal estendido), se Rs=>0
BLEZ Rs, rótulo	06	Rs	00	lmed16			PC ← PC + (lmed16 & “00” com sinal estendido), se Rs<=0
BNE Rs, Rt, rótulo	05	Rs	Rt	lmed16			PC ← PC + (lmed16 & “00” com sinal estendido), se Rs≠Rt
J rótulo	02	lmed26					PC ← PC[31:28] & lmed26 & “00”
JAL rótulo	03	lmed26					\$31 ← NPC; PC ← PC[31:28] & lmed26 & “00”
JALR Rd, Rs	00	Rs	00	Rd	00	09	Rd ← NPC; PC ← Rs
JR Rs	00	Rs	0000			08	PC ← Rs

MULTU	Rs, Rt	00	Rs	Rt	000	19	Hi & Lo \leftarrow Rs * Rt
DIVU	Rs, Rt	00	Rs	Rt	000	1B	Lo \leftarrow Rs / Rt; Hi \leftarrow Rs mod Rt
MFHI	Rd	00	000	Rd	00	10	Rd \leftarrow Hi
MFLO	Rd	00	000	Rd	00	12	Rd \leftarrow Lo

4.1 Classes Funcionais de Instruções

A partir da Tabela 1, propõe-se a seguinte divisão das instruções em classes funcionais:

- As **instruções aritméticas** são **ADDU, SUBU, ADDIU, MULTU, DIVU**;
- As **instruções lógicas** são **AND, OR, XOR, NOR, ANDI, ORI, XORI, SLL, SLLV, SRA, SRAV, SRL e SRLV**;
- As **instruções de movimentação de dados** são **LUI, LBU, LW, SW, SB, SW, MFHI, e MFLO**;
- As **instruções de controle de fluxo de execução** são **BEQ, BGEZ, BLEZ, BNE, J, JAL, JALR, e JR**;
- Existem também **instruções miscelâneas**, **SLT, SLTI, SLTU, e SLTIU**.

4.2 Observações sobre a Semântica de Instruções no Processador MR4

Algumas observações gerais e particulares sobre o conjunto de instruções são apresentadas a seguir.

- A arquitetura foi elaborada para privilegiar a simplicidade do conjunto de instruções sem, contudo, sacrificar sua flexibilidade. Devido à grande limitação de todas as instruções possuírem exatamente o mesmo tamanho, instruções em geral disponíveis em processadores mais poderosos estão ausentes na MR4. Contudo, foi tomado cuidado no projeto desta para que tal funcionalidade possa ser suprida de forma simples via o conceito de pseudo-instruções. **Pseudo-instruções** são instruções inexistentes em uma arquitetura, mas disponibilizadas ao programador em linguagem de montagem. Sua implementação mediante uso de uma seqüência de instruções existentes é feita no código objeto pelo programa montador. Por exemplo, uma instrução capaz de carregar uma constante imediata de 32 bits em um registrador não existe, mas pode ser implementada por uma seqüência de duas instruções, **LUI** e **ORI**. Seria possível a um programa montador para o processador MR4 disponibilizar uma pseudo-instrução **LDI** com um operando imediato de 32 bits, gerando como código para esta uma seqüência de **LUI** e **ORI**. Por exemplo, existe no montador/simulador SPIM uma pseudo-instrução (**LA**, do inglês *load address*) que usa **LUI** e **ORI** para implementar uma função bastante útil, carregar um registrador com o endereço de um rótulo (em inglês, *label*) do programa, seja este da região de dados ou da região de programas.
- As instruções **SLT, SLTU, SLTI e SLTIU** servem para suprir a ausência de biestáveis qualificadores de estado na arquitetura MR4. Elas escrevem no registrador destino da instrução a constante 0 ou a constante 1 para indicar o resultado de uma comparação de magnitude. Após executar uma destas instruções, pode-se testar o valor escrito no registrador de destino contra a constante 0 (por exemplo, usando as instruções **BEQ** ou **BNE**, tendo como um dos operandos a constante 0, disponível no registrador \$0 ou \$zero). Tais seqüências de instrução permitem implementar todas as comparações de igualdade ou magnitude (menor ou maior). **Tarefa 1:** Proponha uma seqüência de instruções do processador MR4 para implementar as seguintes pseudo-instruções: **JGTR, JGER, JLTR e JLER** (respectivamente, **salta se maior, se maior ou igual, se menor e se menor ou igual relativo**). Note-se ainda que as variações destas instruções permitem executar comparações de números naturais (as com sufixo **U**, do inglês *unsigned*) e de números inteiros representados em complemento de 2 (as sem sufixo **U**).

5 REGISTRADORES DO PROCESSADOR MR4 - BLOCO DE DADOS

A partir desta Seção da especificação, apresenta-se uma possível organização para o processador MR4. Esta proposta de organização pode ser aquela implementada no trabalho ou não. Algumas das características mencionadas aqui são partes da arquitetura, sendo assim obrigatórias. Isto será explicitado no texto abaixo, quando for o caso.

A organização aqui proposta deve conter, pelo menos, o seguinte conjunto de registradores:

- **IR** (*instruction register*): armazena o código de operação (*opcode*) da instrução atual e o(s) código(s) do(s) operando(s) desta. Possui 32 bits.
- **PC** (*program counter*): é o contador de programa. Possui 32 bits.
- Um banco de registradores contendo 32 registradores de uso geral, cada um de 32 bits, denominados **\$0** a **\$31**. **Esta é uma característica definida pela arquitetura.** Existe uma denominação textual alternativa para cada um dos registradores. Ela está apresentada na Tabela 2 abaixo, retirada do Apêndice A do livro-texto da disciplina, e disponível na *homepage*. O banco de registradores proposto aqui tem uma porta de escrita e duas de leitura. Isto significa que é possível escrever em apenas um registrador por vez, porém é possível fazer, em

paralelo com a escrita mencionada, duas leituras simultâneas, colocando o conteúdo de um registrador no barramento de saída fonte 1 e o conteúdo de outro registrador (ou o mesmo) no barramento de saída fonte 2.

- **MDR** (*memory data register*): é registrador que recebe dados provenientes da memória ou de outro local que produza um valor que necessite ser escrito no banco de registradores no último ciclo de uma instrução. Possui 32 bits.
- **Hi e Lo**: são dois registradores de 32 bits usados para armazenar o resultado da execução das instruções **MULTU** e **DIVU**. Para manipular estes resultados, existem as instruções **MFHI** e **MFLO**, que movem valores de **Hi** e **Lo** para um registrador de uso geral. A necessidade destes registradores se explica pelo fato de uma multiplicação de dois números de 32 bits gerar um valor de 64 bits, bem como pelo fato de uma divisão (inteira) gerar dois resultados de 32 bits, o quociente e o resto. No caso da instrução **DIVU**, o quociente é colocado em **Lo** e o resto é colocado em **Hi**.

Tabela 2 – Denominação textual de registradores para a arquitetura MR4. Aconselha-se que os registradores a serem usados como registradores de trabalho em programas sejam os registradores 2 a 25 (\$v0 a \$t9). Os demais são reservados para operações de controle do montador, sistema operacional, simulação de pilhas, etc. O montador SPIM aceita a denominação da primeira coluna da tabela, ou da segunda coluna, precedida do caracter \$.

Número (Decimal)	Número (hexa/binário)	Nome	Significado
0	00 / 00000	\$zero	constante 0
1	01 / 00001	\$at	reservado para o programa montador
2	02 / 00010	\$v0	resultado de função
3	03 / 00011	\$v1	resultado de função
4	04 / 00100	\$a0	argumento para função
5	05 / 00101	\$a1	argumento para função
6	06 / 00110	\$a2	argumento para função
7	07 / 00111	\$a3	argumento para função
8	08 / 01000	\$t0	temporário
9	09 / 01001	\$t1	temporário
10	0A / 01010	\$t2	temporário
11	0B / 01011	\$t3	temporário
12	0C / 01100	\$t4	temporário
13	0D / 01101	\$t5	temporário
14	0E / 01110	\$t6	temporário
15	0F / 01111	\$t7	temporário
16	10 / 10000	\$s0	temporário (salvo nas chamadas de função)
17	11 / 10001	\$s1	temporário (salvo nas chamadas de função)
18	12 / 10010	\$s2	temporário (salvo nas chamadas de função)
19	13 / 10011	\$s3	temporário (salvo nas chamadas de função)
20	14 / 10100	\$s4	temporário (salvo nas chamadas de função)
21	15 / 10101	\$s5	temporário (salvo nas chamadas de função)
22	16 / 10110	\$s6	temporário (salvo nas chamadas de função)
23	17 / 10111	\$s7	temporário (salvo nas chamadas de função)
24	18 / 11000	\$t8	temporário
25	19 / 11001	\$t9	temporário
26	1A / 11010	\$k0	reservado para o SO
27	1B / 11011	\$k1	reservado para o SO
28	1C / 11100	\$gp	apontador de área global
29	1D / 11101	\$sp	stack pointer
30	1E / 11110	\$fp	frame pointer
31	1F / 11111	\$ra	registrador para armazenar endereço de retorno de subrotinas

Normalmente, sempre há a necessidade de existirem registradores adicionais, dependendo da organização implementada. Na organização proposta aqui, o valor do registrador **PC** é atualizado após a busca, mas o novo valor não é imediatamente escrito em **PC**, mas em um registrador auxiliar **NPC**, sigla derivada da terminologia *new program counter*, em inglês. Apenas no último ciclo de relógio de uma instrução o valor de **NPC** (ou outro, dependendo da instrução) é escrito no **PC**. Os valores lidos do banco de registradores são armazenados nos registradores **Rs** e **Rt**, e o valor obtido pela execução de uma dada operação lógico-aritmética é armazenado no registrador **RALU**. Apresentam-se detalhes nas Seções a seguir.

6 ORGANIZAÇÃO DO BLOCO DE DADOS DO PROCESSADOR MR4

Com a organização proposta nesta Seção para o processador MR4, a execução de qualquer instrução requer 3 a 5 ciclos de relógio, com a exceção das instruções **MULTU** e **DIVU**. Cada um dos 3/4/5 ciclos executa um conjunto limitado de partes de uma instrução e são assim denominados:

- **Ciclo 1: busca da instrução.** Comum a todas as instruções.

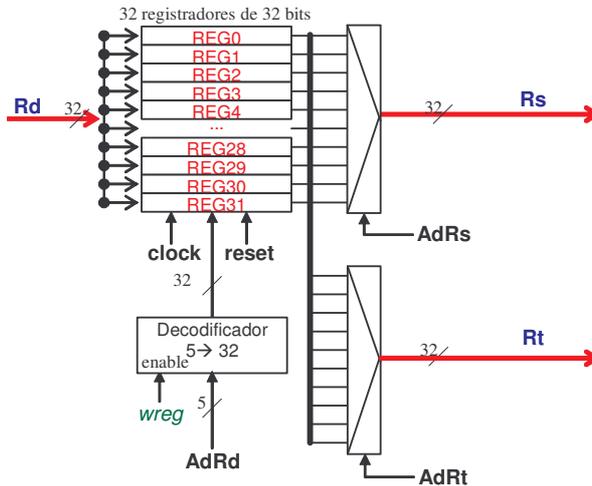


Figura 4 - Diagrama em blocos do banco de registradores de uso geral.

A organização do banco de registradores inclui os 32 registradores em si e a implementação das duas portas de leitura e da porta de escrita, bem como da decodificação do endereço de escrita para geração da habilitação de escrita do registrador em causa. As portas de leitura consistem de multiplexadores (32X32):(1X32), controladas pelos endereços de leitura.

6.2 Resumo das Operações Executadas pela ALU

A Tabela 3 ilustra a estrutura de hardware para efetuar operações da ALU. Esta última é um hardware combinacional que possui duas entradas de dados e uma saída de dados. Além disto, a ALU possui uma entrada de controle para informar a operação a ser realizada em cada instante, derivada da instrução em execução no momento.

O ciclo de operação com a ALU é comum a todas as operações, embora algumas instruções apenas a utilizem para transportar um valor da entrada para a saída. O resultado da operação na ALU é armazenado no registrador **RALU**. A Tabela 3 define quais devem ser as operações da ALU, para cada instrução.

Tabela 3 – Operações da ALU no processador MR4, para cada instrução.

Instruções	Operação da ALU
ADDU, ADDIU, LBU, LW, SB, SW, BEQ, BGEZ, BLEZ, BNE	OP2 + OP1
SUBU	OP2 – OP1
AND, ANDI	OP2 and OP1
OR, ORI	OP2 or OP1
XOR, XORI	OP2 xor OP1
NOR	OP2 nor OP1
SLL	OP1 << OP2[10:6] c/ 0s entrando à direita
SLLV	OP2 << OP1[4:0] c/ 0s entrando à direita
SRA	OP1 >> OP2[10:6] c/ bit 31 entrando à esquerda
SRAV	OP2 >> OP1[4:0] c/ bit 31 entrando à esquerda
SRL	OP1 >> OP2[10:6] c/ 0s entrando à esquerda
SRLV	OP2 >> OP1[4:0] c/ 0s entrando à esquerda
LUI	OP2[15:0] & 0x"0000"
SLT, SLTI	1 se OP1 < OP2 (com sinal), senão 0
SLTU, SLTIU	1 se OP1 < OP2 (sem sinal), senão 0
J, JAL	OP1[31:28] & OP2[27:0]
JR, JALR	OP1
MULTU, DIVU, MFHI, MFLO	-

6.3 Organização para as instruções MULTU e DIVU

As instruções **MULTU** e **DIVU**, **multiplicação e divisão de números inteiros sem sinal**, devido a sua intrínseca maior complexidade, são implementadas fora da ALU. Na presente proposta de implementação as instruções **DIVU** e **MULTU** deverão ser implementadas de maneira estrutural, usando texto VHDL sintetizável. Para tanto, os grupos podem usar um algoritmo de implementação de um multiplicador em hardware, tal como descrito nas transparências 10 a 17 da apresentação disponível no link mostrado abaixo.

<http://www.inf.pucrs.br/~calazans/undergrad/arq1/aulas/aritcomp.pdf>.

O algoritmo de multiplicação ser aquele descrito nas transparências acima. O algoritmo de divisão deve ser o algoritmo sem restauração. Estes e outros algoritmos fáceis de implementar sob a forma de um pequeno bloco de dados e uma máquina de estados de controle estão descritos em algum detalhe no livro-texto da disciplina, na Seção 4.6 da Segunda Edição do livro de Hennessy e Patterson.

7 NÚMERO DE CICLOS PARA AS INSTRUÇÕES DA ORGANIZAÇÃO MR4

Dada a descrição da organização do bloco de dados da MR4 apresentada na Seção anterior, é possível sumarizar o tempo de execução de todas as instruções em termos de ciclos de relógio tomados, o que é então mostrado na Tabela 4 abaixo.

Tabela 4 – Número de ciclos gastos para buscar e executar instruções na organização do processador MR4.

INSTRUÇÃO	NUMERO DE CICLOS	INSTRUÇÃO	NUMERO DE CICLOS
ADDU	4	SB	4
SUBU	4	SW	4
AND	4	SLT	4
OR	4	SLTU	4
XOR	4	SLTI	4
NOR	4	SLTIU	4
SLL	4	BEQ	4
SLLV	4	BGEZ	4
SRA	4	BLEZ	4
SRAV	4	BNE	4
SRL	4	J	4
SRLV	4	JAL	4
ADDIU	4	JALR	4
ANDI	4	JR	4
ORI	4	MULTU	67 ²
XORI	4	DIVU	67 ³
LUI	4	MFHI	4
LBU	5	MFLO	4
LW	5	-	-

8 BLOCO DE CONTROLE

Para comandar a execução de instruções neste processador, define-se uma máquina de estados de controle. A Figura 5 ilustra esta máquina de estados em linhas gerais, onde o próximo estado é função apenas do estado atual e da instrução armazenada no registrador **IR**. Também se indica nesta Figura quais registradores são alterados em cada estado. Não se mostra as ativações dos sinais de controle dos multiplexadores, ou as operações da ALU.

A função dos 8 estados mostrados na Figura é:

- **Idle:** estado inicial após o reset, serve para garantir que a primeira borda de subida de relógio após este sinal defina o início da operação do processador MR4;
- **Sfetch:** primeiro ciclo, busca de instrução;
- **SReg:** segundo ciclo, leitura dos registradores fonte;
- **SAlu:** terceiro ciclo, operação com a ALU e eventual comparação no comparador (em paralelo);

² Valor típico, depende do algoritmo escolhido.

³ Valor típico, depende do algoritmo escolhido.

- **Wbk:** quarto ciclo para a maioria das instruções, onde se escreve o resultado no banco de registradores e atualiza-se o contador de programa (quinto ciclo para as instruções **LW** e **LBU**);
- **Sld:** quarto ciclo das instruções **LW** e **LBU**, onde se lê um dado da memória de dados externa;
- **Sst:** último ciclo das instruções **SW** e **SB**, onde se escreve um dado na memória de dados externa;
- **Ssalta:** último ciclo das instruções de salto condicional ou incondicional, apenas atualiza valor do **PC**.

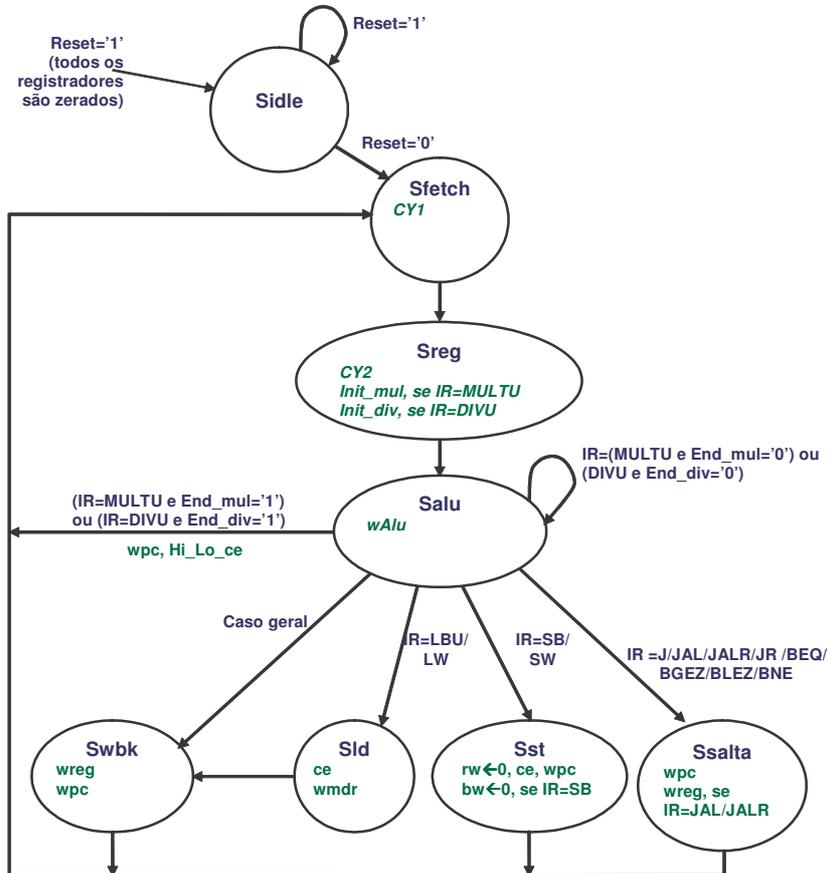


Figura 5 - Máquina de estados de controle para organização MR4 proposta.

9 TRABALHO PRÁTICO A SER DESENVOLVIDO

Implementar em VHDL o processador multiciclo, descrito nas Seções anteriores. O bloco de dados deve ter uma descrição semelhante ao processador Cleópatra e o bloco de controle deve ser implementado através de uma máquina de estados, a exemplo do que foi feito com a Cleópatra. A nota final deste trabalho dará **ENORME ÊNFASE** à execução correta da simulação. Assim, aconselha-se testar cada módulo implementado do hardware. A nota de uma descrição completa sem nenhuma simulação tenderá a 0 (zero). A nota de uma descrição incompleta com boas simulações de cada módulo implementado tenderá ao máximo valor para a parte realizada do trabalho. As regras do jogo são:

- O trabalho de implementação pode ser realizado por até 3 alunos (*grupo*). Mais do que 3 alunos no grupo implicará automaticamente a não avaliação do trabalho, e conseqüente nota.
- A apresentação será oral, teórico-prática, frente ao computador, onde o *grupo* deverá explicar ao professor o projeto, a simulação e a implementação. A avaliação de cada membro do grupo será individual, baseada no desempenho durante a apresentação. Questões individuais serão colocadas aos membros do grupo. Após a apresentação, entregar ao professor um disquete com o projeto (fonte do processador, fonte do *test_bench* e programas de teste em código objeto e linguagem de montagem, ambos **adequadamente comentados**).
- Cada *grupo* deve desenvolver uma aplicação para a MR4, com utilização de pelo menos duas subrotinas e pelo menos uma chamada aninhada de subrotina. Pelo menos uma das subrotinas deve ser chamada em duas linhas distintas da aplicação. A aplicação deve ser um programa em linguagem de montagem da MR4

(aquelas descritas na Tabela 1) contendo, no mínimo, 40 instruções **úteis**. Instruções **úteis** são aquelas que realmente fazem parte do algoritmo, e não instruções usadas para inicializar registradores, tais como combinações de instruções **LUI** e **ORI**. Linhas de comentário também não são, obviamente, contabilizadas como instruções úteis. **Implementações que não respeitem estas regras não serão avaliadas**. Esta aplicação deve ser validada via o montador/simulador do SPIM, limitado ao emprego do conjunto de instruções da MR4, e deve ser entregue no prazo máximo estabelecido na homepage da disciplina, após a disponibilização da especificação do trabalho. Note que na programação é possível usar diretivas do montador à vontade, desde que esta não produzam código objeto contendo códigos de instruções inexistentes na MR4. Consulte esta homepage para saber a data limite máxima para entrega desta parte do trabalho para Turmas específicas. Esta aplicação valerá 30% da nota do Trabalho Prático. A avaliação da aplicação será calculada em 5 itens, mostrados a seguir, em ordem decrescente de importância relativa: (1) a criatividade na escolha do problema a resolver, (2) a capacidade de acrescentar comentários úteis no código fonte, (3) o estilo de programação, (4) a extensão do programa, bem como (5) a otimalidade do código.

- As apresentações finais dos trabalhos ocorrerão em data divulgada na homepage da disciplina (2 aulas para cada turma). Sistemática: metade dos grupos no primeiro dia, metade no segundo. Para marcar dia contatar o professor, desde que o projeto esteja avançado (tipicamente, 50% pronto). A este caberá julgar se o trabalho está adiantado o suficiente para permitir a marcação da data de apresentação. As demais apresentações serão marcadas pelo professor no máximo 7 dias antes da primeira data de apresentação prevista, via sorteio entre os grupos restantes.
- Todos os trabalhos devem ser entregues para o professor no primeiro dia de avaliação em meio magnético (ver dia específico para cada turma na *homepage*), conforme especificado anteriormente, durante o período da aula. Um representante do grupo deverá assinar a ata de entrega do trabalho. **Trabalhos entregues fora desta data e do horário estipulado não serão avaliados**. Não serão permitidas substituições, modificações ou alterações futuras no trabalho após a entrega para o professor.
- Composição média da nota do Trabalho:

BLOCO DE DADOS	BLOCO DE CONTROLE	Simulação VHDL de instruções básicas	Simulação VHDL novas instruções	Aplicação MR4
16%	16%	13%	15%	40%

- **Obs: Instruções básicas são todas as instruções da MR4, exceto as novas em relação à versão anterior do processador (neste caso, SLLV, SRA, SRAV, SRLV, DIVU).**

Recomenda-se desenvolver inicialmente o bloco de dados, depois iniciar o bloco de controle, realizando-se simulações parciais para validar os módulos individuais. De nada adianta um código dito completo, caso não se tenha realizado simulações corretas.

10 PROGRAMA PARA TESTAR TODAS AS INSTRUÇÕES DA MR4

Recomenda-se escrever os programas em linguagem de montagem (*assembly*), gerando-se o código objeto automaticamente, a partir do montador/simulador SPIM. A ferramenta de simulação, assim como documentação de como utilizar a ferramenta encontra-se na *homepage* da disciplina.

A Figura 6 abaixo apresenta um programa escrito em linguagem de montagem para o processador MR4, contendo pelo menos uma instância de cada uma das instruções do processador. Recomenda-se o uso do simulador SPIM para executar este programa e verificar sua funcionalidade, descrita linha a linha do programa, através do campo de comentários.

```
#####  
## ARQUITETURA MR4 - teste de todas as instruções  
## Autor: Ney Calazans (calazans@inf.pucrs.br)  
## Teste é exaustivo quanto ao total de instruções, não quanto ao teste em si  
## Última alteração: 24/11/2005  
#####  
.text # Declaração de início do segmento de texto  
.globl main # Declaração de que o rótulo main é global  
#####  
# testes de instruções individuais  
#####  
main: lui $t0,0xf3 #  
ori $t0,$t0,0x23 # $t0<= 0x00f30023  
lui $t1,0x52 #  
ori $t1,$t1,0xe2 # $t1<= 0x005200e2  
lui $t2,0x00 #  
ori $t2,$t2,0x8f # $t2<= 0x0000008f  
beq $t1,$t2,loop # Obviamente, esta instrução nunca deve saltar  
bne $t1,$t2,next_i # Obviamente, esta instrução sempre deve saltar  
addiu $t2,$t2,0x8f # Obviamente, esta instrução nunca deve executar  
next_i: addu $t3,$t0,$t1 # $t3<= 0x00f30023 + 0x005200e2 = 0x01450105  
subu $t4,$t0,$t1 # $t4<= 0x00f30023 - 0x005200e2 = 0x00a0ff41  
subu $t5,$t1,$t1 # $t5<= 0x0  
and $t6,$t0,$t1 # $t6<= 0x00f30023 and 0x005200e2 = 0x00520022  
or $t7,$t0,$t1 # $t7<= 0x00f30023 or 0x005200e2 = 0x00f300e3  
xor $t8,$t0,$t1 # $t8<= 0x00f30023 xor 0x005200e2 = 0x00a100c1  
nor $t9,$t0,$t1 # $t9<= 0x00f30023 nor 0x005200e2 = 0xff0cff1c  
multu $t0,$t1 # Hi & Lo <= 0x00f30023 * 0x005200e2 = 0x00004dd6e1bc1ee6  
mfhi $s0 # $s0<= 0x00004dd6  
mflo $s1 # $s1<= 0xelb1cee6  
divu $t0,$t1 # Hi,Lo<= 0x00f30023 mod,/ 0x005200e2 = 0x4efe5f,0x00000002  
addiu $t0,$t0,0x00ab # $t0<= 0x00f30023 + 0x000000ab = 0x00f300ce  
andi $t0,$t0,0x00ab # $t0<= 0x00f300ce and 0x000000ab = 0x0000008a  
xori $t0,$t0,0xffab # $t0<= 0x0000008a xor 0x0000ffab = 0x0000ff21  
sll $t0,$t0,4 # $t0<= 0x000ff210 (deslocado 4 bits para a esquerda)  
srl $t0,$t0,9 # $t0<= 0x000007f9 (deslocado 9 bits para a direita)  
addiu $s2,$zero,8 # $s2<= 0x00000008  
sllv $t0,$t9,$s2 # $t0<= 0x0007f900  
sllv $t0,$t0,$s2 # $t0<= 0x07f90000  
sllv $t0,$t0,$s2 # $t0<= 0xf9000000  
sra $t0,$t0,4 # $t0<= 0xff900000  
srav $t0,$t0,$s2 # $t0<= 0xffff9000  
srlv $t0,$t0,$s2 # $t0<= 0x00ffff90  
la $t0,array # coloca em $t0 o endereço inicial do vetor array (0x10010000)  
lbu $t1,6($t0) # $t1<= 0x000000ef (primeiro byte é terceiro byte do segundo elemento)  
xori $t1,$t1,0xff # $t1<= 0x00000010, inverte byte inferior  
sb $t1,6($t0) # segundo byte do segundo elemento do vetor <= 10 (resto não muda)  
# CUIDADO, mudou elemento do array a ser processado por soma_ct  
addiu $t0,$zero,0x1 # $t0<= 0x00000001  
subu $t0,$zero,$t0 # $t0<= 0xffffffff  
bgez $t0,loop # Esta instrução nunca deve saltar, pois $t0 = -1  
slt $t3,$t0,$t1 # $t3<= 0x00000001, pois -1 < 10  
sltu $t3,$t0,$t1 # $t3<= 0x00000000, pois (2^32)-1 > 10  
slti $t3,$t0,0x1 # $t3<= 0x00000001, pois -1 < 1  
sltiu $t3,$t0,0x1 # $t3<= 0x00000000, pois (2^32)-1 > 1  
#####  
# soma uma constante a um vetor  
#####  
soma_ct:la $t0,array # coloca em $t0 o endereço do vetor (0x10010000)  
la $t1,size # coloca em $t1 o endereço do tamanho do vetor  
lw $t1,0($t1) # coloca em $t1 o tamanho do vetor  
la $t2,const # coloca em $t2 o endereço da constante  
lw $t2,0($t2) # coloca em $t2 a constante  
loop: blez $t1,end_add # se/quando tamanho é/torna-se 0, fim do processamento  
lw $t3,0($t0) # coloca em $t3 o próximo elemento do vetor  
addu $t3,$t3,$t2 # soma constante  
sw $t3,0($t0) # atualiza no vetor o valor do elemento  
addiu $t0,$t0,4 # atualiza ponteiro do vetor. Lembrar, 1 palavra=4 posições de memória  
addiu $t1,$t1,-1 # decrementa contador de tamanho do vetor  
j loop # continua execução  
#####  
# teste de subrotinas aninhadas  
#####  
end_add:addiu $sp,$sp,-4 # assume-se $sp inicializado, aloca espaço na pilha  
# no SPIM $sp inicializado com 0x7FFFEFFC. Usamos 0x10010800.  
sw $ra,0($sp) # salva endereço de retorno de quem chamou (trap handler)  
jal sum_tst # salta para subrotina sum_tst  
lw $ra,0($sp) # ao retornar, recupera endereço de retorno da pilha  
addiu $sp,$sp,4 # atualiza apontador de pilha  
end: jr $ra # volta para o "sistema operacional" FIM DO PROGRAMA AQUI  
# Início da primeira subrotina: sum_tst  
sum_tst:la $t0,var_a # pega endereço da primeira variável (pseudo-instrução)  
lw $t0,0($t0) # toma o valor de var_a e coloca em $t0  
la $t1,var_b # pega endereço da segunda variável (pseudo-instrução)  
lw $t1,0($t1) # toma o valor de var_b e coloca em $t1  
addu $t2,$t1,$t0 # soma var_a com var_b e coloca resultado em $t2  
addiu $sp,$sp,-8 # aloca espaço na pilha  
sw $t2,0($sp) # no topo da pilha coloca o resultado da soma  
sw $ra,4($sp) # abaixo do topo coloca o endereço de retorno  
la $t3,ver_ev # pega endereço da subrotina ver_ev (pseudo-instrução)  
jalr $ra,$t3 # chama subrotina que verifica se resultado da soma é par  
lw $ra,4($sp) # ao retornar, recupera endereço de retorno da pilha  
addiu $sp,$sp,8 # atualiza apontador de pilha  
jr $ra # PROGRAMA ACABA AQUI. Retorna para quem chamou  
# Início da segunda subrotina: ver_ev. Trata-se de subrotina folha, que não usa pilha  
ver_ev: lw $t3,0($sp) # tira dados do topo da pilha (parâmetro)  
andi $t3,$t3,1 # $t3 <= 1 se parâmetro é ímpar, 0 caso contrário  
jr $ra # e retorna  
#####  
.data # área de dados  
#####  
# para trecho que soma constante a vetor  
array: .word 0xabcd03 0xcd03 0xfab35 0xbadcf0 0xcfebacd 0xf0badc77 0xdefabc53 0xcbafe45  
# terceiro byte da segunda palavra (0xef) vira 0x10 antes de exec soma_ct  
size: .word 0x8  
const: .word 0xffffffff # constante -1 em complemento de 2  
# para trecho de teste de chamadas de subrotinas  
var_a: .word 0xff #  
var_b: .word 0x100 #
```

Figura 6 - Programa exemplo para teste da arquitetura MR4.

O simulador SPIM gera um arquivo de saída resultado da execução que contém o código objeto do programa, entre outras informações. Este arquivo pode ser salvo usando a opção de menu File --> Save Log File... do simulador. Ele será útil para gerar a entrada do processo de simulação da implementação de hardware a ser realizada neste trabalho.

A Figura 7 mostra a interface gráfica do simulador, composta de 4 subjanelas:

- a subjanela Registers mostra os conteúdos dos registradores da arquitetura. Ignore aqueles não definidos para a arquitetura MR4, tais como os registradores de ponto flutuante;
- a subjanela Text Segment mostra as instruções do programa ora em simulação;
- a subjanela Data Segment mostra os dados do programa ora em simulação;
- a subjanela Messages mostra mensagens que o montador/simulador gera para o usuário durante a carga, montagem e execução de programas.

A ferramenta de montagem tem como entrada o nome do programa em linguagem descrito em linguagem de montagem (tipicamente com nome <file>.asm) e o nome da arquitetura. É possível gerar um arquivo de saída com o conteúdo de todas as janelas, incluindo o código objeto gerado. Este mais tarde poderá, após algum tratamento manual, ser usado como entrada para a simulação VHDL do processador.

A ferramenta de montagem é transparente para o usuário, pois a mesma está integrada ao simulador. O código objeto é gerado como subproduto da execução do montador após a leitura de um arquivo texto descrito em linguagem de montagem do MIPS.

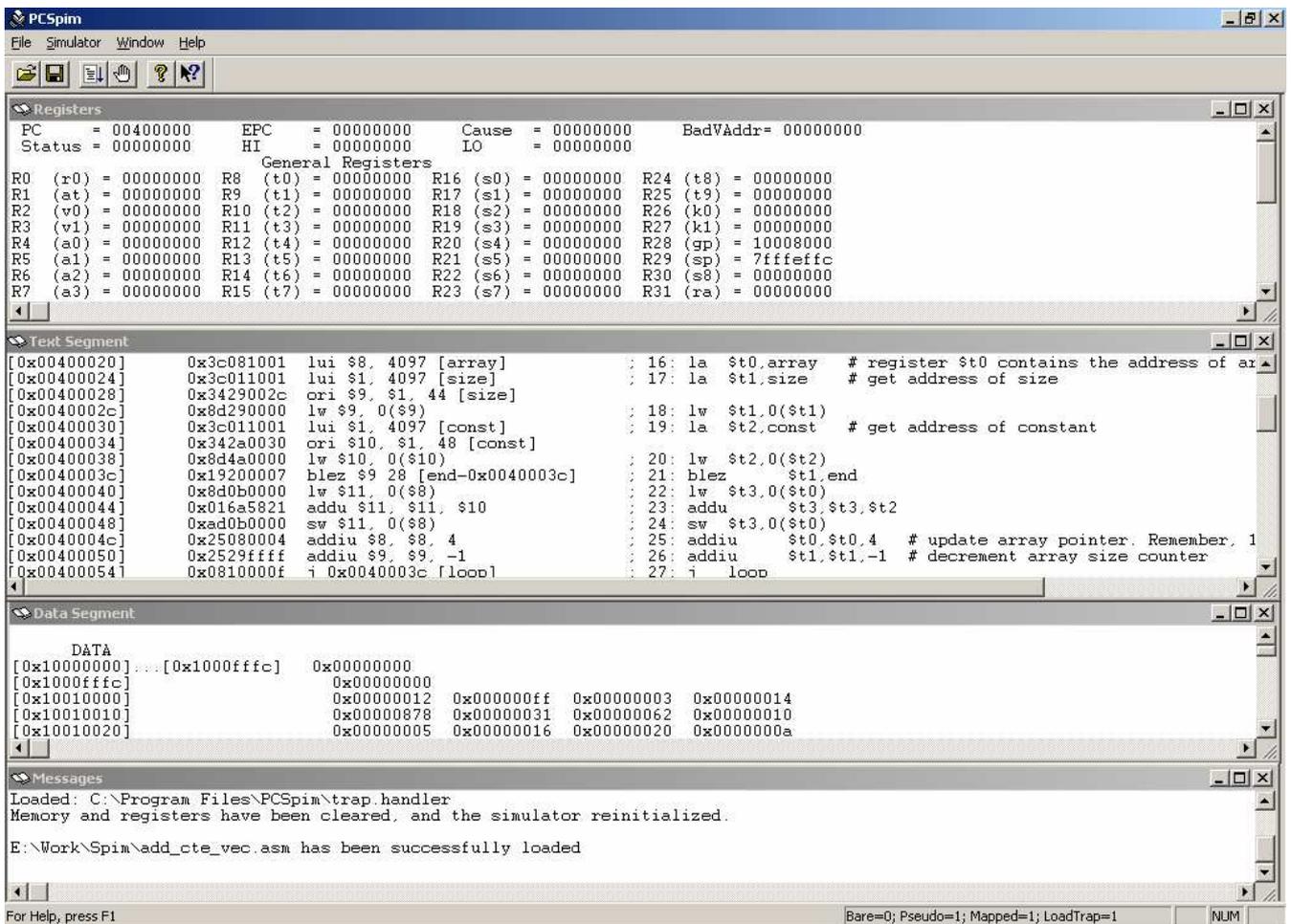


Figura 7 - Interface gráfica do montador/simulador SPIM.