1 TUTORIAL DE SIMULAÇÃO (VHDL) DO PROCESSADOR MR4 c/ BRAMS

Este tutorial visa iniciar os alunos à simulação de um processador que dá suporte a um subconjunto amplo de instruções da arquitetura MIPS, denominado MR4. Este processador implementa uma organização que pode executar uma boa parte das instruções da arquitetura MIPS R2000. Uma especificação detalhada do MR4 está disponível na área de download da disciplina, ou diretamente no link http://www.inf.pucrs.br/~calazans/undergrad/orgcomp/arq_MR4.pdf. 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.

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. Assim, 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 MR4.

1.1 O ambiente de simulação do processador MR4

O ambiente de simulação/síntese a ser usado neste tutorial encapsula o processador MR4 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 controles para ler o conteúdo da memória de dados.

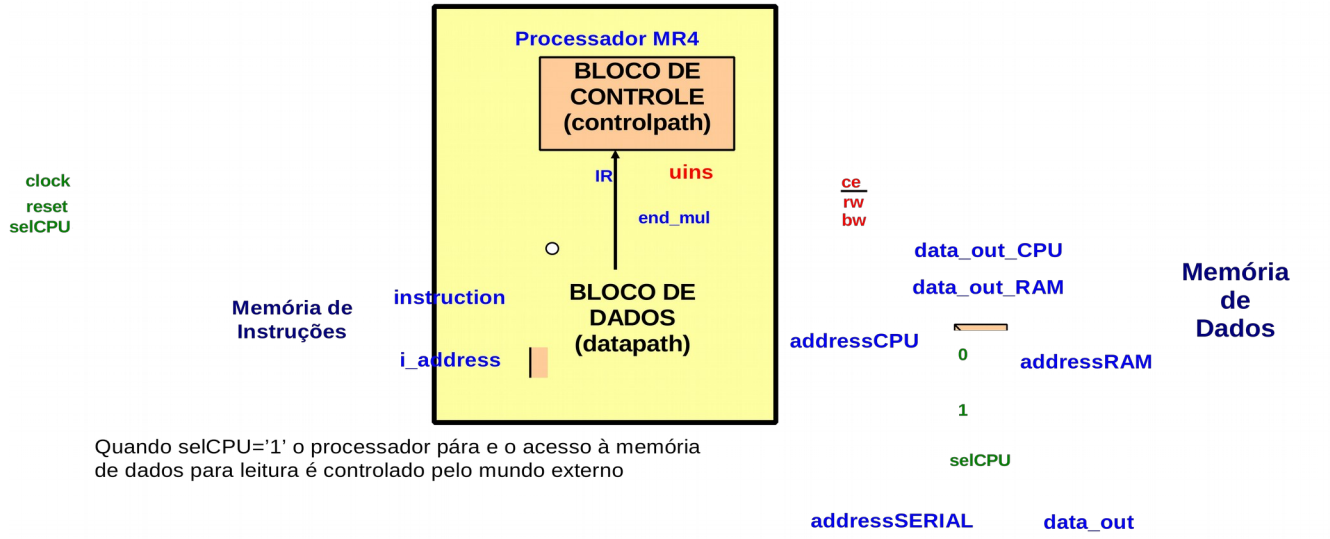

Figura 1 – Diagrama de blocos do ambiente contendo o processador MR4 e suas memórias de instruções e dados. Deve-se notar que o bloco que constitui o processador MR4 em si (retângulo amarelo e seus conteúdos) não aparece de forma explícita na descrição VHDL fornecida, apenas seus conteúdos (o Bloco de Controle e o Bloco de Dados) existem como pares entidade/arquitetura instanciados e conectados às memórias de instruções e de dados. Vários sinais de interfaces dos blocos foram omitidos na Figura, para aumentar a clareza do diagrama.

A Figura 2 mostra uma visão parcial do código VHDL do ambiente. Basicamente, este arquivo contém as instâncias dos 4 blocos principais do ambiente: memória de instruções, memória de dados, bloco de dados do MR4 e bloco de controle do MR4, mais lógica de "cola" para permitir o controle da comunicação entre as diversas partes do ambiente entre si.

```
entity MR4_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 MR4_with_memories;

architecture a1 of MR4_with_memories is
--- vários sinais são declarados aqui (não mostrados, por fins de concisão)
begin
    dp: entity work.datapath port map
    (ck=>clock, rst=>rst_cpu, end_mul=>end_mul, end_div =>end_div, IR_OUT=>IR,
        i_address=>i_address, instruction=>instruction, d_address=>addressCPU,
        data_out=>data_out_CPU, data_in=>data_out_RAM, uins=>uins);

    ct: entity work.control_unit port map
    (ck=>clock,rst=>rst_cpu,end_mul=>end_mul, end_div=>end_div, IR=>IR, uins=>uins);

    int_address <= i_address(10 downto 2);

    m1: entity work.program_memory port map    ---- MEMÓRIA RAM COM AS INSTRUÇÕES
    (clock=>clock, address=>int_address, instruction=>instruction);

    m2: entity work.data_memory port map       ---- MEMÓRIA RAM COM OS DADOS
    (clock=>clock, ce=>uins.ce, we=> uins.rw, bw=>uins.bw, data_in=>data_out_CPU,
        address=>addressRAM(12 downto 2), data_out=>data_out_RAM);

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

    rst_cpu <= reset or selCPU;

    data_out <= data_out_RAM;
end a1;
```

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

1.2 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 de algum porte (na ordem das dezenas a milhares de Kbits). 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 MR4 faz acesso.

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

1.2.1 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

[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.

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 `vcomponents`). O componente adequado para usar aqui é denominado RAMB16_S36[2]. 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"8e52000343200083c0110018e310000343100043c011001343d08003c011000",
    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 MR4. 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).

[2] A nomenclatura RAMB16_S36 vem da Xilinx, e pode se revelar um pouco confusa para neófitos. Neste caso, o primeiro numeral (16) significa que o bloco de 18 Kbits está configurado sem acesso a paridade (portanto disponibilizando apenas 16 Kbits dos 18 Kbits de memória existentes. O segundo número (36) significa que cada leitura/escrita de dado da/na memória faz acesso a 36 bits. Descontando-se os 4 bits de paridade (devido ao fato de não se usar esta, como visto na interpretação do primeiro numeral) sabe-se que a porta de acesso é de 32 bits. BRAMs são memórias de porta dupla. Neste caso, a existência de apenas 1 parâmetro após o caractere sublinhado (_) indica que apenas uma porta é usada. Outro exemplo de configuração seria RAMB16_S8_S16, onde não se usa paridade (RAMB16) e se tem duas portas de acesso aos 16Kbits, uma de 8 bits de dados (S8) e uma de 16 bits de dados (S16).

1.2.2 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.

1.3 Como se gera o conteúdo para os INITs?

O ponto de partida é um *dump* de memória gerado por um programa montador. No ambiente MARS, após carregar o arquivo fonte, deve-se usar a opção de menu ***File → Dump Memory***, escolhendo fazer *dump* dos segmentos de texto e de dados, usando o formato de *dump* ***Text/Data Segment Window***. No ambiente PCSPIM, pode-se simplesmente salvar um arquivo ***PCSpim.log***, o qual contém o código objeto e a área de dados e muito mais informações em um único arquivo.

O exemplo da Figura 4 corresponde ao programa que gerou os INITs acima. Este procedimento de inicialização é propenso a erros, se feito de forma manual. Por isto são disponibilizados os seguintes recursos no diretório MARS (um outro diretório possui material similar para o PCSPIM):

1. **le_mars.exe** – executável que lê um arquivo de nome ***Mars.log*** (um arquivo que deve conter o *dump* de texto e de dados) e gera o arquivo VHDL denominado ***memory.vhd***;
2. **cygwin1.dll** – *dll* necessária para executar o programa **le_mars.exe**;
3. **le_mars.c**– código fonte do programa **le_mars.exe** (não é necessário. Serve apenas para poder ilustrar a funcionalidade do mesmo, e para permitir alterações no programa para quem entende o suficiente do que ele faz);
4. **soma_vet.asm** – arquivo fonte com um programa exemplo em linguagem de montagem do MIPS.

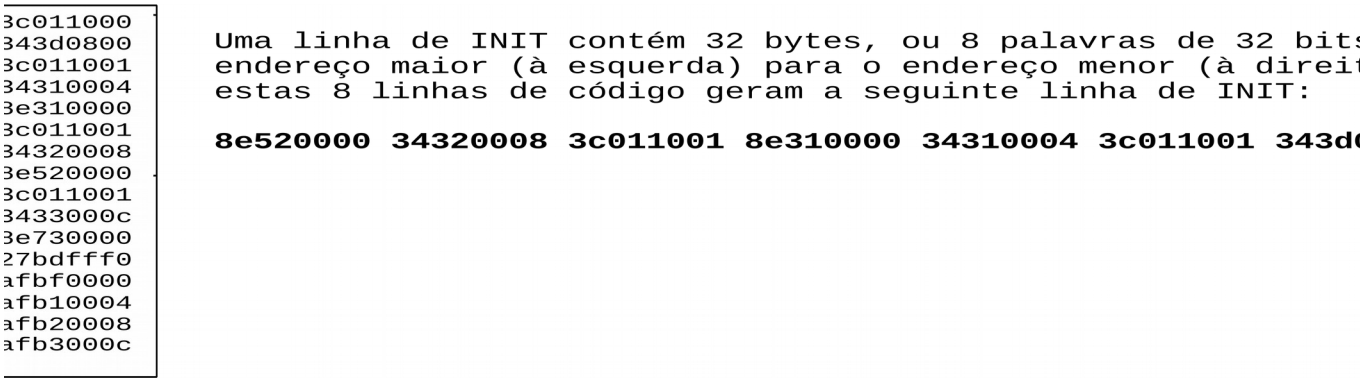

Figura 4 – Exemplo de trecho de arquivo de *dump* usado para gerar o arquivo de inicialização das memórias do processador MR4.

2 Tarefas Acessórias Iniciais

O objetivo desta Seção é prover um treinamento básico em como gerar o código objeto de um programa qualquer do MIPS e preparar os arquivos necessários para realizar uma simulação VHDL do MR4 executando tal programa.

1. Abra o arquivo ***soma_vet.asm*** no simulador MARS, monte o código objeto (tecla **F3**). Note que o programa 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, por exemplo ***Mars_t.log*** e ***Mars_d.log***, usando o formato de *dump* ***Text/Data Segment Window***. Em seguida, junte os dois arquivos em um único com o formato dado no exemplo da Figura 5. Note o

texto adicionado antes da área de programa (***Text Segment*** exatamente a partir da primeira coluna da linha) e o texto adicionado antes da área de dados (***Data Segment*** exatamente a partir da primeira coluna da linha). Estes textos devem obrigatoriamente aparecer em alguma linha do arquivo antes do início dos respectivos segmentos que eles introduzem. Linhas diferentes destas e das linhas de especificação de código e dados são ignoradas (como comentários. Ver, por exemplo, as linhas contendo = = = = = na Figura 5 abaixo). Note, também, que não é necessário inserir todo o segmento de dados, mas apenas o trecho de dados onde os endereços de memória contêm valores diferentes de zero (pois o *dump* de memória da área de dados é sempre de uma área de tamanho fixo, e grande).

2. Executar o programa **le_mars.exe**[3] no diretório onde foi gravado o arquivo Mars.log. Para isto basta dar um duplo clique sobre o ícone do programa, ou executá-lo via linha de comando. A execução do **le_mars.exe** gera o arquivo VHDL **memory.vhd**, contendo o código objeto mapeado nas memórias de instruções e de dados. Importante: o arquivo **cygwin1.dll** deve estar no mesmo diretório do executável.
3. Criar um projeto no Active-HDL adicionando a estes os arquivos:
    i. **memory.vhd** (recém gerado no passo anterior),
    ii. **mrstd.vhd** (presente no diretório MR4), e
    iii. **mrstd_tb.vhd** (presente no diretório MR4).

Observação: Caso a biblioteca `UNISIM` não esteja disponível, tente adicioná-la dando um duplo clique na biblioteca do projeto e na janela do browser de biblioteca escolhendo a opção Attach Library → Unisim. Caso não ache a biblioteca, talvez seja necessário baixar e compilar a mesma. Contacte o professor para resolver o problema.

```
  Text Segment
=================================
0x00400000  0x3c011001  lui $1,4097         10      la      $t0, V1          # generate pointer to V1 source vector
0x00400004  0x3428002c  ori $8,$1,44
0x00400008  0x3c011001  lui $1,4097         11      la      $t1, V2          # generate pointer to V2 source vector
0x0040000c  0x34290058  ori $9,$1,88
0x00400010  0x3c011001  lui $1,4097         12      la      $t2, V3          # generate pointer to V3 target vector
0x00400014  0x342a0000  ori $10,$1,0
0x00400018  0x3c011001  lui $1,4097         14      la      $t3, size        # get address of size
0x0040001c  0x342b0084  ori $11,$1,132
0x00400020  0x8d6b0000  lw $11,0($11)       15      lw      $t3, 0($t3)      # register $t1 contains the array size
0x00400024  0x19600009  blez $11,9           18      blez    $t3, end         # if size is/becomes 0, end of processing
0x00400028  0x8d0c0000  lw $12,0($8)        19      lw      $t4, 0($t0)
0x0040002c  0x8d2d0000  lw $13,0($9)        20      lw      $t5, 0($t1)
0x00400030  0x018d6021  addu $12,$12,$13     21      addu    $t4, $t4, $t5
0x00400034  0xad4c0000  sw $12,0($10)       22      sw      $t4, 0($t2)      # update V3 vector element in memory
0x00400038  0x25080004  addiu $8,$8,4        23      addiu   $t0, $t0, 4      # advance pointers
0x0040003c  0x25290004  addiu $9,$9,4        24      addiu   $t1, $t1, 4
0x00400040  0x254a0004  addiu $10,$10,4      25      addiu   $t2, $t2, 4
0x00400044  0x256bffff  addiu $11,$11,-1     26      addiu   $t3, $t3, -1     # decrement elements to process counter
0x00400048  0x08100009  j 4194340            27      j       loop             # execute the loop another time
0x0040004c  0x08100013  j 4194380            30   end:       j       end


Data Segment
=================================
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
```

Figura 5 – Formato de arquivo aceito pelo programa le_mars.exe para gerar o arquivo VHDL com instanciação e preenchimento de memórias de instruções e de dados para o processador MR4.

4. Compilar os arquivos fonte VHDL e inicializar a simulação. Em uma janela do tipo forma de onda inserir os sinais conforme mostrado na Figura 6. Note que se trata de uma simulação completa do

[3] O arquivo fonte do programa **le_mars.exe** é o arquivo **le_mars**.c. Este programa considera como sua entrada um arquivo de nome "**Mars.log**" e gera como saída um arquivo de nome "**memory.vhd**". Caso deseje efetuar alguma alteração neste programa, basta realizá-la, compilar novamente o arquivo, e executar o novo programa.

programa (***soma_vet.asm***) de soma de dois vetores de 11 elementos. Algumas das informações perceptíveis na Figura são:

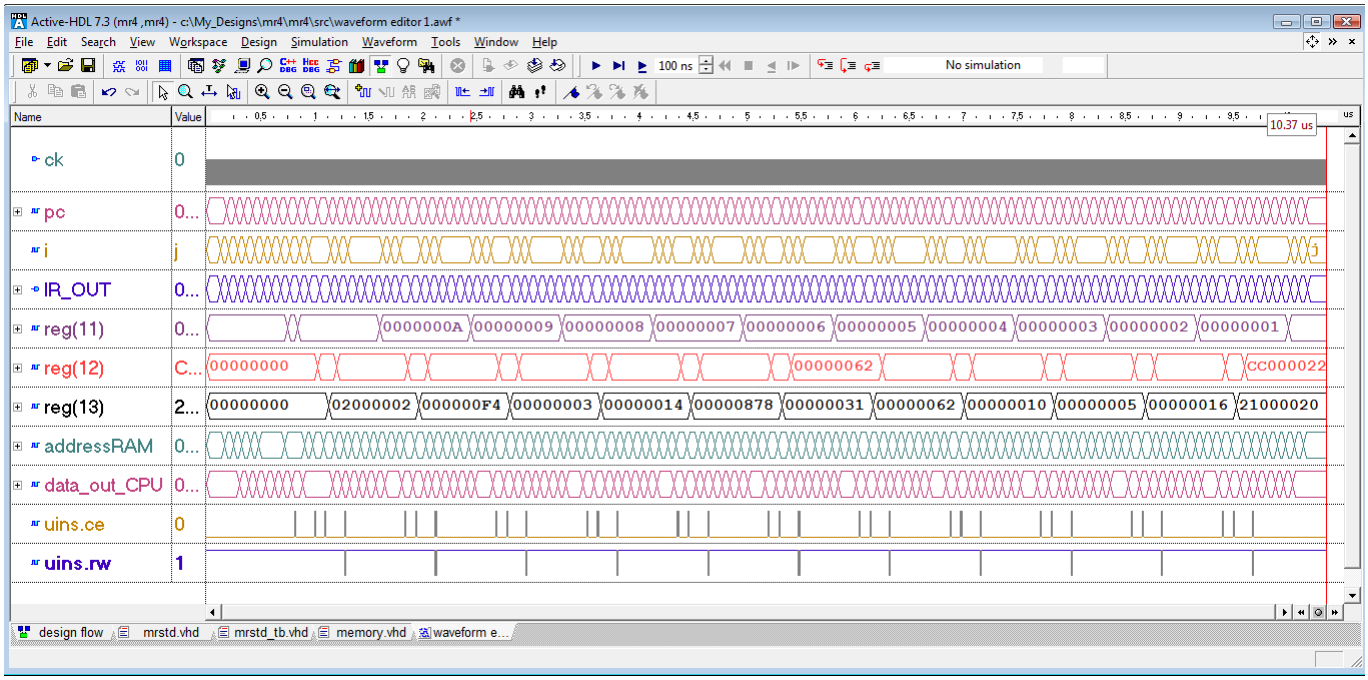

Figura 6 – Simulação completa da execução do programa *soma_vet.asm* no processador MR4.

A. O tempo de simulação do programa completo (10.37 µs);
B. As mudanças nos registradores $t3 (Reg(11) na forma de onda), $t4 (Reg(12) na forma de onda) e $t5 (Reg(13) na forma de onda), que contém respectivamente o contador de elementos que vai decrescendo de 11 (0xB) a 0, o resultado da soma dos dois elementos correntes de V1 e V2 e o elemento corrente de V2;
C. Os momentos de escrita na memória, quando os sinais uins.ce=‘1’ e uins.rw=‘0’. Isto ocorre exatamente 11 vezes na Figura.