SIMULAÇÃO VHDL DO PROCESSADOR MRStd

Este tutorial visa iniciar os alunos à simulação de um processador que dá suporte a um subconjunto amplo de instruções da arquitetura MIPS, denominado *MRStd*. Este processador implementa uma organização que pode executar uma boa parte das instruções da arquitetura MIPS R2000. As principais instruções às quais esta arquitetura não dá suporte são todas as instruções de manipulação de números de ponto flutuante, as instruções de acesso à memória a meias palavras e as de acesso desalinhado à memória. Contudo, o acesso desalinhado a byte com instruções *lb*, *lbu* e *sb* é possível.

Como o que se pretende é mais tarde prototipar o processador nos FPGAs de plataformas disponíveis em laboratório, a estrutura do processador foi acrescida de descrições de memórias de instruções e dados que podem ser sintetizadas facilmente em FPGAs Xilinx, conforme explicado abaixo. Ao contrário dos modelos de simulação de processadores vistos nas aulas teóricas, onde um *testbench* complexo permite carregar antes da simulação as memórias com um programa qualquer e seus dados, o que se mostra aqui é o processo de sintetizar um sistema processador-memórias pronto para executar um programa específico sobre dados específicos. Uma desvantagem desta abordagem é que para cada novo conjunto programa/dados a executar, o sistema deve ser totalmente re-sintetizado usando o ambiente ISE.

Neste tutorial apresenta-se apenas a estrutura geral da descrição simulável e sintetizável e procede-se a alguns exercícios de simulação para levar os alunos a dominar a estrutura do sistema processador-memórias MRStd. A prototipação é apresentada posteriormente.

Arquivos necessários para este tutorial: (1) código fonte do programa gerador da inicialização da memória: *le_mars.c*; (2) descrição do processador: *mrstd.vhd*; (3) *testbench* do processador: *mrstd_tb.vhd*; (4) aplicação exemplo: *soma_vet.asm*.

1 AMBIENTE DE SIMULAÇÃO DO PROCESSADOR MRSTD

O ambiente de simulação/síntese a ser usado neste tutorial encapsula o processador MRStd e as duas memórias (de instruções e dados) necessárias à execução de programas por este processador. A organização do ambiente é ilustrada na Figura 1. As únicas portas da interface externa, além do **clock** e do **reset**, são sinais de controle que permitem a uma entidade externa ler o conteúdo da memória de dados.

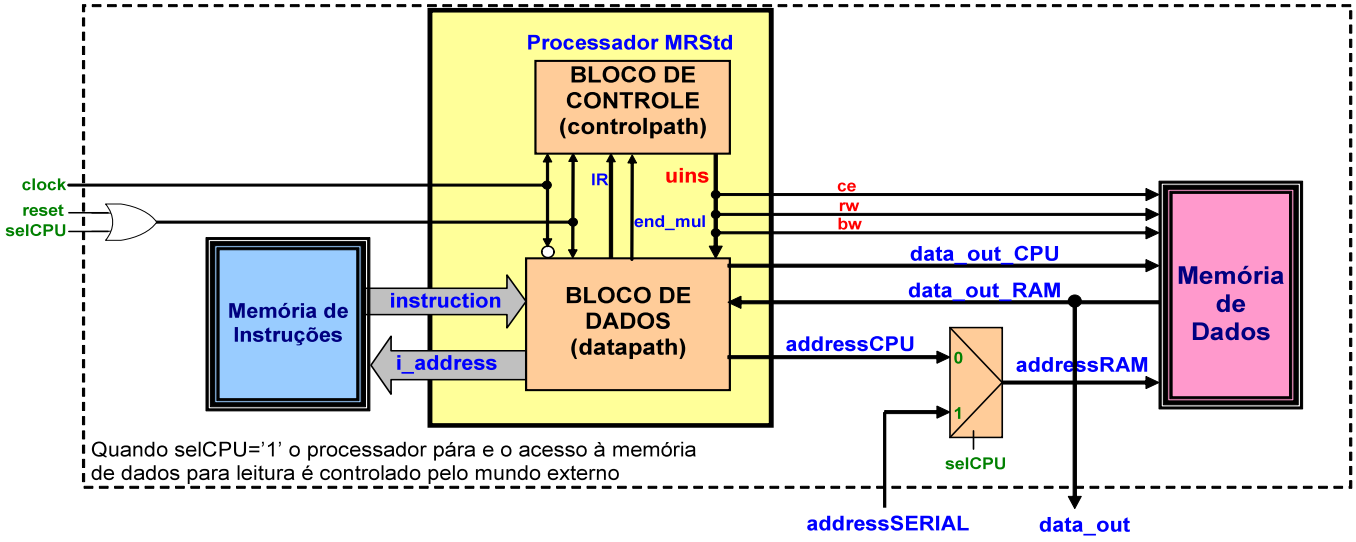


Figura 1 – Diagrama de blocos do ambiente contendo o processador MRStd e suas memórias de instruções e dados. O processador MRStd em si (retângulo amarelo e seus conteúdos) conecta-se às memórias de instruções e de dados. Vários sinais de interfaces dos blocos foram omitidos na Figura, para beneficiar a clareza do diagrama.

A Figura 2 mostra uma visão parcial do código VHDL do ambiente. Basicamente, este arquivo contém as instâncias dos 3 blocos principais do ambiente: memória de instruções, memória de dados, e o processador MRStd, mais uma "lógica de cola" simples para permitir o controle da comunicação entre as diversas partes do ambiente entre si.

```
entity MRStd_with_memories is
    port
    (   clock, reset, selCPU: in std_logic;
        addressSERIAL: in std_logic_vector(31 downto 0);
        data_out: out std_logic_vector(31 downto 0));
end MRStd_with_memories;

architecture a1 of MRStd_with_memories is
--- vários sinais são declarados aqui (não mostrados, por fins de concisão)
begin

    MRstd_inst: entity work.MRstd
       port map ( clock=>clock, reset=>rst_cpu, ce=>ce, rw=>rw, bw=>bw,
              i_address=>i_address, d_address=>addressCPU,
              instruction=>instruction, data_in=>data_out_RAM, data_out=>data_out_CPU
              );

     int_address <=  i_address(10 downto 2);
     m1: entity work.program_memory
        port map( clock=>clock, address=>int_address, instruction=>instruction);

     m2: entity work.data_memory
          port map (clock=>clock, ce=>ce, we=>rw, bw=>bw,
                    address=>addressRAM(12 downto 0), data_in=>data_out_CPU,
                                       data_out=>data_out_RAM);

    addressRAM <=   addressCPU  when selCPU='0' else  addressSERIAL;

    rst_cpu <= reset or selCPU;

    data_out <= data_out_RAM;end a1;

end MRStd_with_memories;
```

Figura 2 – Descrição parcial do código VHDL que descreve o par entidade/arquitetura ilustrado na Figura 1.

1.1 Como as memórias do ambiente são organizadas?

As memórias empregadas neste ambiente devem ser sintetizáveis. Uma maneira eficiente de fazer isto é utilizar estruturas especiais existentes em FPGAs para implementar memórias. Os FPGAs da família Spartan3 da Xilinx possuem certa quantidade de blocos de memória de 16 Kbits configuráveis [1], denominados de BlockRAMs. BlockRAMs foram usadas neste ambiente para implementar estruturas compatíveis com os blocos de memória aos quais o MRStd faz acesso.

Descreve-se a seguir o mapeamento das memórias de instruções e dados para BlockRAMs em FPGAs da família Spartan3 da Xilinx.

1.1.1 Memória de instruções

O acesso à memória de instruções do MIPS é sempre realizado buscando palavras de 32 bits alinhadas em endereços múltiplos de 4, conforme visto em aula teórica. Assim, ao invés de se utilizar uma organização a byte, pode-se usar uma memória organizada a palavras de 32 bits. No caso do ambiente optou-se então por utilizar uma única BlockRAM para instruções, configurando seus 16Kbits como uma memória de 512 palavras de 32 bits. Isto limita os programas do processador a não terem mais de 512 instruções, o que é mais que suficiente para os fins didáticos a que se destina o ambiente.

A configuração desta memória deve usar uma codificação específica em VHDL, baseada em blocos pré-definidos de um par biblioteca/pacote fornecidos pela Xilinx (biblioteca `UNISIM`, pacote

[1] Na realidade, BRAMs são blocos de 18Kbits, mas para cada byte (8 bits) desta memória existe 1 bit de paridade, para fins de detecção de erros de leitura/escrita. Quando se ignora os bits de paridade (o que é feito aqui), tem-se uma memória de 16Kbits.

vcomponents). O componente adequado para usar aqui é denominado RAMB16_S36. Para usar tal componente, basta declarar a biblioteca e o pacote no código VHDL que usa a memória e instanciar esta. Um exemplo é mostrado na Figura 3.

```
programa: RAMB16_S36 generic map
(
    INIT_00 => X"8e520000343200083c0110018e310000343100043c011001343d08003c011000",
    INIT_01 => X"afb3000cafb20008afb10004afbf000027bdfff08e7300003433000c3c011001",
    INIT_02 => X"342800003c01100127bd00108fb400048fbf00000100f809342800883c010040",
    INIT_03 => X"afa90008afa80004afbf000027bdfff48fa900088fa8000403e00008ad140000",
    INIT_04 => X"27bdfff48fa9000c27bd000c8fa800048fbf00000100f809342800ec3c010040",
                                    ...
    INIT_3D => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_3E => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_3F => X"0000000000000000000000000000000000000000000000000000000000000000"
)

port map
(
    CLK  => clock,
    ADDR => address,
    EN   => '1',
    WE   => '0',
    DI   => x"00000000",
    DIP  => x"0",
    DO   => instruction,
    SSR  => '0'
);
```

Figura 3 – Exemplo de estrutura de inicialização e instanciação de memórias específicas para FPGAs Xilinx. No caso, apresenta-se a estrutura da memória de instruções.

A inicialização da memória de instruções com o programa a executar é feita usando parâmetros INIT, declarados via comando VHDL **`generic map`**, conforme a Figura 3. O preenchimento em tempo de inicialização destes blocos pressupõe que cada linha **`INIT_xx`** desta codificação descreve como preencher 8 palavras consecutivas de 32 bits (4 bytes) de memória. Ou seja, cada linha especifica o conteúdo de 256 bits (32 bytes ou 8 palavras de 32 bits) da memória, perfazendo um total de 64 linhas (em hexa, 0x3F) por BlockRAM (de 16.384 bits, 64 linhas×256 bits).

O comando **`port map`** da Figura 3 conecta pinos da memória (pré-definidos pela organização das BlockRAMS nos FPGAs) a sinais do processador MRStd. Note que no exemplo da Figura 3, o sinal **`WE`** (habilitação de escrita ou *write-enable*) é igual a '0', caracterizando esta memória como apenas de leitura (ROM).

1.1.2 Memória de dados

A memória de dados tem estrutura distinta da memória de instruções, pois deve possibilitar a escrita em bytes isolados, como necessário ao executar a instrução **`SB`**, por exemplo. Para tanto, utiliza-se 4 BlockRAMs de memória com acesso a byte e possibilidade de acesso paralelo a um byte de cada bloco em um dado instante. Cada bloco é configurado como uma memória de 2.048 palavras de 8 bits. Assim, existe no total uma memória de dados de 2.048 palavras de 32 bits (2.048 × 32 = 65.536 bits) ou 64Kbits ou 8Kbytes, ou 2Kpalavras de 32 bits. O componente adequado da biblioteca UNISIM, no pacote vcomponents a usar aqui é denominado RAMB16_S9.

Exemplo passo a passo para gerar o arquivo de inicialização das memórias

1. Abra o MARS, carregue o programa exemplo (*soma_vet.asm*), realize a montagem do código objeto (F3), e gere os *dumps* de memória (^D). Note que o programa assembly termina com uma instrução de salto para ela mesma, uma forma de virtualmente "travar" a simulação após o término da execução das tarefas úteis do programa. Use a opção de menu ***File → Dump Memory***, escolhendo fazer *dump* dos segmentos de texto e de dados em dois arquivos distintos, com os nomes ***mars_i*** e ***mars_d***, respectivamente, no formato indicado na Figura 4.

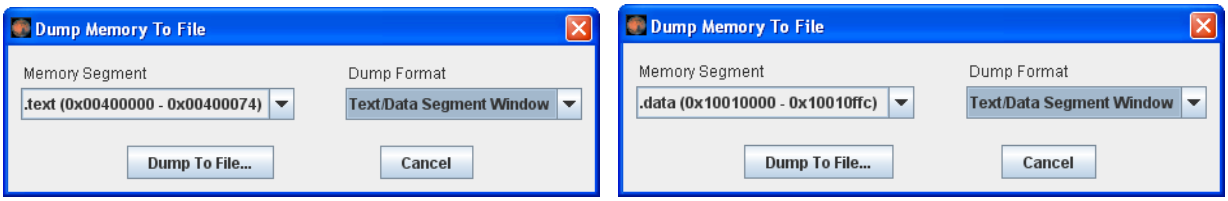


Figura 4 – *Dump* das áreas de memória correspondendo às instruções e aos dados.

O exemplo da Figura 5 corresponde a trechos dos dumps de memória.

```
Address    Code        Basic
0x00400000  0x3c011001  lui $1,0x1001
0x00400004  0x3428002c  ori $8,$1,0x002c
0x00400008  0x3c011001  lui $1,0x1001
0x0040000c  0x34290058  ori $9,$1,0x0058
0x00400010  0x3c011001  lui $1,0x1001
0x00400014  0x342a0000  ori $10,$1,0x0000
0x00400018  0x3c011001  lui $1,0x1001
0x0040001c  0x342b0084  ori $11,$1,0x0084
0x00400020  0x8d6b0000  lw $11,0x0000($11)
0x00400024  0x19600009  blez $11,0x0009
0x00400028  0x8d0c0000  lw $12,0x0000($8)
..
```

Área de texto (instruções)
arquivo *mars_i*

```
0x10010000  0x11111111 0x22222222 0x32333333 0x44444444 0x55555555 0x66666666 0x77777777 0x88888888
0x10010020  0x99999999 0xaaaaaaaa 0xbbbbbbbb 0x00111111 0x00222222 0x00333333 0x00444444 0x00555555
0x10010040  0x00666666 0x00777777 0x00888888 0x00999999 0x00aaaaaa 0x00bbbbbb 0x11000000 0x22000000
0x10010060  0x32000000 0x44000000 0x55000000 0x66000000 0x77000000 0x88000000 0x99000000 0xaa000000
0x10010080  0xbb000000 0x0000000b 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100a0  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100c0  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100e0  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010100  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010120  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010140  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010160  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010180  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100101a0  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
```

Área de dados
arquivo *mars_d*

Figura 5 – Exemplo dos trechos dos arquivos de *dump* usados para gerar o arquivo de inicialização das memórias do processador MRStd.

2. Compile o programa **le_mars.c** e gere um executável **le_mars.exe** (este procedimento será efetuado uma única vez). Este é um programa que lê os dois arquivos *dump* e gera um arquivo VHDL com a instanciação das memórias.

3. Execute o *le_mars.exe*.

```
$ ./Le_mars

This program reads two memory dump files (mars_i and mars_d) and generates the memory.vhd file

..Reading instruction file
..Reading data file
..Generating output file
```

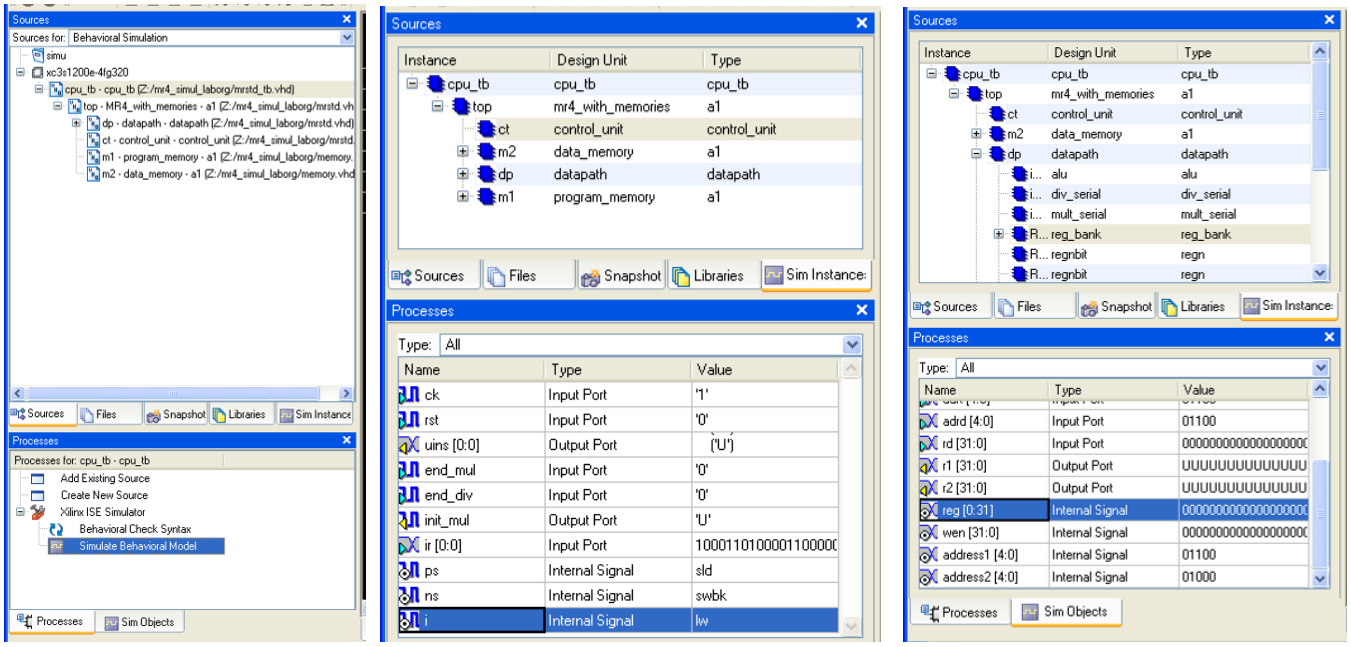
Abra o arquivo memory.vhd e analise o código gerado. Neste código VHDL há 4 blocos de memória inicializados, 1 para a memória de instruções e 3 para a memória de dados.

Simulação do processador mr4

1. Criar um projeto no ISE Simulator adicionando os arquivos:
   - **memory.vhd** (gerado no passo anterior)
   - **mrstd.vhd** (presente na distribuição deste laboratório)
   - **mrstd_tb.vhd** (presente na distribuição deste laboratório)

2. Duplo-clique em *simulate behavioral model* (Figura 6(a)). Vai haver uma certa demora no passo:
   ```
   Restoring VHDL parse-tree unisim.vpkg from c:/xilinx/10.1/ise/vhdl/hdp/nt/unisim/unisim.vdbl
   ```

3. Navegar na hierarquia do projeto e exibir na janela com as formas de onda a instrução corrente (Figura 6(b)) e o banco de registradores (Figura 6(c)). Para exibir um dado sinal basta selecionar o mesmo e arrastá-lo para a janela com as formas de onda.



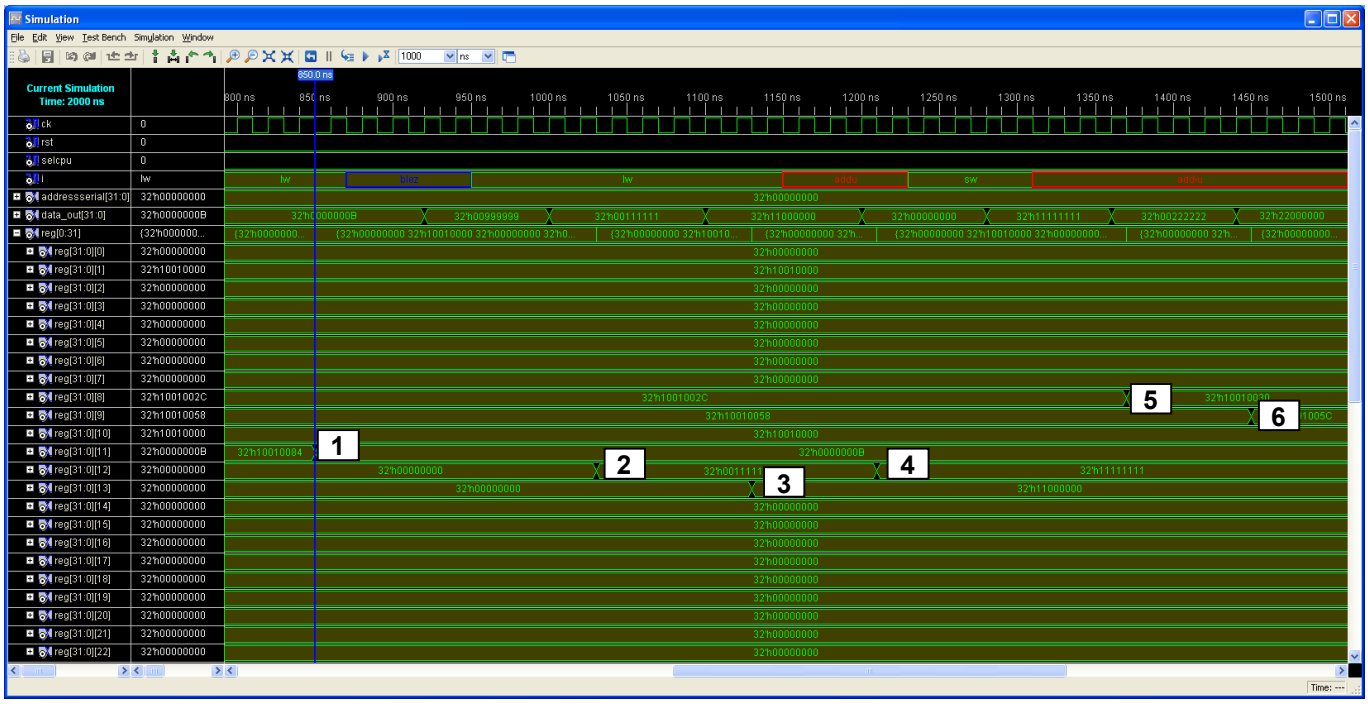
(a) (b) (c)

Figura 6 – (a) Inicialização da simulação; (b) seleção da instrução corrente; (c) seleção do banco de registradores.

4. Reinicializar a simulação (menu *Simulation* → *restart*), e simule por 2 us. Dica: pode-se escrever *restart* e *run 2 us* na janela com mensagens.

5. Fazer um zoom entre 800ns e 1500ns (podes-se usar o botão do meio do mouse). Os eventos marcados na simulação têm a seguinte correspondência com o código *assembly* (acompanhar a instrução tanto no código *assembly* quanto na simulação):

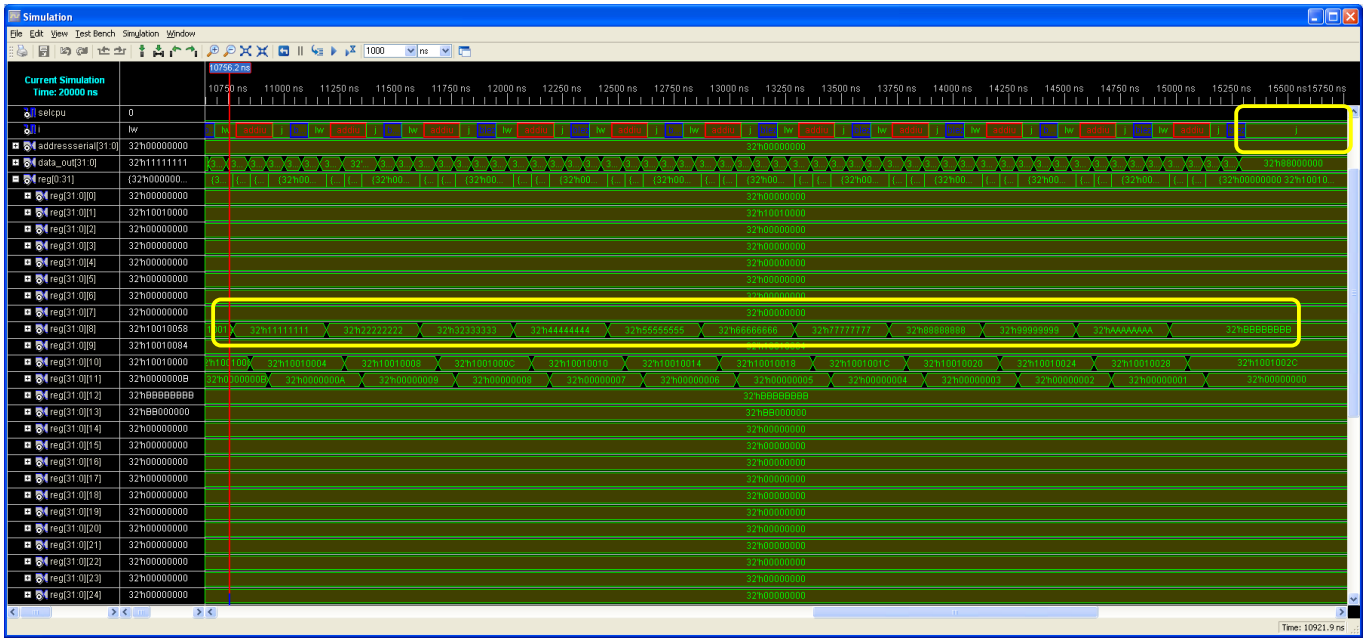
```
lw      $t3,0($t3)      # EVENTO 1: carregou B no reg11 ($t3)
blez    $t3,le_m
lw      $t4,0($t0)      # EVENTO 2: carregou 00111111 no reg12 ($t4)
lw      $t5,0($t1)      # EVENTO 3: carregou 11000000 no reg13 ($t5)
addu    $t4,$t4,$t5     # EVENTO 4: fez a soma dos valores e gravou em $t4
sw      $t4,0($t2)
addiu   $t0,$t0,4       # EVENTO 5: somou 4 no reg8 ($t0)
addiu   $t1,$t1,4       # EVENTO 6: somou 4 no reg9 ($t1)
```



6. Simular mais 18 us (run 18 us). Fazer um zoom entre 10us e 15 us.

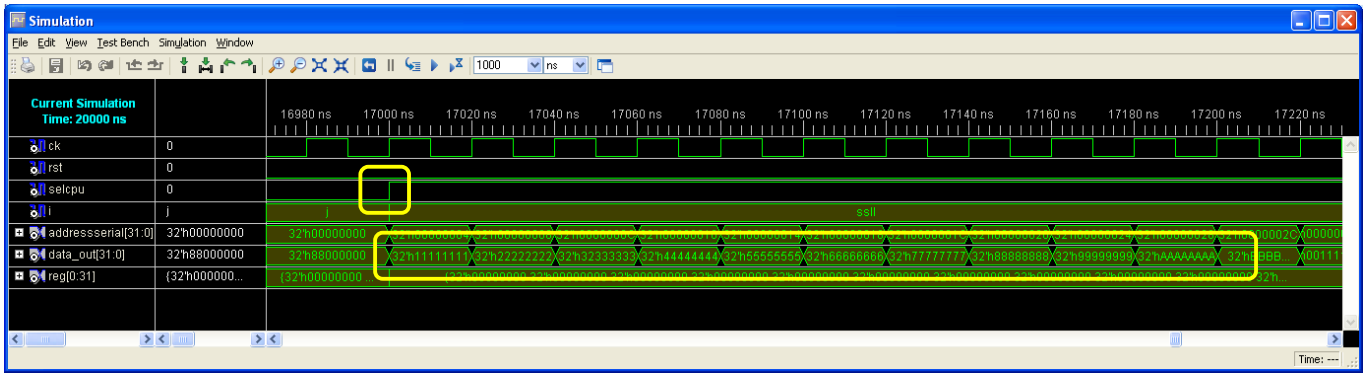
Observar que a partir de 10 us o registrador $t0 (reg8) começa ler os dados resultantes da soma. Esta é uma forma de **verificar** se o programa funcionou corretamente – lendo os dados escritos nas posições de memória onde se armazenou o resultado do processamento. O laço de verificação é:

```
loop2:  blez    $t3,end
        lw      $t0,0($t2)
        addiu   $t2,$t2,4
        addiu   $t3,$t3,-1
j       loop2           # ao final da simulação o programa fica em loop (instrução j)
```



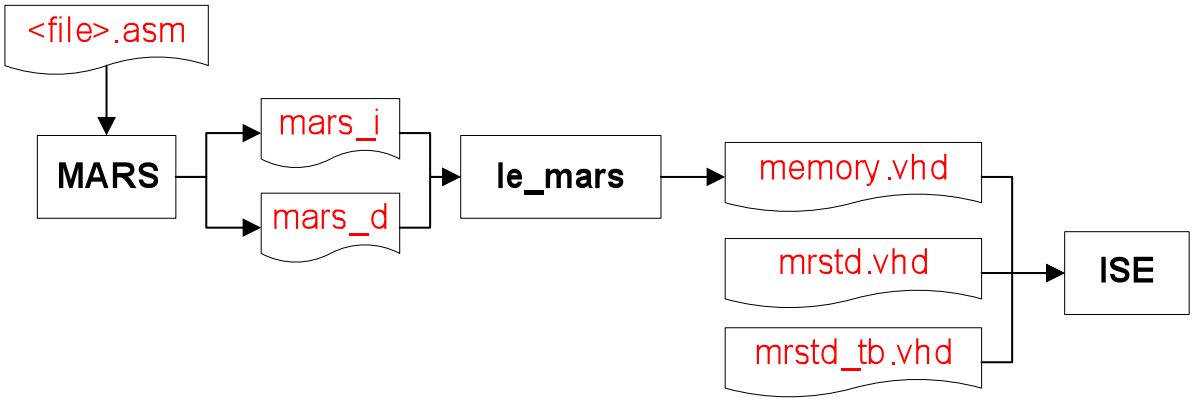
7. Fazer um zoom full (F6), e depois fazer um zoom entre 17us e 17,2 us. Em 17 us o sinal *setcpu* sobe (tempo definido no testbench) e o conteúdo da memória para a ser lido em *data_out*. Esta é **outra forma de se verificar se a simulação** está correta – só cuidar com o tempo em que se deve

subir o *setcpu* – deve-se simular uma vez com este valor em zero no *testbench*, anotar o tempo de quando termina a simulação, e depois setar no *testbench* este tempo em *setcpu*.



2 Fluxo de Verificação - resumo

A figura abaixa ilustra o fluxo de verificação com as ferramentas e arquivos envolvidos.



3 A Fazer e a Entregar

1. Cada grupo de alunos deverá realizar um exercício de programação em linguagem de montagem do MIPS, usando o subconjunto de instruções ao qual o MRStd dá suporte. Durante a aula, o professor atribuirá a cada grupo o exercício. Validar inicialmente no MARS.
2. Após validar no MARS o grupo deve simular o exercício no ISE – usar o fluxo apresentado anteriormente.
3. Capturar formas de onda que demonstrem o correto funcionamento do circuito.
4. O grupo **deve entregar**, para cada aplicação implementada:
    a. código fonte em linguagem de montagem do exercício.
    b. arquivo memory.vhd.
    c. documentação, com o código documentado através de um código equivalente em linguagem C ou um pseudo-código de C. Incluir neste documento as formas de onda capturadas. Explicar as formas de onda obtidas!

CONJUNTO DE EXERCÍCIOS PROPOSTOS :

1. Escrever um programa para mover um vetor armazenado entre os endereços de memória com rótulos *inicio1* e *fim1* para os endereços cujos rótulos são *inicio2* e *fim2*. Assumir que as seguintes condições são verdadeiras: valor(*inicio1*) < valor(*fim1*), valor(*inicio2*) < valor(*fim2*), (*fim1* - *inicio1*) = (*fim2* - *inicio2*) = (tamanho do vetor –1).

2. Descobrir se um número **n** é múltiplo de um número **m** qualquer. Pressupor que **n** e **m** sejam apenas números positivos.

3. Faça um algoritmo que calcula a divisão de dois números (armazenados nas posições de memória com rótulos *v1* e *v2*) através de subtrações sucessivas. Ao final do algoritmo a parte inteira da divisão deve estar na posição de memória com rótulo *int* e o resto na posição de memória com rótulo *resto*.

4. Multiplicar por somas sucessivas 2 inteiros positivos, armazenando o resultado em um inteiro longo. O multiplicando e o multiplicador devem estar armazenados nas posições de memória com rótulos *n1* e *n2*, respectivamente. O resultado deverá ser armazenado nas posições de memória com rótulos *mh* (a parte mais significativa do resultado) e *ml* (a parte menos significativa do resultado). O número de somas sucessivas deve ser o menor possível.

5. Fazer um programa que gere os *n* primeiros números da sequência de Fibonnacci, e armazene a seqüência em endereços consecutivos a partir da posição de memória com rótulo *idx*.

6. Escreva um programa para criar um vetor novo, a partir de dois vetores *v1* e *v2* de mesma dimensão *n*, segundo a relação estabelecida na equação abaixo. A interpretação da equação é: cada elemento de novo, na posição *k*, receberá o somatório dos máximos dos vetores v1 e v2, entre 0 e k.

$$novo_k = \sum_{i=0}^{k} \max(v1_i, v2_i), \quad 0 \le k < n$$

7. Implemente um algoritmo simples de ordenação crescente. O vetor de origem deve estar armazenado entre as posições de memória de rótulos *ini1* e *fim1*, respectivamente, e o vetor ordenado deve estar armazenado entre as posições de memória *ini2* e *fim2*, respectivamente.

8. Escrever um algoritmo que calcule os primeiros *n* números primos e os armazene seqüencialmente, a partir da posição de memória cujo rótulo é *nprimos*.