

# O PROCESSADOR MULTICICLO MIPS\_S

## 1 CARACTERÍSTICAS GERAIS DA ARQUITETURA MIPS

- A arquitetura MIPS é do tipo *load-store*, ou seja, as instruções lógicas e aritméticas são executadas exclusivamente entre registradores da arquitetura ou entre constantes imediatas e registradores. As instruções de acesso à memória só executam ou uma leitura da memória (*load*) ou uma escrita na memória (*store*).
- Devido à característica *load-store*, o processador disponibiliza um conjunto relativamente grande de registradores, para reduzir o número de acessos à memória externa, pois estes últimos representam perda de desempenho significativa em relação a operações entre registradores internos ao processador. Esta característica difere de arquiteturas baseadas em acumulador, que mantêm todos os dados em memória, realizando operações aritméticas entre um conteúdo que está em memória e um, ou poucos registradores de dados, os 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 menor que o tempo de acesso a uma posição de memória (ou seja, 10 a 100 vezes menor, dependendo de aspectos tais como tecnologia de implementação do processador e das memórias, padrão de comunicação processador-memória etc.), percebe-se o ganho de desempenho que arquiteturas *load-store* podem auferir em relação a arquiteturas baseadas em acumulador(es).
- Dados e endereços na arquitetura MIPS são de 32 bits. Logo, diz-se que a **palavra** deste processador é de 32 bits, ou simplesmente que se trata de um processador de 32 bits. Obviamente, existem processadores com outro tamanho de palavra, tal como o 8051 da Intel (8 bits), o TMS9900 da Texas (16 bits) e o Itanium da Intel (64 bits).
- O endereçamento de memória no MIPS é orientado a byte, ou seja, cada endereço de memória é um identificador de uma posição de memória onde se armazena apenas 1 byte=8 bits. Então, uma palavra do processador armazenada em memória ocupa 4 posições de memória, tipicamente em quatro endereços consecutivos.
- O banco de registradores da arquitetura 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, que pode ser referenciada como um registrador ordinário. Este é disponibilizado para instruções que empregam o modo de endereçamento a registrador entre outros, e que necessitem usar o valor 0, muito empregado na prática de programação.
- No MIPS há um tamanho regular definido para as instruções. Todas possuem exatamente o mesmo tamanho, e ocupam 1 palavra em memória (32 bits), ou seja, o equivalente a 4 endereços consecutivos em memória. 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 uma instrução possui os dois últimos bits iguais a 0 (e.g. endereços 0, 4, 16, 32008 etc. que podem ser representados em decimal ou em hexadecimal).
- Os modos de endereçamento mais importantes nesta arquitetura são os modos *a registrador*, *base-deslocamento* e *relativo ao PC*.
- O processador não possui suporte em hardware para gerenciar estruturas do tipo pilha em memória, mas existe um registrador do banco, o **\$sp** ou **\$29**, que associado a instruções operando sobre este, em particular com duas instruções específicas (**JAL/JALR**), permite simular estruturas do tipo pilha em memória para por exemplo realizar o salvamento de contexto de funções e sub-rotinas.
- Não existem registradores para armazenar explicitamente qualificadores sobre a execução de operações, tais como as informações de que a execução de uma instrução resultou na constante 0, ou que uma instrução aritmética gerou vai-um (em inglês, *carry*), ou ainda que seu resultado não é correto porque necessita recursos de armazenamento de maior capacidade que os disponíveis no processador (situação chamada de transbordo, ou, em inglês, *overflow*). A detecção destas situações deve ser realizada usando instruções específicas de comparação disponíveis na arquitetura especificamente para isto (ver o comportamento das instruções cujo mnemônico possui o prefixo **SLT**).

O que é descrito aqui é um processador que implementa um subconjunto da arquitetura do MIPS (donde o nome **MIPS\_S**, abreviatura de **MIPS Subset** ou **Subconjunto do MIPS**). Trata-se praticamente de uma máquina RISC completa. Faltam, contudo algumas características que existem em qualquer RISC, tal como *pipelines*, assunto deixado para ser introduzido e estudado em profundidade em outra oportunidade.

## 2 A RELAÇÃO PROCESSADOR - MEMÓRIAS - AMBIENTE EXTERNO

A Figura 1(a) mostra a visão de alto nível de uma proposta de organização para implementar o processador MIPS\_S. Nela, está explicitada a relação entre processador, suas memórias externas e o 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 é uma **organização Harvard**, ou seja, o processador usa interfaces distintas para comunicação com as memórias de instruções e de dados. Outra possibilidade seria o uso de uma interface de memória unificada para instruções e dados, caracterizando uma **organização von Neumann**. Segue agora uma breve discussão da organização MIPS\_S. Ela será detalhada em Seções posteriores deste documento.

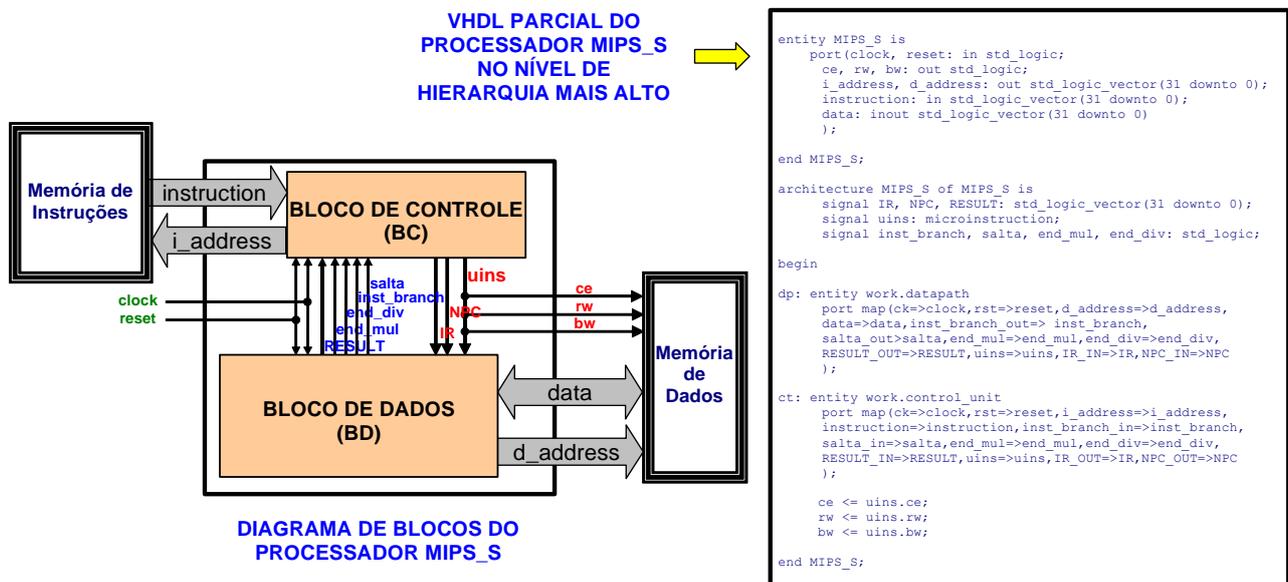


Figura 1 – Proposta de uma organização para o processador MIPS\_S. Ilustra-se: (a) a conectividade entre o processador, ambiente externo, memórias externas e (b) o esboço da descrição VHDL de nível hierárquico mais alto para a organização.

O processador MIPS\_S recebe do mundo externo dois sinais de controle. O primeiro, denominado **clock**, sincroniza todos os eventos internos do processador. O sinal **reset** leva o processador a reiniciar a execução de instruções a partir do endereço 0x00400000<sup>1</sup> de memória (este endereço é definido por compatibilidade com o endereço assumido pelo simulador MARS como aquele de início da área de programas). O sinal **reset** pode ser usado para provocar o reinício da operação do processador.

Os sinais providos pelo processador MIPS\_S para a troca de informações com o mundo externo, representado pelas memórias e pela geração dos sinais **clock** e **reset**, são:

- ◆ **i\_address** – um barramento unidirecional de 32 bits: define sempre o endereço da posição de memória contendo a instrução a ser buscada a seguir;
- ◆ **instruction** – um barramento unidirecional de 32 bits: contém sempre o código objeto da instrução contida na posição de memória apontada por **i\_address**;
- ◆ **d\_address** – um barramento unidirecional de 32 bits: contém o endereço da posição de memória a ser lida ou escrita, da ou para a memória de dados, respectivamente;
- ◆ **data** - um barramento bidirecional de 32 bits: transporta dados do ou para o processador MIPS\_S;

<sup>1</sup> A notação introduzida aqui, 0x00400000 é uma forma de representar números na base hexadecimal. Adota-se a convenção usada no montador/simulador MARS do MIPS, que emprega o prefixo “0x” para indicar que um número está representado na base 16.

Além dos sinais que permitem a interação com o mundo externo, os principais componentes da organização do processador MIPS\_S, o Bloco de Controle (BC) e o Bloco de Dados (BD) interagem através de um conjunto de sinais específicos. São 3 sinais que passam do BC para o BD e 5 sinais qualificadores que passam do BD para o BC.

Os sinais gerados no BC e enviados ao BD são:

- ❖ A palavra de microinstrução (**uins**), para comandar a execução passo a passo das instruções pelo BD. A microinstrução é responsável por especificar cada uma das ações unitárias que serão executadas pelo hardware do BD a cada ciclo de relógio, também chamadas de *micro-operações*. Exemplos destas são cada uma das três seleções de registradores a serem escritos/lidos no/do banco de registradores, a operação que a Unidade Lógico-aritmética (ULA ou, usando a terminologia em inglês, *Arithmetic Logic Unit* ou *ALU*) executará, e os sinais de controle de acesso à memória de dados externa;
- ❖ O endereço potencial da próxima instrução a ser executada (conteúdo do registrador **NPC**, localizado no BC);
- ❖ O código objeto da instrução corrente (conteúdo do registrador **IR**, localizado no BC).

Os sinais gerados no BD são:

- ❖ O sinal **salta**, que contém o resultado do teste de condições de salto nas instruções de salto condicional;
- ❖ O sinal **inst\_branch**, que qualifica se a instrução ora em execução é ou não um salto condicional;
- ❖ Os qualificadores **end\_mul** e **end\_div**, produzidos no BD pelos operadores de multiplicação e divisão, que informam ao BC o final da execução destas operações com um pulso de duração de 1 ciclo de relógio em '1'. Isto é necessário porque trata-se de operações mais complexas que gastam, nesta organização, múltiplos ciclos de relógio para executar. O BC comanda o início da execução delas e aguarda pela informação de seu término para seguir com o fluxo normal de ciclos de controle;
- ❖ O sinal **RESULT\_OUT**, que eventualmente contém um endereço calculado durante instruções de salto (incondicionais ou condicionais), valor este possivelmente escrito no registrador **PC**, que se encontra no BC.

É importante ressaltar que o BC e o BD operam sempre e apenas na borda de subida do sinal de relógio (**clock**). A cada borda de subida do **clock** (e após a desativação do sinal de **reset**), o BC gera uma palavra de microinstrução, e na borda de subida seguinte o BD executa as ações comandadas por esta, modificando seus registradores. É necessário que sempre existam dados estáveis antes das bordas de subida relógio em cada um dos blocos<sup>2</sup>.

A Figura 1(b) representa um exemplo de codificação textual do nível mais alto da hierarquia do processador, em linguagem VHDL. Nesta Figura, deve-se notar que o BC e o BD são instanciados e conectados entre si por *sinais*, conforme definido pelos comandos *port map* no processo de instanciação.

## 2.1 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 às quais o processador tem acesso na forma de uma tabela. Cada posição deste mapa possui um endereço associado e armazena um valor. Na arquitetura MIPS\_S, ambos os mapas (de instruções e de dados) possuem endereços de 32 bits e conteúdos de 8 bits associados a cada posição da tabela. Assim, diz-se que as memórias de instruções e de dados são **endereçadas a byte** (1 byte = 8 bits) e cada um destes mapas possui  $2^{32}$  posições (4.294.967.296 posições, valor normalmente denominado 4Gigabytes ou 4GB, correspondendo ao número de endereços diferentes de 32 bits, entre 0x00000000 e 0xFFFFFFFF).

A memória de instruções armazena apenas instruções. Cada instrução na MIPS\_S ocupa exatamente 32 bits. Assim, uma instrução ocupa 4 posições de memória. Como visto na Figura 1, contudo, o barramento de instruções é de 32 bits. 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 outra limitação imposta para facilitar o acesso à memória de instruções é **alinhar instruções em uma fronteira inteira de palavra**. Esta expressão significa que cada instrução da MIPS\_S só pode começar a partir de um endereço múltiplo de 4, que correspondem àqueles cujos 2 últimos bits são 00. Assim, por exemplo, uma única instrução pode ocupar os endereços de **0xAAA00004** a **0xAAA00007** na memória de instruções, mas nunca os endereços **0xFF004677** a **0xFF00467A** da mesma memória, embora ambas as faixas correspondam a 4 bytes

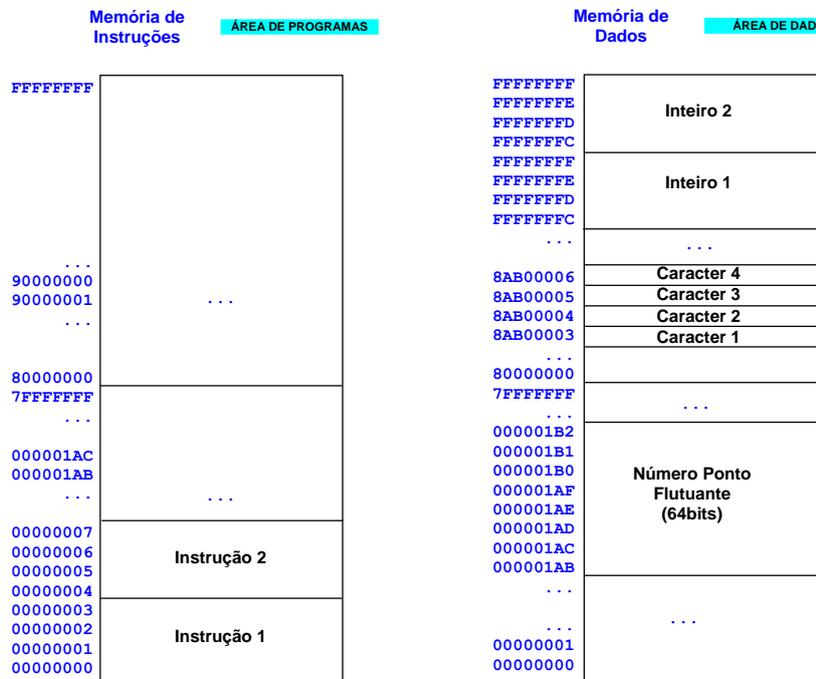
---

<sup>2</sup> Deve estar claro que esta última afirmação se verifica apenas se a frequência de relógio for suficientemente baixa para permitir que os sinais estabilizem em no máximo um período de relógio (entre duas bordas de subida consecutivas do **clock**).

consecutivos na memória. Dada esta situação, o barramento **i\_address** não precisaria ter 32 bits, mas apenas 30, pois os 2 últimos bits deste endereço valem sempre '00'.

Não existem sinais de controle para acesso à memória de instruções além do endereço em **i\_address**. Isto não é necessário, pois não há fluxo bidirecional de informação entre o processador e esta memória. A memória de instruções é vista pela organização MIPS\_S como uma memória de apenas leitura, que fornece informações na sua saída (instruções) a partir do estabelecimento do endereço de memória pelo processador, via o barramento **i\_address**.

A memória de dados possui uma interface um pouco mais elaborada, devido ao fato de poder ser lida ou escrita. Além disto, os dados lidos ou gravados nesta memória podem ter tamanho diversos. Estas necessidades fazem com que o acesso à informação deva ter um formato mais flexível. Por exemplo, um inteiro típico ocupa 32 bits, mas um caractere ocupa exatamente 1 byte, e um número em ponto flutuante pode ser de 32 ou de 64 bits. Além disto, estruturas complexas, tais como aquelas usadas em programação de alto nível (e.g. **struct** de C/Java ou **record** de Pascal, e listas em LISP), podem ter tamanho variável, em alguns casos até sem limitações de tamanho mínimo ou máximo. Vetores e arranjos multidimensionais de números ou caracteres e de estruturas complexas complicam ainda mais a situação. Logo, a interface de acesso da memória de dados deve ser flexível para prover desempenho e proporcionar economia de espaço. Ainda assim, o barramento de dados (**data**) é fixo, de 32 bits. Lê-se sempre 4 bytes (ou posições) de memória de cada vez. O que é lido pode ser, por exemplo, 1 inteiro ou 4 caracteres ou metade de um número em ponto flutuante de precisão dupla. Como a leitura é alinhada em qualquer fronteira, é até mesmo possível ler metade de um inteiro e dois caracteres nos 32 bits.



**Figura 2 - Ilustração da estrutura dos mapas de memória do MIPS\_S. Cada mapa possui 4Gposições de 1 byte. Cada instrução ocupa exatamente 4 bytes consecutivos na memória de instruções, e sempre começa em um endereço onde os 2 últimos bits são '00' (configurando um acesso alinhado a palavra).**

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. 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 4 bytes (ou seja, na posição explicitada na instrução e nas três seguintes). Isto ocorre quando se faz uma escrita de palavra com **bw** = 1.

## 2.2 Uma palavra sobre “Endianismo”

Processadores podem numerar os bytes dentro de uma palavra fazendo o byte com menor índice ser 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 (mais precisamente, 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 (MARS) opera com ambas as ordens. A ordem específica do MARS é determinada pela ordem dos bytes da máquina que executa o simulador. Assim, ao rodar o simulador em uma máquina baseada em processador que use a arquitetura x86 da Intel, MARS é *little-endian* (pois a arquitetura Intel x86 é *little-endian*), enquanto ao rodar MARS em uma estação de trabalho Sun (baseada no processador SPARC), MARS é *big-endian* (pois a arquitetura SPARC é *big-endian*).

### 3 CONJUNTO DE INSTRUÇÕES

A Tabela 1 resume o conjunto de instruções da arquitetura MIPS\_S. Notem que todas as 37 instruções listadas aqui existem exatamente na mesma forma na arquitetura MIPS, justificando dizer que a arquitetura MIPS\_S é um subconjunto da arquitetura MIPS. Usa-se a seguir as seguintes convenções:

#### Convenções Utilizadas na Tabela 1:

- ◆ **Rd (destination register)** é o registrador usado na maioria das instruções de três operandos como destino dos dados processados, especificado por um código binário de 5 bits. Veja a Seção 4 deste documento e para um detalhamento mais amplo refira-se ao Apêndice A do livro texto da disciplina;
- ◆ **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. Veja a Seção 4;
- ◆ Registradores específicos da máquina são indicados por **\$xx**, onde **xx** é o número decimal do registrador.;
- ◆ Os registradores **HI** e **LO** são especiais (ficam fora do banco de registradores de uso geral), e usados para armazenar resultados das operações de multiplicação e divisão (**MULTU** e **DIVU**, respectivamente);
- ◆ O sinal **←** é usado para designar atribuição (escrita) de valores resultantes da avaliação da expressão à direita do sinal ao registrador ou à 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** (do inglês, *shift amount*) representa a quantidade de bits a deslocar nas instruções **SLL**, **SLLV**, **SRA**, **SRAV**, **SRL** e **SRLV**. O operador **mod** representa o resto da divisão inteira do elemento à esquerda do **mod** pelo elemento à direita deste;
- ◆ 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 ao valor representado pelo vetor menor. Consiste em copiar o bit de sinal do vetor, ou seja, o seu bit mais significativo, localizado na extrema esquerda deste vetor, 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, representado em 16 bits), a extensão de sinal transforma este vetor em 1111 1111 1111 1111 1111 1111 1111 1111 (-1 em complemento de 2, representado em 32 bits). O leitor deve perceber que esta operação é trivialmente correta para números positivos, onde o bit de sinal estendido é 0. Quando se menciona extensão de sinal, os valores imediatos são interpretados como números em complemento de 2. Caso contrário, os números são representações binárias puras, como ocorre na instrução **J<sup>3</sup>**;
- ◆ 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-os por 4 antes de operar (o que é realizado deslocando o operando dois bits à esquerda, inserindo dois 0s). Isto vale apenas para as instruções **BEQ**, **BGEZ**, **BLEZ**, **BNE**, **J** e **JAL**.

<sup>3</sup> **CUIDADO!!!:** Existe um outro simulador bem conhecido, o SPIM, onde a extensão de sinal é feita sobre os 14 bits, e não sobre 16 bits (pode ser um *bug* do simulador ou uma definição da arquitetura; O importante é que isto afeta a implementação do processador aqui especificado).

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

Instrução	FORMATO DA INSTRUÇÃO						SEMÂNTICA DA AÇÃO DA INSTRUÇÃO
	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 ← NPC + (lmed16 & "00" com sinal estendido), se Rs=Rt
BGEZ Rs, rótulo	01	Rs	01		lmed16		PC ← NPC + (lmed16 & "00" com sinal estendido), se Rs=>0
BLEZ Rs, rótulo	06	Rs	00		lmed16		PC ← NPC + (lmed16 & "00" com sinal estendido), se Rs<=0
BNE Rs, Rt, rótulo	05	Rs	Rt		lmed16		PC ← NPC + (lmed16 & "00" com sinal estendido), se Rs≠Rt
J rótulo	02				lmed26		PC ← NPC[31:28] & lmed26 & "00"
JAL rótulo	03				lmed26		\$31 ← NPC; PC ← NPC[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 ← Rs * Rt
DIVU Rs, Rt	00	Rs	Rt		000	1B	LO ← Rs / Rt; HI ← Rs mod Rt
MFHI Rd	00		000	Rd	00	10	Rd ← HI
MFLO Rd	00		000	Rd	00	12	Rd ← LO

### 3.1 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, SRA, SRAV**;
- As **instruções lógicas** são **AND, OR, XOR, NOR, ANDI, ORI, XORI, SLL, SLLV, SRL e SRLV**;
- As **instruções de movimentação de dados** são **LUI, LBU, LW, 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**.

### 3.2 Observações sobre a Semântica de Instruções no Processador MIPS\_S

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

- A arquitetura MIPS foi elaborada para privilegiar a simplicidade do conjunto de instruções sem, contudo, sacrificar sua flexibilidade. Devido à limitação de todas as instruções possuírem exatamente o mesmo tamanho, instruções em geral disponíveis em outros processadores estão ausentes no MIPS. Contudo, foi tomado cuidado no projeto desta arquitetura 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 sequência 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 obviamente não existe, mas pode ser implementada por uma sequência de duas instruções, **LUI** e **ORI**. É possível a programas montadores para o processador MIPS\_S disponibilizar uma pseudo-instrução **LI** com um operando imediato de 32 bits, gerando como código para esta uma sequência **LUI + ORI**. Por exemplo, existe no montador/simulador MARS uma pseudo-instrução (**LA**, do inglês *load address*) que usa **LUI + 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 sinais qualificadores de estado específicos na arquitetura MIPS\_S. Elas escrevem no registrador destino da instrução ou 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 neste registrador de destino contra a constante 0 (por exemplo, usando as instruções **BEQ, BGEZ, BNE** ou **BLEZ**, tendo como um dos operandos a constante 0, disponível no registrador **\$0** ou **\$zero**). Tais sequências de instrução permitem implementar todas as comparações de igualdade ou magnitude (“menor”, “maior”, “menor ou igual” e “maior ou igual”).

**Exercício:** Proponha uma sequência de instruções do processador MIPS\_S para implementar as seguintes pseudo-instruções: **BGT, BGE, BLT** e **BLE** (respectivamente, **salta se maior, se maior ou igual, se menor** e **se menor ou igual relativo**). Note-se ainda que as variações das instruções **SLTxx** 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**).

## 4 BANCO DE REGISTRADORES DO PROCESSADOR MIPS\_S

A partir desta Seção do presente documento, apresenta-se uma possível organização para o processador MIPS\_S. Algumas das características mencionadas aqui provêm da arquitetura do MIPS. Isto será explicitado no texto a seguir, quando for o caso. O leitor pode desde já referir-se ao diagrama de blocos da organização proposta, apresentado na Figura 3 da Seção 5, de forma a acompanhar a leitura a seguir com uma ilustração adequada das estruturas mencionadas e de sua interconexão a demais estruturas presentes no hardware.

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

- Um banco de registradores contendo 32 registradores de uso geral, cada um de 32 bits, denominados **\$0** a **\$31**. **Esta é uma característica definida na arquitetura MIPS**. Existe uma denominação textual alternativa para cada um dos registradores. Ela está apresentada na Tabela 2 abaixo, retirada do Apêndice A do livro Organização e Projeto de Computadores de D. A. Patterson e J. L. Hennessy. Este Apêndice é de domínio

público e pode ser facilmente encontrado na Internet<sup>4</sup>. O banco de registradores proposto aqui tem uma porta de escrita e duas portas 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 R1 e o conteúdo de outro registrador (ou do mesmo registrador) no barramento de saída R2;

- Registradores **R1**, **R2** e **R3**: São registradores que irão conter os operandos da maioria das instruções executadas;
- **RALU**: um registrador de 32 bits usado para armazenar o resultado produzido pela ALU;
- **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 deriva do fato de uma multiplicação de dois números de 32 bits gerar um valor de 64 bits, bem como do 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**;
- **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. Ele também possui 32 bits;

**Tabela 2 – Denominações dos registradores da arquitetura MIPS\_S. 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 MARS aceita a denominação da primeira coluna da tabela, ou da segunda coluna, ambas precedidas do caractere especial \$.**

Nome	Número (hexa/binário)	Nome Alternativo	Significado ou Convenção de Utilização
\$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/subrotina)
\$17	11 / 10001	\$s1	temporário (salvo nas chamadas de função/subrotina)
\$18	12 / 10010	\$s2	temporário (salvo nas chamadas de função/subrotina)
\$19	13 / 10011	\$s3	temporário (salvo nas chamadas de função/subrotina)
\$20	14 / 10100	\$s4	temporário (salvo nas chamadas de função/subrotina)
\$21	15 / 10101	\$s5	temporário (salvo nas chamadas de função/subrotina)
\$22	16 / 10110	\$s6	temporário (salvo nas chamadas de função/subrotina)
\$23	17 / 10111	\$s7	temporário (salvo nas chamadas de função/subrotina)
\$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	armazena 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 no **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 **R1** e **R2**, e o valor obtido pela execução de uma dada operação lógico-aritmética é armazenado no registrador **RALU**. Apresentam-se detalhes nas Seções a seguir.

<sup>4</sup> Nas páginas de Internet da disciplina existe uma versão do Apêndice A, anotada e corrigida em vários pontos onde o texto original apresenta incorreções ou omissões.



Os sinais de controle dos multiplexadores **M2** a **M11** não são gerados no BC. Eles são derivados no BD diretamente da instrução corrente. Para uma ideia mais completa da implementação do BD do processador MIPS\_S, resta apresentar as organizações internas do banco de registradores, da unidade lógica e aritmética, das operações de multiplicação e divisão e o Bloco de Controle (BC) do MIPS\_S. As três Seções a seguir discutem as organizações dos três primeiros blocos citados, enquanto a Seção 7 discute o BC.

### 5.1 Banco de Registradores

A Figura 4 ilustra a organização do banco de registradores, sob a forma de um diagrama de blocos. A ALU será discutida em parte na Seção 5.2. 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 questão. As portas de leitura consistem em multiplexadores (32X32):(1X32), controladas pelos endereços de leitura.

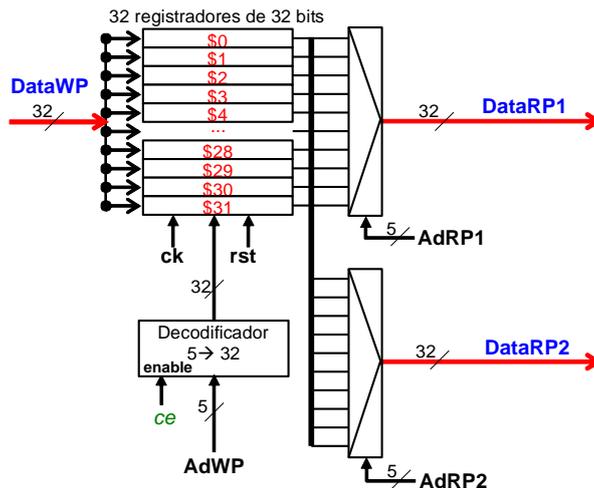


Figura 4 - Diagrama em blocos do banco de registradores de uso geral da organização MIPS\_S.

### 5.2 Resumo das Operações Executadas pela ALU

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

Tabela 3 – Operações da ALU no processador MIPS\_S, 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	Nenhuma operação

A operação da ALU é relevante para quase todas as instruções, embora algumas destas apenas a utilizem para transportar um valor da entrada para a saída (como **JR** e **JALR**) e outras (como **MULTU** e **DIVU**) sequer utilizam a ALU. O resultado da operação na ALU é sempre armazenado no registrador **RALU**. A Tabela 3 define quais devem ser as operações da ALU, para cada instrução.

### 5.3 Organização para as instruções MULTU e DIVU

As instruções **MULTU** e **DIVU**, que realizam a **multiplicação e a 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** são implementadas de maneira estrutural, usando texto VHDL sintetizável. Para tanto, escolheu-se usar um algoritmo de implementação de multiplicador e divisor em hardware, tal como descrito nas transparências 10 a 17 da apresentação disponível no link mostrado abaixo.

<http://www.inf.pucrs.br/~calazans/undergrad/arg1/aulas/aritcomp.pdf>.

O algoritmo de multiplicação é 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 Organização e Projeto de Computadores de D. A. Patterson e J. L. Hennessy, na Seção 4.6 da segunda edição deste livro.

## 6 CICLOS PARA EXECUTAR INSTRUÇÕES DA ORGANIZAÇÃO MIPS\_S

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

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

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 <sup>5</sup>
XORI	4	DIVU	67 <sup>6</sup>
LUI	4	MFHI	4
LBU	5	MFLO	4
LW	5		

## 7 O Bloco de Controle (BC) do processador MIPS\_S

Para comandar a execução de instruções neste processador, o BC do processador MIPS\_S possui um conjunto de quatro estruturas de hardware. São estas:

- **Um registrador** que guarda o código da instrução atualmente em execução (IR);
- Estruturas para controlar a posição de memória onde reside a instrução atual e a próxima a ser executada, compostas por **dois registradores (PC e NPC)** e **um incrementador**;
- **A decodificação de instruções**, uma tabela que para cada código de 32 bits contido no IR identifica a instrução que este representa, ou o fato de o código não se referir a nenhuma instrução válida (valor *invalid instruction*);
- **Uma máquina de estados de controle**, que gera uma sequência de sinais usados em cada ciclo de execução de uma instrução;

<sup>5</sup> Valor típico, depende do algoritmo escolhido.

<sup>6</sup> Valor típico, depende do algoritmo escolhido.

A Figura 5 ilustra a estrutura da máquina de estados de controle da MIPS\_S, 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.

- **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.

Também se mostra na Figura 5 quais registradores são alterados, através da indicação de qual/quais sinal/sinais de controle de habilitação de escrita é/são ativado/ativados em cada estado. Não se mostram as ativações dos sinais de controle dos multiplexadores, ou as operações da ALU. A função dos 7 estados mostrados na Figura são as seguintes:

- **Sfetch**: primeiro ciclo, busca de instrução;
- **SReg**: segundo ciclo, leitura dos registradores fonte;
- **Salu**: terceiro ciclo, operação com a ALU e eventual uso do comparador, do hardware de multiplicação ou do hardware de divisão;
- **Swbk**: 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, atualiza-se o valor do **PC**. Condicionalmente se escreve no banco de registradores.

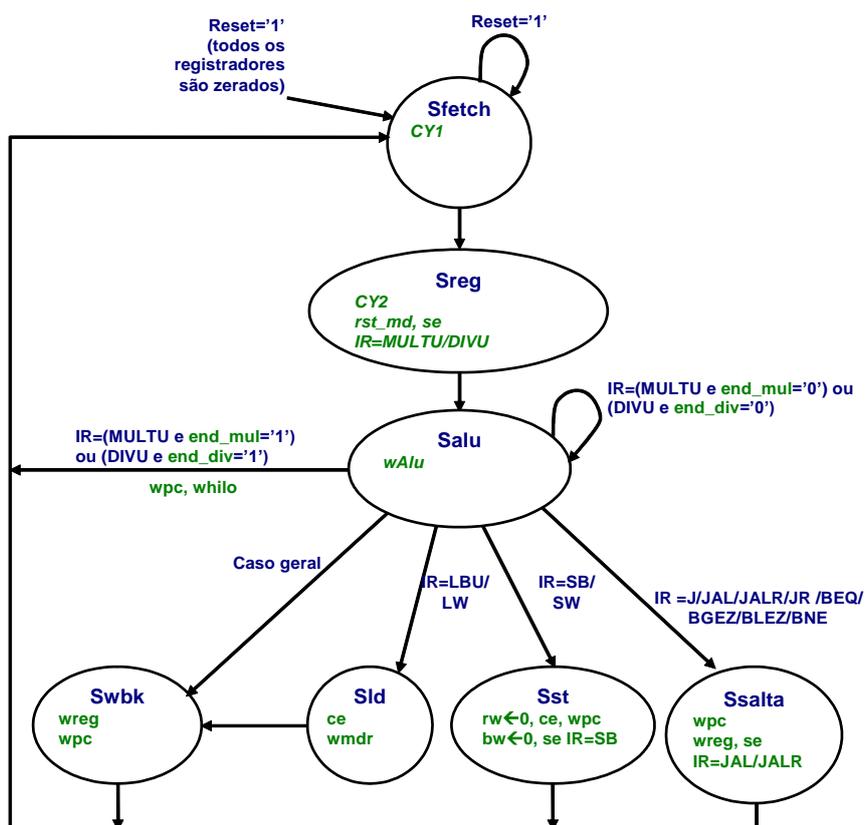


Figura 5 - Máquina de estados de controle para a organização MIPS\_S.

## 8 PROGRAMA PARA TESTAR TODAS AS INSTRUÇÕES DA MIPS\_S

A Figura 6 mostra a interface gráfica do simulador MARS, composta de 5 subjanelas (4 subjanelas estão ocultas, aquela da aba Edit, que mostra o editor de textos do ambiente MARS, duas outras para uso de coprocessadores, Coproc 1 e Coproc 2 e uma para operações de entrada e saída, RUN I/O):

- a subjanela **Text Segment** mostra as instruções do programa ora em simulação, em três formatos: código-objeto, código intermediário e código fonte;
- a subjanela **Data Segment** mostra os dados estáticos do programa ora em simulação (várias outras áreas de memória estão disponíveis para escolha, tais como a área de pilha apontada pelo registrador \$sp, o heap etc.);
- a subjanela **Labels** mostra dados sobre os rótulos do programa, tanto da área de instruções como da área de dados;
- a subjanela **Registers** mostra os conteúdos dos registradores da arquitetura (os 32 registradores do banco, o PC, o Hi e o Lo);
- a subjanela **Mars 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 descrito em linguagem de montagem (tipicamente com nome <file>.asm). É possível gerar um arquivo de saída com o conteúdo das janelas de instruções e dados do programa, incluindo o código objeto gerado. Este mais tarde é usado, após algum tratamento manual, como entrada para a simulação do hardware do processador.

O código objeto pode ser gerado após a leitura e montagem de um arquivo contendo um programa corretamente descrito. Para tanto, basta salvar como em arquivos texto as áreas de instruções e dados como é visto em aula.

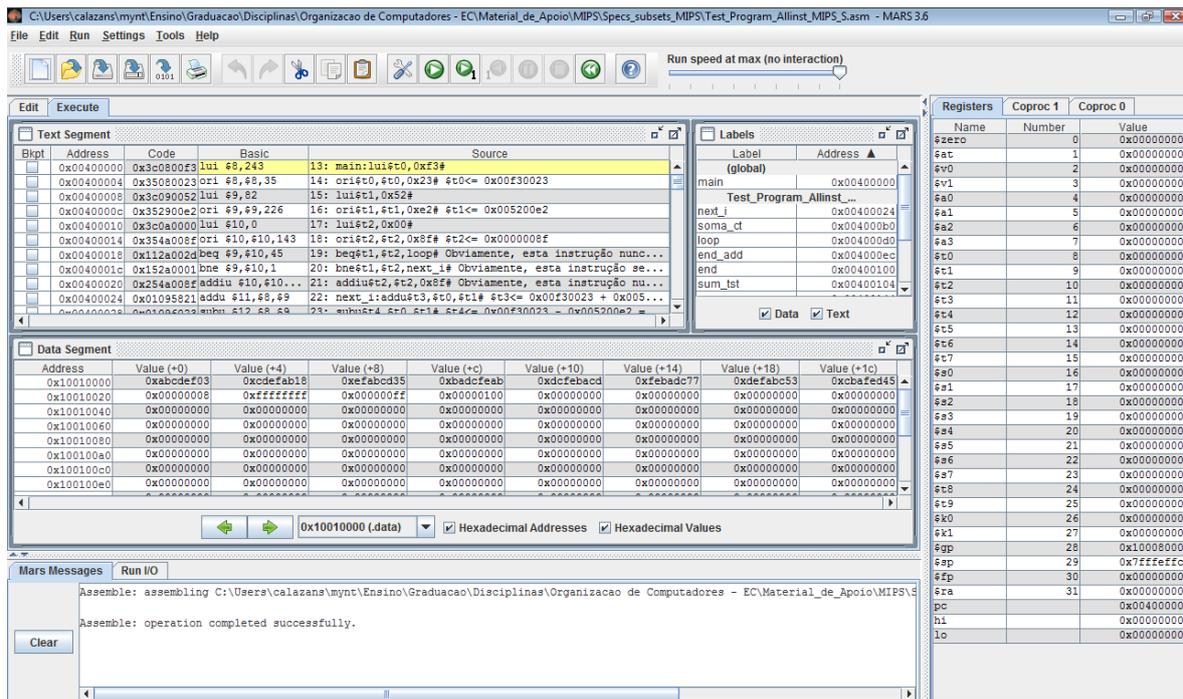


Figura 6 - Interface gráfica do montador/simulador MARS.

Recomenda-se **escrever os programas em linguagem de montagem (assembly)**, gerando-se o código objeto automaticamente, a partir do montador/simulador MARS. A ferramenta de simulação, assim como documentação de como utilizar a ferramenta encontram-se apontadas no material de apoio da disciplina, são encontradas facilmente na Internet.

A Figura 7 apresenta um programa escrito em linguagem de montagem para o processador MIPS\_S, contendo pelo menos uma instância de cada uma das instruções 37 instruções deste processador. Recomenda-se o uso do simulador MARS para executar este programa e verificar sua funcionalidade, cujos efeitos são descritos linha a linha no programa, no campo de comentários.

O simulador MARS pode gerar arquivos de saída contendo o código objeto das áreas de dados e de instruções de qualquer programa, entre outras informações. Tais arquivos podem ser salvos usando a opção de

menu File → Dump Memory do simulador. Ele será útil para gerar a entrada do processo de simulação da implementação de hardware a ser realizada neste trabalho.

```
#####
## ARQUITETURA MIPS S - teste de todas as instruções
## Autor: Ney Calazans (ney.calazans@puccrs.br)
## Este teste é exaustivo quanto ao número total de instruções
## executadas, não quanto à cobertura dos casos de teste
## possíveis
##
## Para simular este programa no MIPS_S,
## execute-o por 13,76 microssegundos (assumindo um clock de 50Mhz)
##
## Última alteração: 04/06/2021
#####
.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 = 0x00a0fff1
    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 = 0xff0cfff1c
    multu $t0,$t1 # Hi & Lo <= 0x00f30023 * 0x005200e2 = 0x00004dd6elb1c1ee6
    mfltu $a0 # $a0<= 0x00004dd6
    mfl0 $a1 # $a1<= 0x0elb1c1ee6
    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<= 0xffff9000
    srav $t0,$t0,$s2 # $t0<= 0xffff9000
    srlv $t0,$t0,$s2 # $t0<= 0x0ffff90
    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 (o resto não muda)
    addiu $t0,$zero,0x1 # CUIDADO, mudou elemento do array a ser processado por soma_ct
    subu $t0,$zero,$t0 # $t0<= 0x00000001
    bgez $t0,loop # $t0<= 0xfffffff
    slt $t3,$t0,$t1 # Esta instrução nunca deve saltar, pois $t0 = -1
    sltu $t3,$t0,$t1 # $t3<= 0x00000001, pois -1 < 10
    slti $t3,$t0,0x1 # $t3<= 0x00000000, pois (2^32)-1 > 10
    sltiu $t3,$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. Lembre, 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. 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 # SUBROTINA 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 0xabcdef03 0xcdefab18 0xefabcd35 0xbadcfeab 0xdcfebacd 0xfedbacd7 0xdefabc53 0xcbafead5
# o terceiro byte da segunda palavra (0xef) vira 0x10 antes da execução de soma_ct
size: .word 0x8
const: .word 0xfffffff # constante -1 em complemento de 2
var_a: .word 0xff # para trecho de teste de chamadas de subrotinas
var_b: .word 0x100 #
```

Figura 7 - Programa exemplo para teste da organização MIPS\_S.