



Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática - FACIN

LABORG

Prof. Dr. Rafael Garibotti

AULA SOBRE:

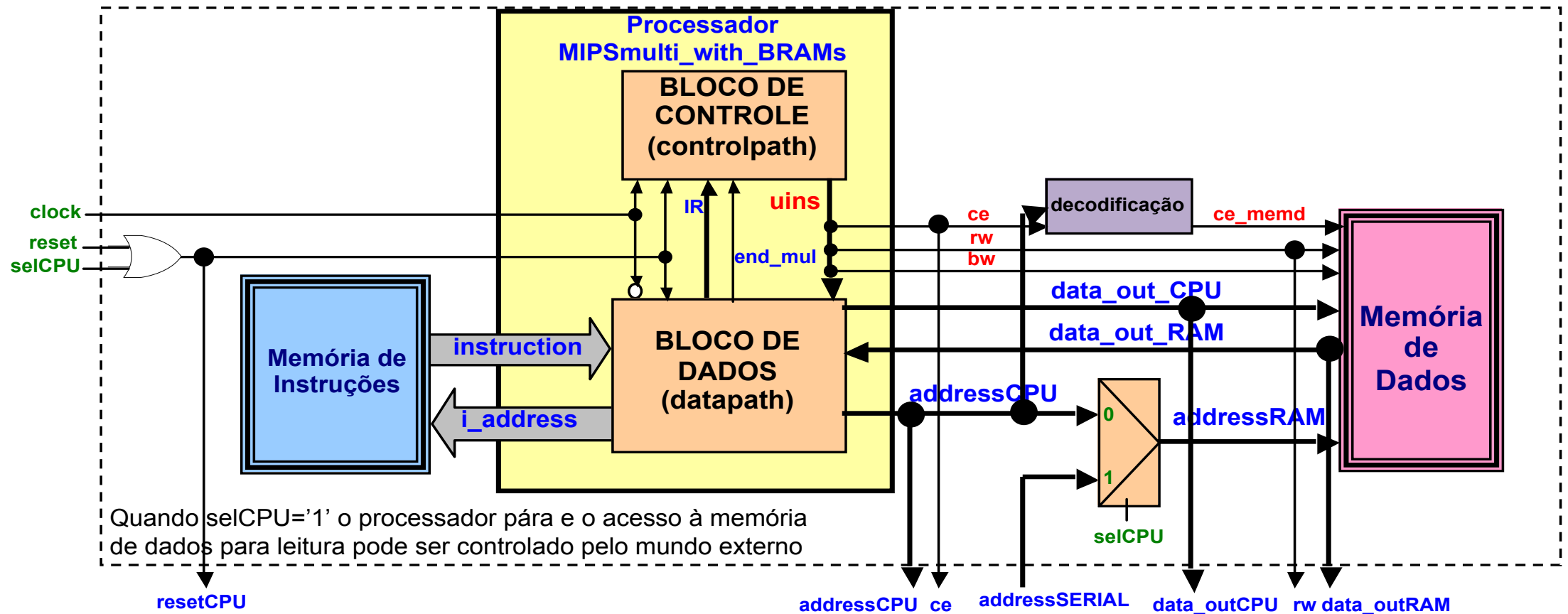
SIMULAÇÃO E PROTOTIPAGEM DE PROCESSADORES E ENTRADA E SAÍDA

INTRODUÇÃO

- Neste trabalho, deve-se simular e prototipar um processador na plataforma *Digilent*, empregando o esquema que provê saída de dados pelo processador.
- Disponibiliza-se uma implementação funcional simulável e prototipável de um sistema processador+memórias, baseado na arquitetura MR4, um subconjunto da arquitetura MIPS 2000. Esta arquitetura e uma organização para a mesma estão descritas no [documento disponível no Moodle](#).

INTRODUÇÃO

- Abaixo mostra-se o diagrama de blocos do sistema MR4+memórias. Após conseguir simular este sistema, deve-se prototipá-lo.



VHDL PARCIAL DO TOP

```
entity MIPSmulti_with_BRAMs is
  port
  (   clock, reset, selCPU: in std_logic;
      addressSERIAL: in reg32;
      ce, rw, resetCPU: out std_logic;
      addressCPU: out reg32;
      data_outRAM, data_outCPU: out reg32);
end MIPSmulti_with_BRAMs;

architecture MIPSmulti_with_BRAMs of MIPSmulti_with_BRAMs is
  --- vários sinais são declarados aqui (não mostrados, por fins de concisão)
begin
  addressCPU <= addressCPU_int;
  ce <= ce_int;
  rw <= rw_int;
  resetCPU <= rst_cpu;

  MIPS_multi: entity work.MIPSmulti
    port map( clock=>clock, reset=>rst_cpu,
              ce=>ce_int, rw=>rw_int, bw=>bw, i_address=>i_address,
              d_address=>addressCPU_int, instruction=>instruction,
              data_out=>data_out_CPU, data_in=>data_out_RAM);

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

  D_Mem: 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_int  when selCPU='0' else  addressSERIAL;

  rst_cpu <= reset or selCPU;

  data_outRAM <= data_out_RAM;
  data_outCPU <= data_out_CPU;

end MIPSmulti_with_BRAMs;
```

PROGRAMA SOMA_VET EM ASSEMBLY

```
.data
V3:      .word      0x0      0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
V1:      .word      0x1000011 0xff 0x3 0x14 0x878 0x31 0x62 0x10 0x5 0x16 0xAB000002
V2:      .word      0x2000002 0xf4 0x3 0x14 0x878 0x31 0x62 0x10 0x5 0x16 0x21000020
size:    .word      11

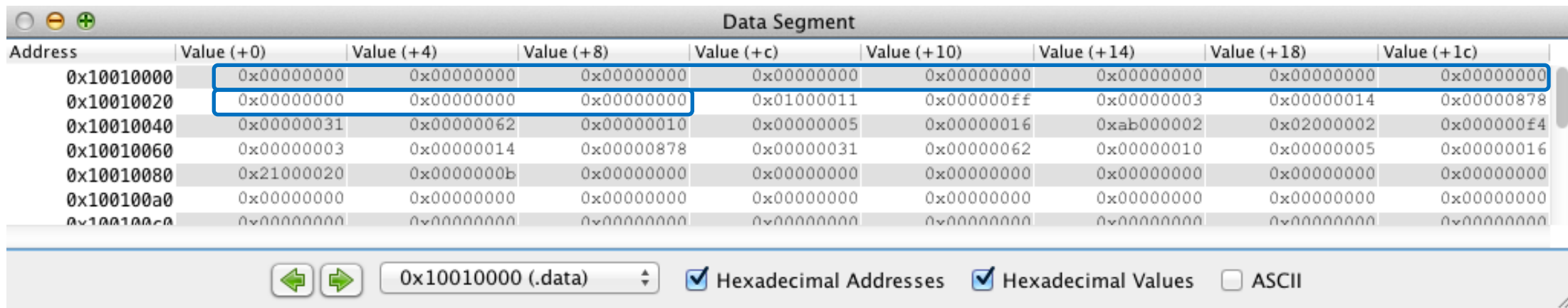
.text
.globl   main
main:
    la    $t0,V1      # generate pointer to V1 source vector
    la    $t1,V2      # generate pointer to V2 source vector
    la    $t2,V3      # generate pointer to V3 destination vector
    la    $t3,size    # get address of size
    lw    $t3,0($t3)  # register $t1 contains the size of the array

loop:
    blez  $t3,end     # if size is/becomes 0, end of processing
    lw    $t4,0($t0)
    lw    $t5,0($t1)
    addu  $t4,$t4,$t5
    sw    $t4,0($t2)  # update V3 vector element in memory
    addiu $t0,$t0,4    # advance pointers
    addiu $t1,$t1,4
    addiu $t2,$t2,4
    addiu $t3,$t3,-1   # decrement elements to process counter
    j     loop        # execute the loop another time

    # now, stop
end:    j     end
```

SIMULANDO COM O MARS

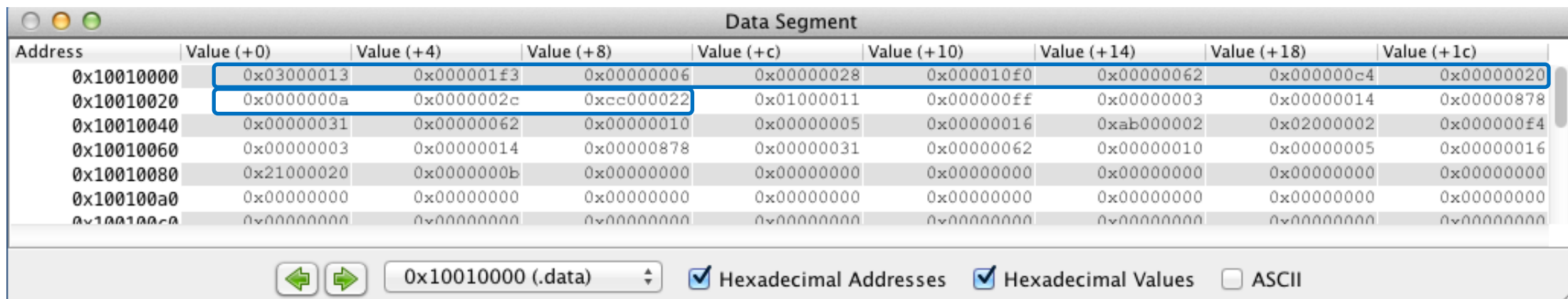
- Área de dados antes e depois da soma.
 - ✓ V3 começa com todos seus elementos em 0.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x01000011	0x000000ff	0x00000003	0x00000014	0x00000878
0x10010040	0x00000031	0x00000062	0x00000010	0x00000005	0x00000016	0xab000002	0x02000002	0x000000f4
0x10010060	0x00000003	0x00000014	0x00000878	0x00000031	0x00000062	0x00000010	0x00000005	0x00000016
0x10010080	0x21000020	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

Navigation: [Left] [Right] 0x10010000 (.data) [Hexadecimal Addresses] [Hexadecimal Values] [ASCII]

- ✓ Resultado da soma dos vetores V1 e V2 em V3.

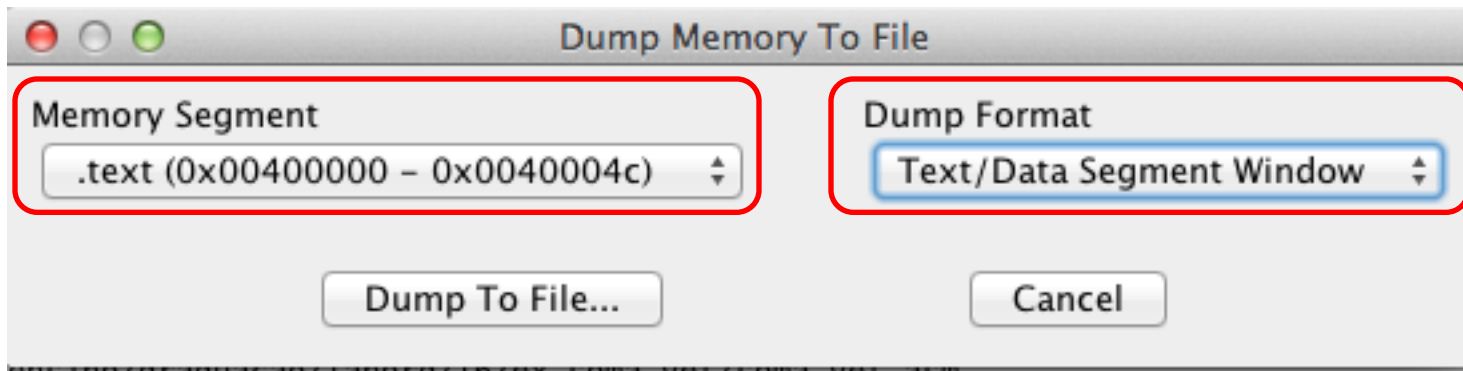


Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x03000013	0x000001f3	0x00000006	0x00000028	0x000010f0	0x00000062	0x000000c4	0x00000020
0x10010020	0x0000000a	0x0000002c	0xcc000022	0x01000011	0x000000ff	0x00000003	0x00000014	0x00000878
0x10010040	0x00000031	0x00000062	0x00000010	0x00000005	0x00000016	0xab000002	0x02000002	0x000000f4
0x10010060	0x00000003	0x00000014	0x00000878	0x00000031	0x00000062	0x00000010	0x00000005	0x00000016
0x10010080	0x21000020	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

Navigation: [Left] [Right] 0x10010000 (.data) [Hexadecimal Addresses] [Hexadecimal Values] [ASCII]

GERANDO O *DUMP* DE MEMÓRIA (PROG)

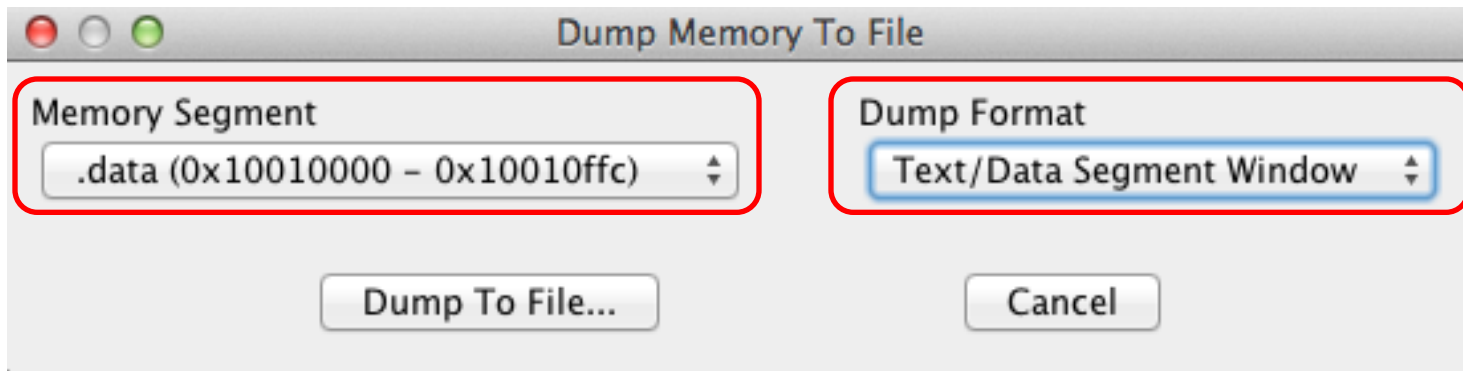
➤ No MARS: File → Dump Memory → arquivo **prog.txt**



Address	Code	Basic	Source
0x00400000	0x3c011001	lui \$1,0x00001001	10 1a \$t0,v1
0x00400004	0x3428002c	ori \$8,\$1,0x0000002c	
0x00400008	0x3c011001	lui \$1,0x00001001	11 1a \$t1,v2
0x0040000c	0x34290058	ori \$9,\$1,0x00000058	
0x00400010	0x3c011001	lui \$1,0x00001001	12 1a \$t2,v3
0x00400014	0x342a0000	ori \$10,\$1,0x00000000	
...			
0x0040004c	0x08100013	j 0x0040004c	30 end: j end

GERANDO O *DUMP* DE MEMÓRIA (DATA)

➤ No MARS: File → Dump Memory → arquivo **data.txt**



```
0x10010000 0x03000013 0x000001f3 0x00000006 0x00000028 0x000010f0 0x00000062 0x000000c4 0x00000020
0x10010020 0x0000000a 0x0000002c 0xcc000022 0x01000011 0x000000ff 0x00000003 0x00000014 0x00000878
0x10010040 0x00000031 0x00000062 0x00000010 0x00000005 0x00000016 0xab000002 0x02000002 0x000000f4
0x10010060 0x00000003 0x00000014 0x00000878 0x00000031 0x00000062 0x00000010 0x00000005 0x00000016
..... e continua .....
```

MEMÓRIAS NO FPGA – INSTRUÇÕES

```
library IEEE;
use IEEE.Std_Logic_1164.all;
library UNISIM;
use UNISIM.vcomponents.all;
```

```
entity program_memory is
    port( clock: in std_logic;
          address: in std_logic_vector(8 downto 0);
          instruction: out std_logic_vector(31 downto 0));
end program_memory;
```

```
architecture al of program_memory is
begin
```

➤ Blocos de 16Kbits. Para as instruções temos 512x32 (2Kbytes).

➤ **Limitação:** programa com no máximo 512 instruções.

✓ Interface com o processador:

- Endereço: 9 bits (512 palavras).
- Instrução: só leitura, 32 bits.

```
    programa : RAMB16_S36
    generic map (
        INIT_00 => X"342b00843c011001342a00003c011001342900583c0110013428002c3c011001",
        INIT_01 => X"2529000425080004ad4c0000018d60218d2d00008d0c0000196000098d6b0000",
        INIT_02 => X"000000000000000000000000000000000810001308100009256bffff254a0004",
        INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
        INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
        ...
        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'
    );
```

- ✓ Sempre habilitada para leitura (en=1).
- ✓ Nunca é escrita (we=0).
- ✓ Configurada desta forma como uma ROM.

✓ Primeira Instrução do Programa.

✓ Inicialização dos 16Kbits através de 64 vetores (3F) com strings de 256 bits (64 dígitos hexa).

```
end al;
```

MEMÓRIAS NO FPGA – DADOS

- 4 blocos de 16Kbits (8kbytes).
- Memória entrelaçada - Cada de 16Kbits contém 8 bits de uma região de 32 bits (uma palavra).
- Exemplo de conteúdo na memória de dados:

Endereço	Conteúdo
0x1001000	00 00 00 00
0x1001004	10 00 00 AA
0x1001008	20 00 BB 00
0x100100C	30 CC 00 00

- A inicialização da memória com este conteúdo é (lembrem-se que o simulador e o Hw trabalham com endereçamento “little-endian”. Logo, mem0 tem os bytes menos significativos de cada conjunto de 32 bits):

```
mem 3:  INIT_00 => X"00....30201000",
mem 2:  INIT_00 => X"00....CC000000",
mem 1:  INIT_00 => X"00....00BB0000",
mem 0:  INIT_00 => X"00....0000AA00",
```

MEMÓRIAS NO FPGA – DADOS

```
library IEEE;
use IEEE.Std_Logic_1164.all;
library UNISIM;
use UNISIM.vcomponents.all;
```

```
entity data_memory is
```

```
    port( clock, ce, we, bw: in std_logic;
          address: in std_logic_vector(10 downto 0);
          data_in: in std_logic_vector(31 downto 0);
          data_out: out std_logic_vector(31 downto 0));
```

```
end data_memory;
```

```
architecture a1 of data_memory is
```

```
    signal we3, we2, we1, we0 : std_logic;
```

```
begin
```

```
    we3 <= '1' when we='0' and bw='1' else '0';
    we2 <= '1' when we='0' and bw='1' else '0';
    we1 <= '1' when we='0' and bw='1' else '0';
    we0 <= '1' when we='0'           else '0';
```

Interface com o processador:

- Endereço: 11 bits ($2^{11}=2048$).
- Dados: **data_in** e **data_out**, 32 bits.

bw controla escrita de byte ('0' em sb).

```
-----
-- bytes 31 a 24      ✓  BRAM PARA OS BITS 31 a 24
-----
```

```
mem_3 : RAMB16_S9
```

```
    generic map (
```

```
        INIT 00 => X"000000000000000000002ab000000000000
```

MEMÓRIAS NO FPGA – DADOS

- Exemplo de configuração da BRAM dos bits menos significativos:

```
-----  
-- bytes 7 a 0  
-----  
mem_0 : RAMB16_S9  
  generic map (  
    INIT_00 => X"1605106231781403f402021605106231781403ff110000000000000000000000",  
    INIT_01 => X"00000000000000000000000000000000000000000000000000000000b20",  
    INIT_02 => X"000000000000000000000000000000000000000000000000000000000",  
    INIT_03 => X"00000000000000000000000000000000000000000000000000000000",  
    ...  
    INIT_3F => X"00000000000000000000000000000000000000000000000000000000")  
  port map (  
    CLK      => clock,  
    ADDR     => address,  
    EN       => '1',  
    WE       => we3,  
    DI       => data_in(7 downto 0),  
    DIP      => "0",  
    DO       => data_out(7 downto 0),  
    SSR      => '0'  
  );  
  
end a1;
```

COMO GERAR O VHDL DESTAS MEMÓRIAS?

➤ Usando o programa **le_mars** (fonte fornecido).

➤ Executando **le_mars** sem parâmetros:

```
*****
*      PROGRAMA QUE LE DOIS ARQUIVOS: instrucoes e codigo      *
*      uso:      ./le_mars <arquivo programa> <arquivo dados> [d]      *
*      [d] pode ser qualquer caracter, sendo opcional, indica debug      *
*      Please report all bugs to {fernando.moraes,ney.calazans}@pucrs.br      *
*****
```

➤ Executando: **./le_mars** prog.txt data.txt

```
----- PROCESSING INSTRUCTIONS
----- PROCESSING DATA
----- GENERATING VHDL FILE   memory.vhd
```

➤ Arquivo **memory.vhd** gerado

SIMULANDO O PROCESSADOR

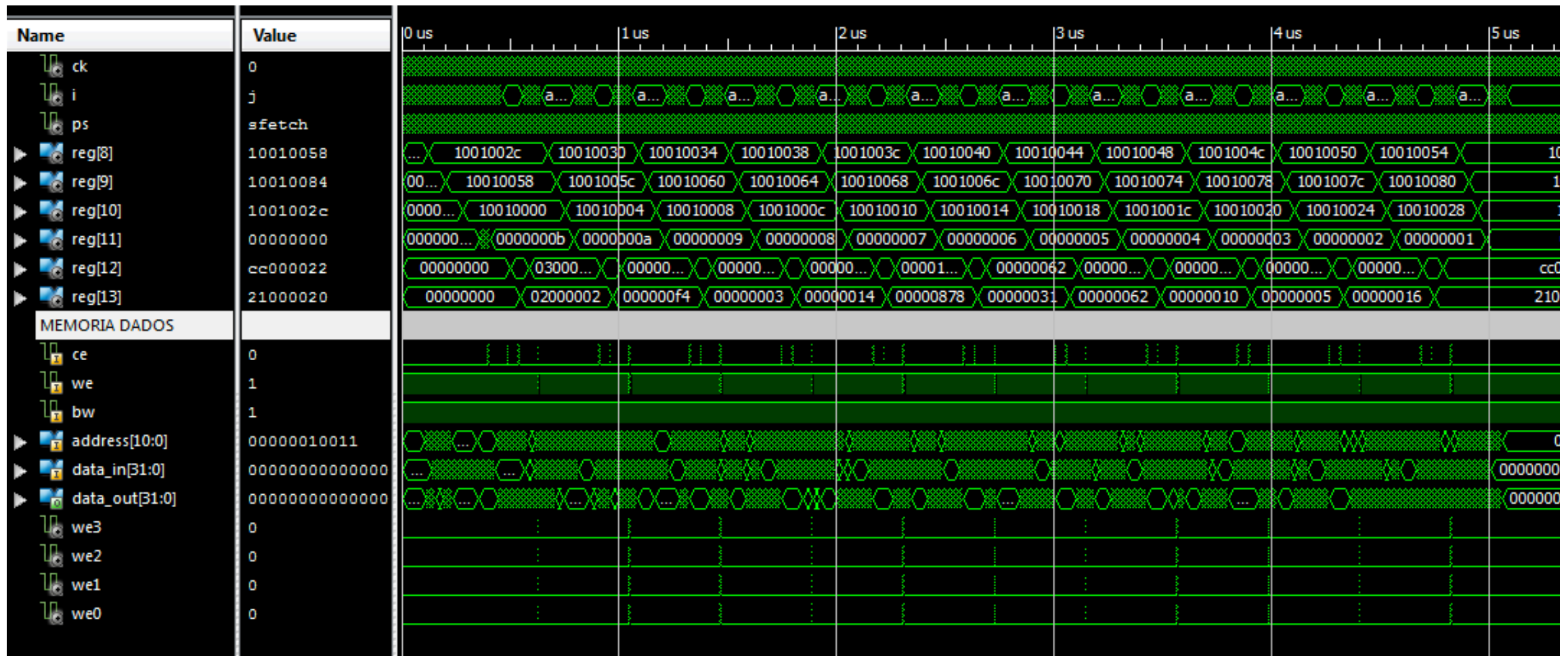
- Criar um projeto no ISE com 3 arquivos:
 - ✓ `memory.vhd` (gerado pelo programa `le_mars` a partir dos segmentos de texto e dados do MARS)
 - ✓ `MIPSmulti_com_BRAMS.vhd`
 - ✓ `mips_sem_perif_tb.vhd`
- Simular por 6 μ s (microsegundos)

Name	Value
ck	0
i	lw
ps	sld
reg[8]	1001002c
reg[9]	10010058
reg[10]	10010000
reg[11]	10010084
reg[12]	00000000
reg[13]	00000000

Timing diagram showing clock (ck), instruction pointer (i), program status (ps), and register values (reg[8] to reg[13]) over time. The diagram is divided into three segments: 250 ns, 300 ns, and 350 ns. Instructions are shown as boxes with labels like lui, ori, swbk, sfetch, sreg, salu. Register values are shown in hexadecimal.

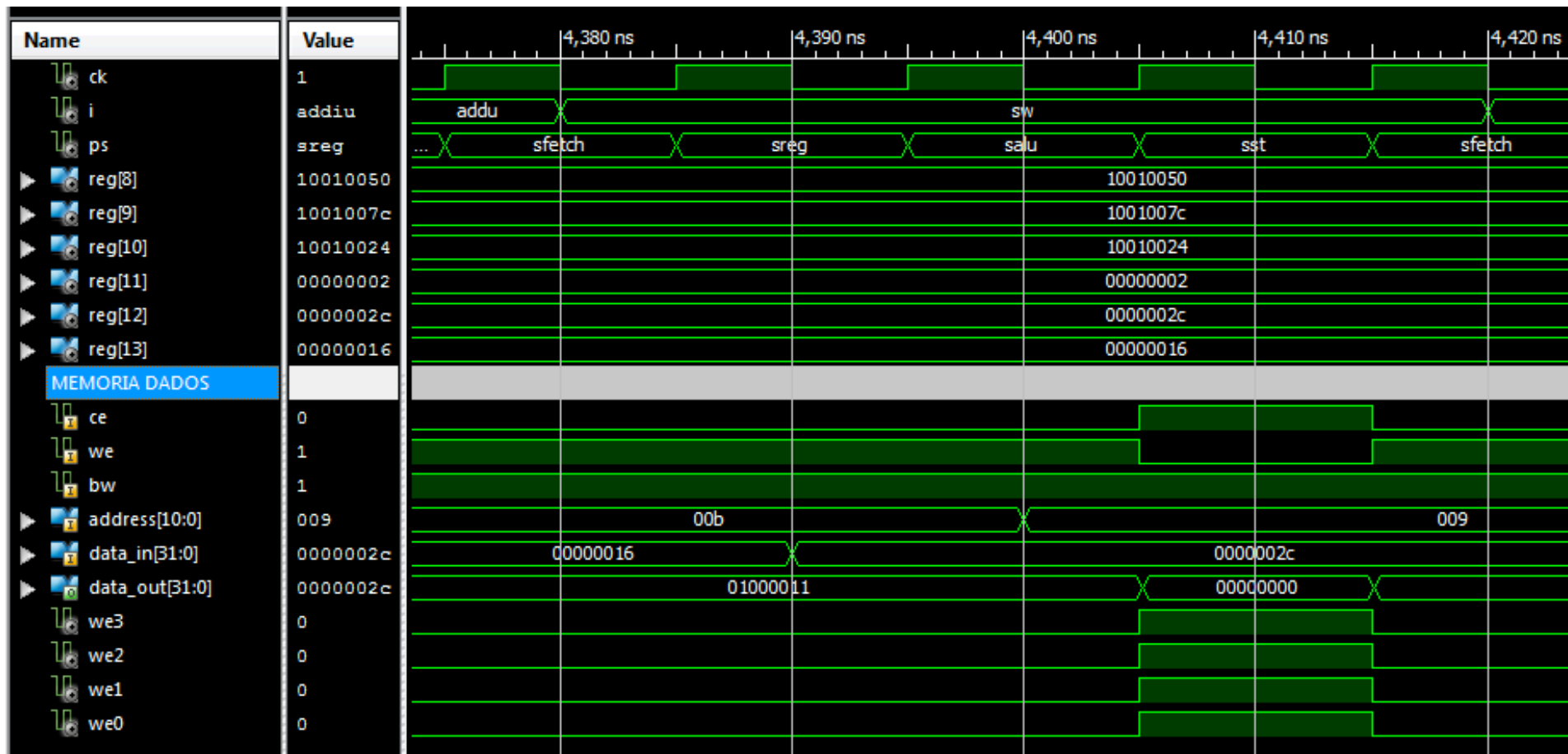
VISÃO MACRO DA SIMULAÇÃO

- Percebe-se 11 escritas na memória de dados (resultados das somas):

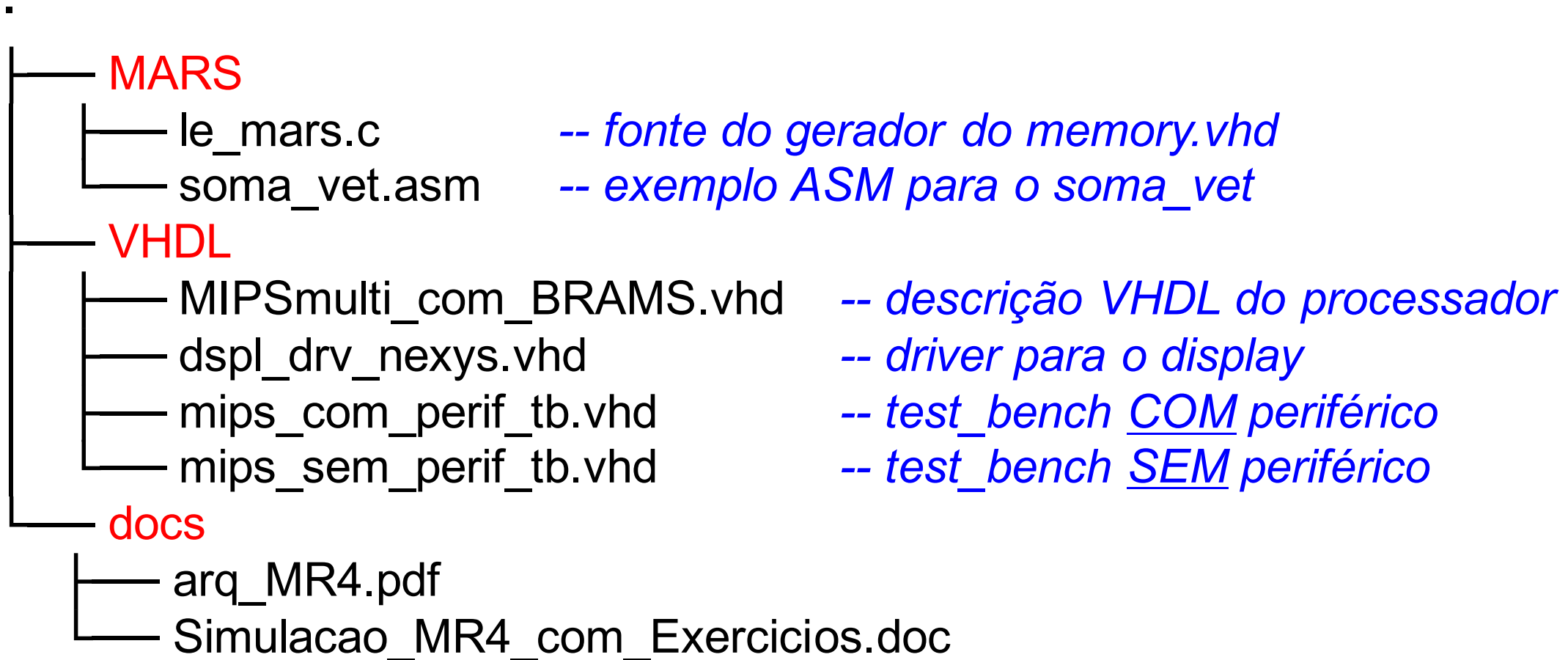


UMA ESCRITA NA MEMÓRIA

- Escrita (sw), leva 4 ciclos.
- Escrita do valor 2c (reg(12)= $\$t4$) no endereço 9.



MATERIAL DE APOIO



RESUMINDO O PROCESSO

1. Simular/validar no MARS um programa escrito em linguagem de montagem.
 - ✓ No exemplo, o `soma_vet.asm`.
2. Fazer os dumps de memória:
 - ✓ `prog.txt`
 - ✓ `data.txt`
3. Gerar o `memory.vhd` através do programa *le_mars*.
4. Simular:
 - ✓ `memory.vhd`
 - ✓ `MIPSmulti_com_BRAMS.vhd`
 - ✓ `mips_sem_perif_tb.vhd`

TRABALHO

PROJETO 1

1. Simular o exemplo soma de vetores para dominar o ambiente de simulação do processador MR4 (MIPS) com memórias do tipo BRAM:

- ✓ [memory.vhd](#)
- ✓ [MIPSmulti_com_BRAMS.vhd](#)
- ✓ [mips_sem_perif_tb.vhd](#)

➤ No material de apoio, tem um documento de referência:

- ✓ [Simulacao_MR4_com_Exercicios.doc](#).

PROJETO 2

2. Escrever um contador de segundos que conte de 000,0s a 999,9s (com precisão de décimos de segundos) e em seguida pare. Para tanto:
 - ✓ Implemente um rotina que conte exatamente 1 décimo de segundo - basta fazer um laço que execute cerca de 5 milhões de instruções do MIPS, supondo que o clock do processador é de 50MHz.
 - ✓ O algoritmo do programa é simples e pode ser resumido em quatro passos:
 1. Zerar quatro registradores do MIPS (digamos \$s0, \$s1, \$s2 e \$s3).
 2. Escrever na saída os 4 valores, correspondendo a décimos, unidades, dezenas e centenas de segundos. Se todos estes valores forem simultaneamente 9, parar.
 3. Chamar uma função que espera exatamente 1 décimo de seg.
 4. Incrementar contagem e ir para passo 2.

PROJETO 2

- Validar este programa no MARS:
 - ✓ OBS: usem um laço com valor de contagem menor.
- Sugestão para a área de dados (endereços de entrada e saída):

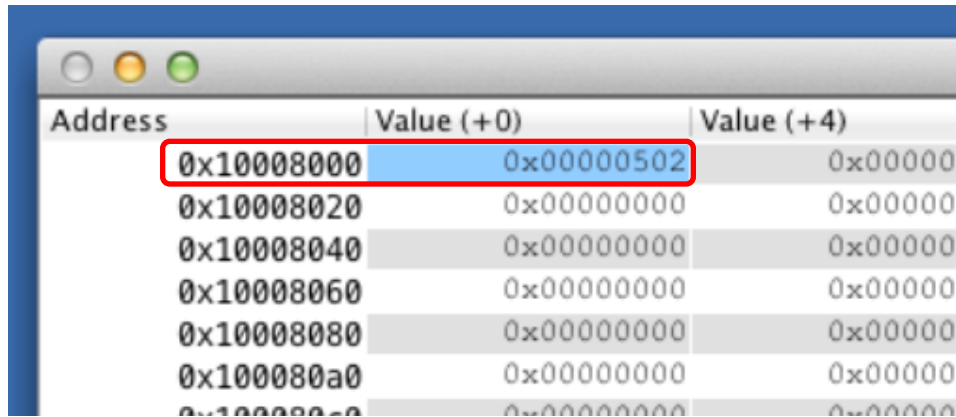
```
out_dec_s:    .word    0x10008000
out_un:       .word    0x10008001
out_dz:       .word    0x10008002
out_ct:       .word    0x10008003
```

- Como escrever nesta área de dados

```
la $t0, out_dec_s # obtém endereço de reg periférico décimos
lw $t0, 0($t0)
sb $s0, 0($t0)    # escreve valor de contagem ($s0) no periférico
```

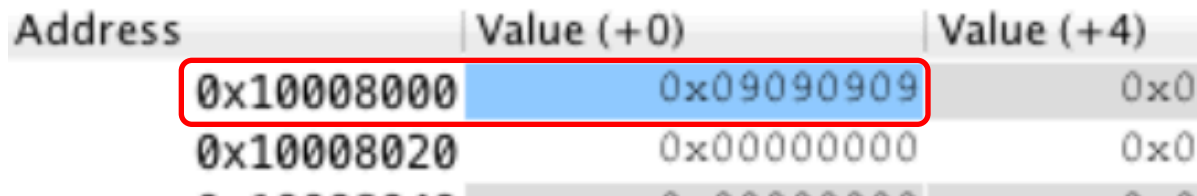
PROJETO 2

- Simulação no MARS:
- Estado parcial da simulação (0/0/5/2)



Address	Value (+0)	Value (+4)
0x10008000	0x00000502	0x00000000
0x10008020	0x00000000	0x00000000
0x10008040	0x00000000	0x00000000
0x10008060	0x00000000	0x00000000
0x10008080	0x00000000	0x00000000
0x100080a0	0x00000000	0x00000000
0x100080c0	0x00000000	0x00000000

- Final da contagem: 09 / 09 / 09/ 09



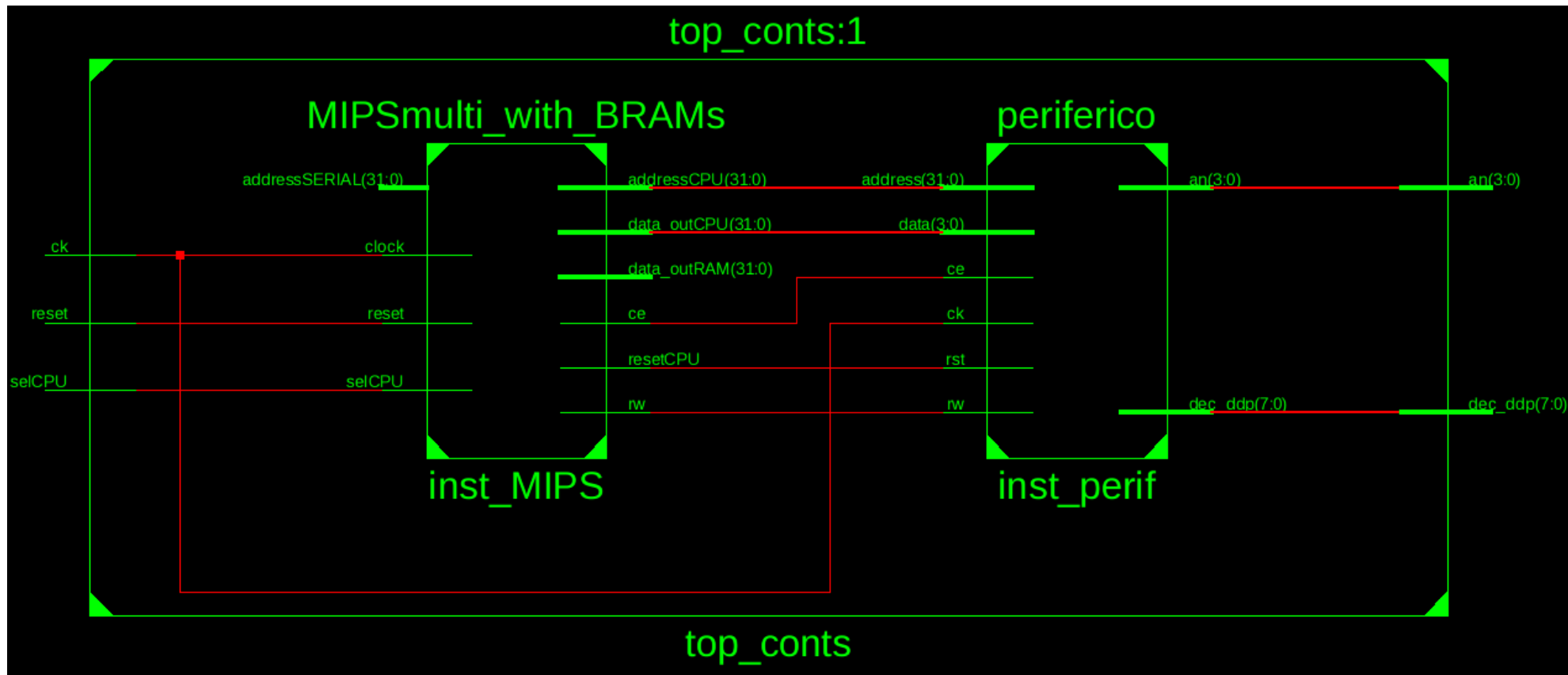
Address	Value (+0)	Value (+4)
0x10008000	0x09090909	0x00000000
0x10008020	0x00000000	0x00000000
0x10008040	0x00000000	0x00000000

PROJETO 3

3. Acrescentar hardware ao MIPS_multi_with_BRAMs, um conjunto de 4 registradores de 4 bits para armazenar o valor de contagem de segundos. Como realizar isto?
 - a. Os 4 registradores de saída estarão mapeados nos endereços do início da área apontada pelo registrador \$gp do MIPS (que no MARS possui valor inicial 0x10008000).
 - b. Note que os registradores externos devem ser síncronos em relação ao processador (ou seja, usam os mesmos sinais de clock e reset que este).
 - c. Produza um processo de decodificação individual para cada um dos quatro registradores, baseado nos sinais ce, rw e addressCPU gerados pelo processador.
 - d. Após criar os registradores, ligue cada um destes a um dos *displays* da placa Nexys2, através do *driver* de *display* já usado em outras ocasiões.

PROJETO 3

- Exemplo de nível mais alto do protótipo – o retângulo mais externo corresponde aos limites do FPGA.

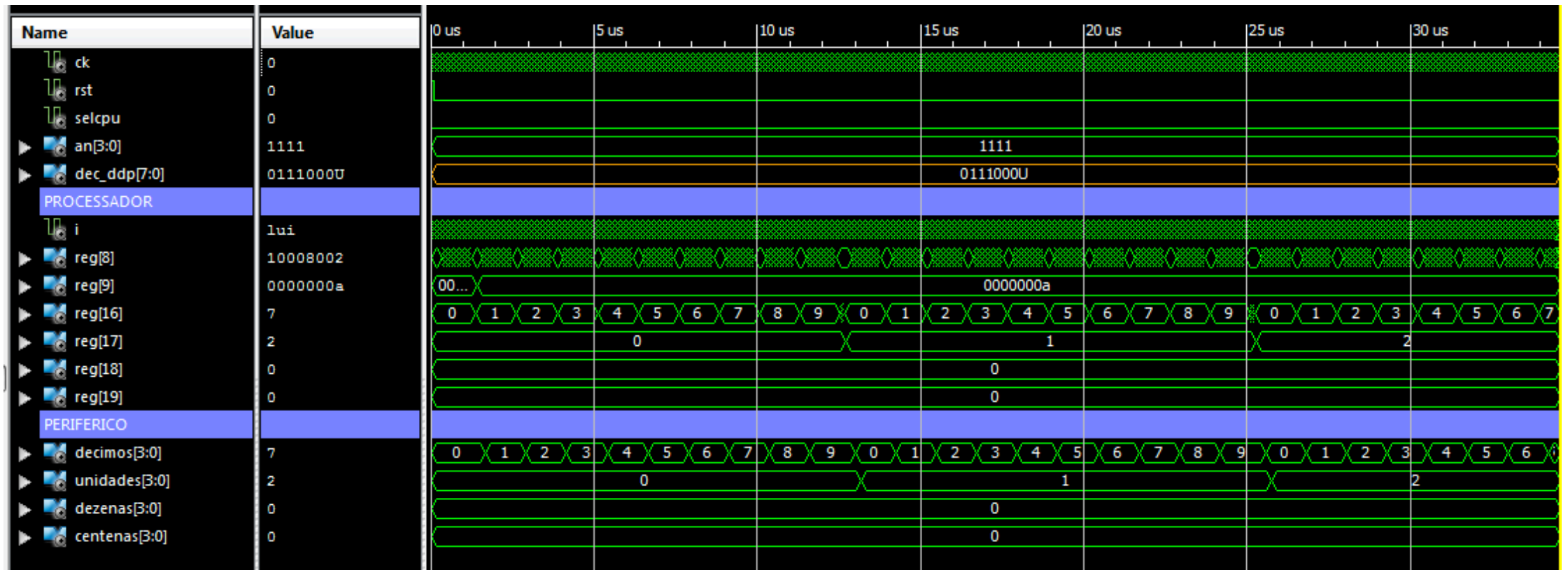


PROJETO 3

- Simular a aplicação, visualizando nos 4 registradores de contagem o valor do contador.
- 6 Arquivos VHDL para a simulação:
 - ✓ `memory.vhd`: programa contador
 - ✓ `dspl_drv_nexys.vhd`: driver que aciona os *displays* (fornecido).
 - ✓ `periferico.vhd`
 - ✓ `MIPSmulti_com_BRAMS.vhd`
 - ✓ `top_conts.vhd`: contendo o periférico e o processador
 - ✓ `mips_com_perif_tb.vhd` (fornecido)

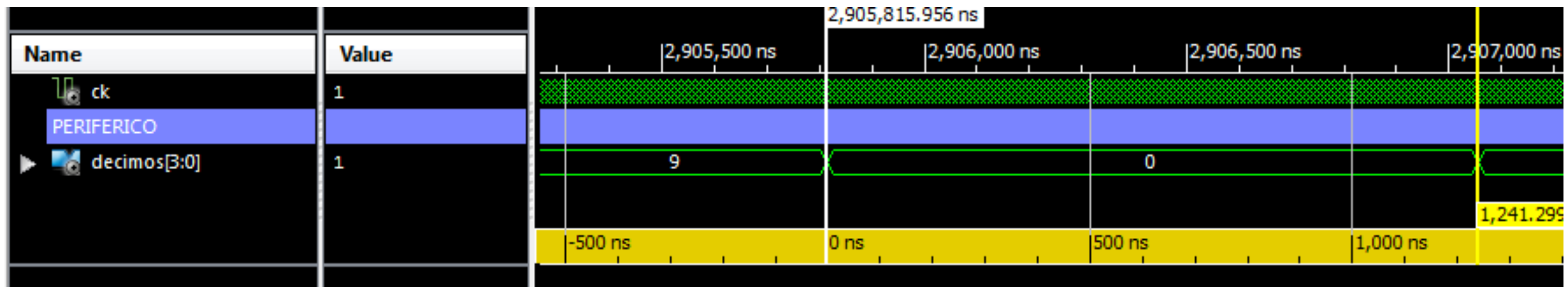
SIMULAÇÃO INICIAL

- Visualizar registradores reg[8], reg[9], reg[16]-reg[19] (\$s0-\$s3).
- No periférico, visualizar os registradores.



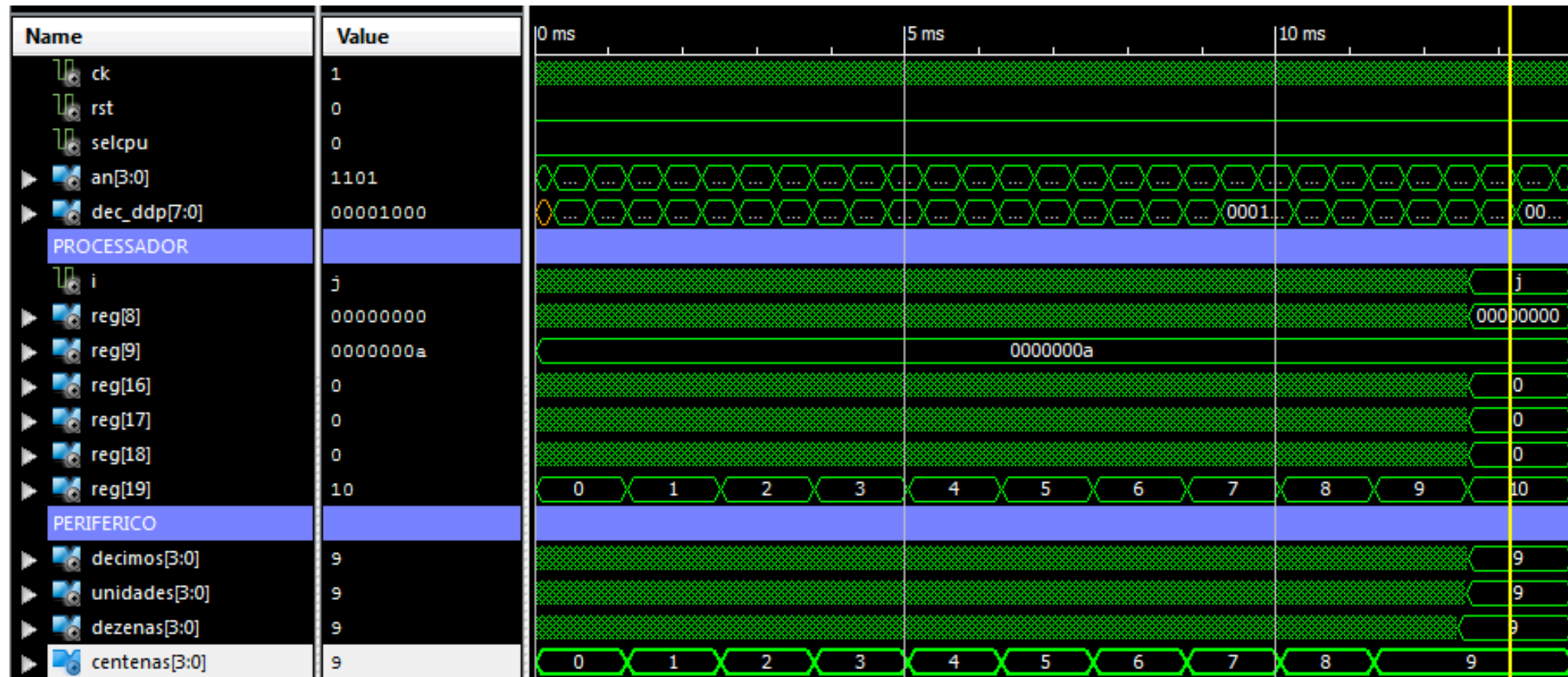
CONTANDO O TEMPO DE SIMULAÇÃO

- Medir o tempo para 1 décimo de segundo.
- No meu código foi 1240 ns (varia de acordo com a forma como o programa é escrito).
- Multiplicando $1240\text{ns} * 10000 = 0,0124 = \mathbf{12,4\text{ ms}}$



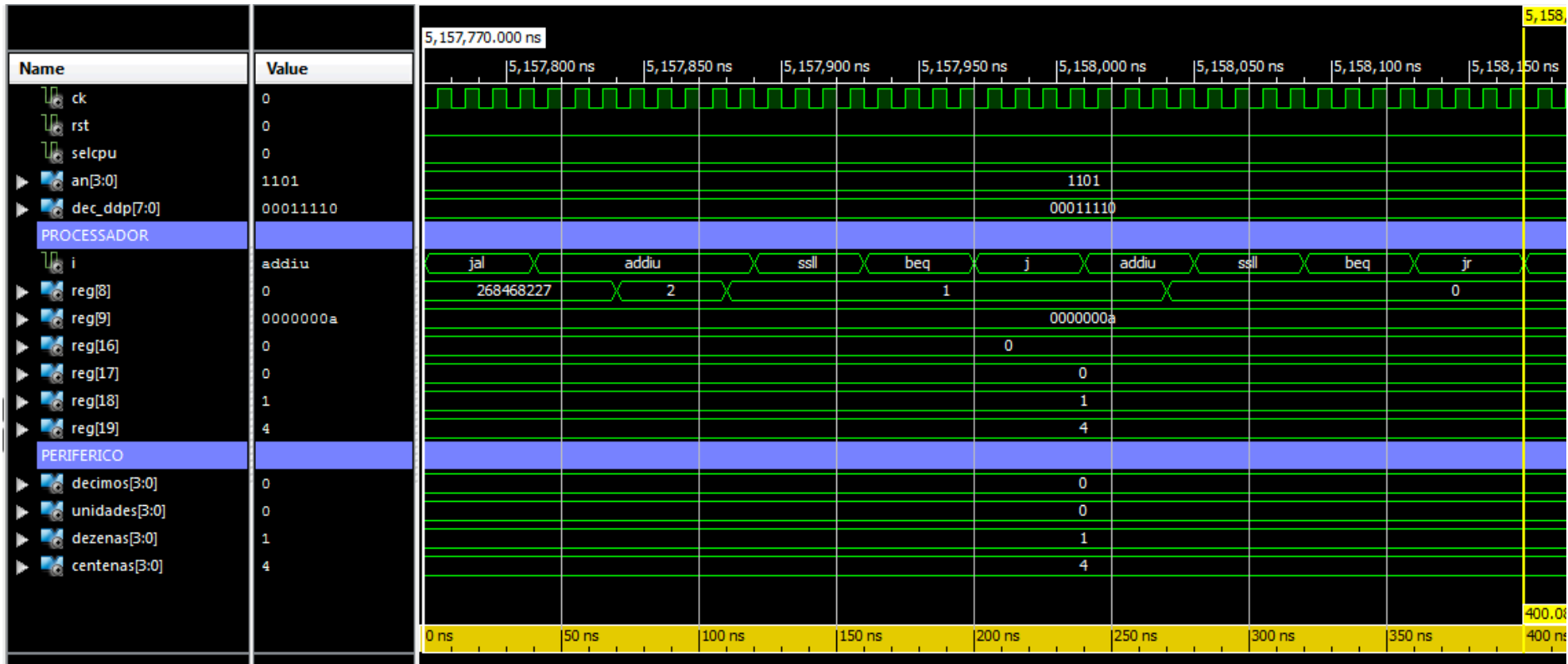
SIMULANDO AGORA POR 13ms

- Visualizar os registradores reg[8], reg[9], reg[16]-reg[19] (\$s0-\$s3).
- 13 ms (milissegundos).



ROTINA DO TEMPO

- Chamada por jal.
- Inicializa \$t0 com 2, e decrementa até chegar a zero.
- Volta com jr \$ra.



PROJETO 4

4. Prototipar o sistema completo:

- ✓ Não esquecer de dar a devida atenção à definição da interface do FPGA com o mundo externo, através da especificação do arquivo UCF.
- ✓ Não esquecer ao prototipar o sistema de garantir que as memórias de instruções e dados são carregadas desde o início da execução com as informações necessárias para executar o programa que produz o contador de segundos.
- ✓ Modificar o valor de controle da contagem para precisão de décimo de segundo.

RESUMO DO TRABALHO 3B

- O **Trabalho 3B** (T3B) consiste em um arquivo compactado (.zip) contendo:
 - ✓ Um relatório em PDF descrevendo a estrutura de hardware adicionada, bem como do programa executado no protótipo.
 - ✓ Código assembly da aplicação, comentado.
 - ✓ Diretório com os 6 VHDLs, o UCF e o .bit.
 - ✓ O projeto ISE completo do protótipo gerado.
 - ✓ **DEVE-SE** mostrar a operação do projeto em uma plataforma de prototipação ao professor em alguma aula.