

1 TUTORIAL DE SIMULAÇÃO (VHDL) DO PROCESSADOR MIPS_S com BRAMS

Este tutorial visa iniciar os alunos à simulação do sistema MIPS_S_withBRAMs, que inclui um processador que dá suporte a um subconjunto de instruções da arquitetura MIPS, denominado MIPS_S. Este processador implementa uma organização que pode executar um subconjunto das instruções da arquitetura MIPS R2000 (exatamente 37 instruções). Uma especificação detalhada do processador do MIPS_S_withBRAMs está disponível em:

https://www.inf.pucrs.br/~calazans/undergrad/OAP/Apoio/Arg_MIPS_Multiciclo_Spec.pdf¹

Este material também pode ser encontrado no material de apoio para este trabalho.

As principais instruções da MIPS R2000 às quais a arquitetura MIPS_S **não dá suporte** são as instruções de manipulação de números de ponto flutuante, as instruções de acesso à memória por 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 prototipar o sistema em FPGAs de plataformas disponíveis, ao processador acrescentou-se memórias de instruções e dados baseadas em estruturas de hardware disponíveis internamente em FPGAs da Xilinx, conforme explicado abaixo. Nos modelos de simulação de processadores vistos na disciplina teórica companheira desta, um *testbench* complexo permite carregar memórias não sintetizáveis com um programa qualquer (que use o subconjunto de instruções ao qual a MIPS_S dá suporte) e seus dados, isto no início da simulação. Em relação à abordagem de simulação vista na disciplina teórica, se procede aqui de forma diferente, usando um processo que sintetiza 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, por exemplo, o ambiente VIVADO. 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 MIPS_S_withBRAMs.

1.1 O ambiente de simulação do processador MIPS_S_withBRAMs

O ambiente de simulação/síntese a ser usado aqui encapsula o processador MIPS_S 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 portas da interface externa, além do **clock** e do **reset**, são sinais que permitem a entidades externas ler o conteúdo da memória de dados e eventualmente capturar endereços de dispositivos de saída mapeados em memória.

¹ Note que a organização MIPS_S usada aqui e aquela usada na disciplina teórica são algo distintas. As principais diferenças são: (1) a MIPS_S vista na disciplina teórica é uma organização mais recente. Ela é síncrona e emprega exclusivamente a borda de subida do sinal **clock** para sincronizar sua operação, enquanto a MIPS_S usada aqui também é síncrona, mas emprega a borda de subida do **clock** para sincronizar ações no Bloco de Controle e a borda de descida do mesmo sinal para sincronizar ações no Bloco de Dados; (2) A MIPS_S da disciplina teórica é voltada para simulação, sobretudo (embora seja sintetizável). A MIPS_S usada aqui foi adaptada para lidar eficientemente com as blocos de memórias de FPGAs Xilinx (denominados BRAMs) que podem ser configurados com duas portas de acesso separadas. Assim, o processador teve sua interface externa alterada para usar dois conjuntos de pinos de acesso à memória de dados: 32 fios de saída para escrita de um dado na memória (sinal **data_out**) e 32 fios de entrada para leitura da memória (sinal **data_in**).

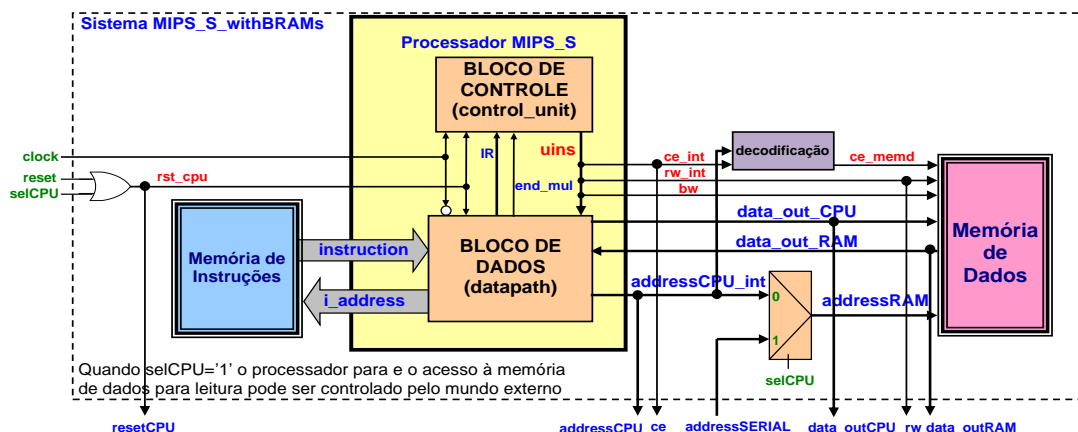


Figura 1 – Diagrama de blocos do sistema MIPS_S_withBRAMs contendo o processador MIPS_S, suas memórias de instruções e dados e alguma lógica de cola entre estes. O processador MIPS_S em si (retângulo amarelo e seus conteúdos) conecta-se às memórias de instruções e de dados. Alguns sinais ou nomes de sinais podem ter sido 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: a memória de instruções, a memória de dados, e o processador MIPS_S_withBRAMs, acrescido de uma “lógica de cola” (*glue-logic*) simples para permitir o controle da comunicação entre as diversas partes do ambiente.

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

architecture MIPS_S_withBRAMs of MIPS_S_withBRAMs 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;

  I_MIPS_S: entity work.MIPS_S
    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 MIPS_S_withBRAMs;
```

Figura 2 – Código VHDL (parcial) que implementa 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 dentro de FPGAs para implementar memórias de algum porte (na ordem das dezenas a milhares de Kbits). Os FPGAs da família Artix-7 da Xilinx possuem certa

quantidade de blocos de memória de 16 Kbits configuráveis², denominados de BlockRAMs ou BRAMs. BRAMs foram usadas neste ambiente para implementar os blocos de memória necessários na MIPS_S_withBRAMs.

Descreve-se a seguir o mapeamento das memórias de instruções e dados para BRAMs em FPGAs da família Artix-7 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 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 BRAM 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. Estes valores podem ser aumentados sem muita dificuldade.**

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_S36³. Para usar tal componente, basta declarar a biblioteca UNISIM e o pacote vcomponents 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"afb3000cafb20008afb10004afb000027bdf08e7300003433000c3c011001",
    INIT_02 => X"342800003c01100127bd00108fb400048fbf00000100f809342800883c010040",
    INIT_03 => X"afa90008afa80004afb000027bdf08fa900088fa8000403e00008ad140000",
    INIT_04 => X"27bdf08fa9000c27bd000c8fa800048fbf00000100f809342800ec3c010040",
    ...
    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.

² Na realidade, BRAMs são blocos de 18Kbits, pois 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.

³ 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).

A inicialização da memória de instruções com o programa a executar é feita usando parâmetros genéricos 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 (ou seja, 4 bytes) de memória. Assim, 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 BRAM (de 16.384 bits, 64 linhas×256 bits).

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

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 BRAMs 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 ($2.048 \times 32 = 65.536$ 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. GERAM-SE DOIS ARQUIVOS. Exemplos de nomes para estes são: *prog.txt* e *data.txt*.

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 do material de apoio:

[0x00400020]	0x3c011000	<p>Uma linha de INIT contém 32 bytes, ou 8 palavras de 32 bits, ordenadas do endereço maior (à esquerda) para o endereço menor (à direita). Assim, estas 8 linhas de código geram a seguinte linha de INIT:</p> <p>8e520000343200083c0110018e310000343100043c011001343d08003c011000</p>
[0x00400024]	0x343d0800	
[0x00400028]	0x3c011001	
[0x0040002c]	0x34310004	
[0x00400030]	0x8e310000	
[0x00400034]	0x3c011001	
[0x00400038]	0x34320008	
[0x0040003c]	0x8e520000	
[0x00400040]	0x3c011001	
[0x00400044]	0x3433000c	
[0x00400048]	0x8e730000	
[0x0040004c]	0x27bdf000	
[0x00400050]	0xafbf0000	
[0x00400054]	0xafb10004	
[0x00400058]	0xafb20008	
[0x0040005c]	0xafb3000c	

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

1. **le_mars.c**– código fonte C para converter os dois programas de *dump* para o arquivo *memory.vhd*. É tarefa do aluno gerar o executável. Por exemplo, em um computador com um compilador da linguagem C da GNU instalado, um comando útil é em um shell de comandos digitar “gcc -Wall -o le_mars le_mars.c”. O exemplo é baseado em ambientes Linux, mas pode ser definido de forma similar em ambientes Windows e/ou MacOs. Para quem prefere usar um ambiente dotado de interface gráfica, pode-se instalar e usar sistemas gratuitos (*open-source*) de desenvolvimento de programas escritos em C como o Code::Blocks (<https://www.codeblocks.org/>). Depois de compilar o programa e gerar seu código executável, pode-se usar em um ambiente Linux o seguinte comando (em um shell de comandos): *./le_mars prog.txt data.txt* (com o

comando `./le_mars` sem parâmetro nenhum obtém-se um texto de ajuda). A execução da primeira linha produz como saída um arquivo de nome `memory.vhd`, que possui como conteúdo um texto válido em VHDL que implementa as memórias de instrução e dados, pré-carregadas com o código objeto do programa e seus dados.

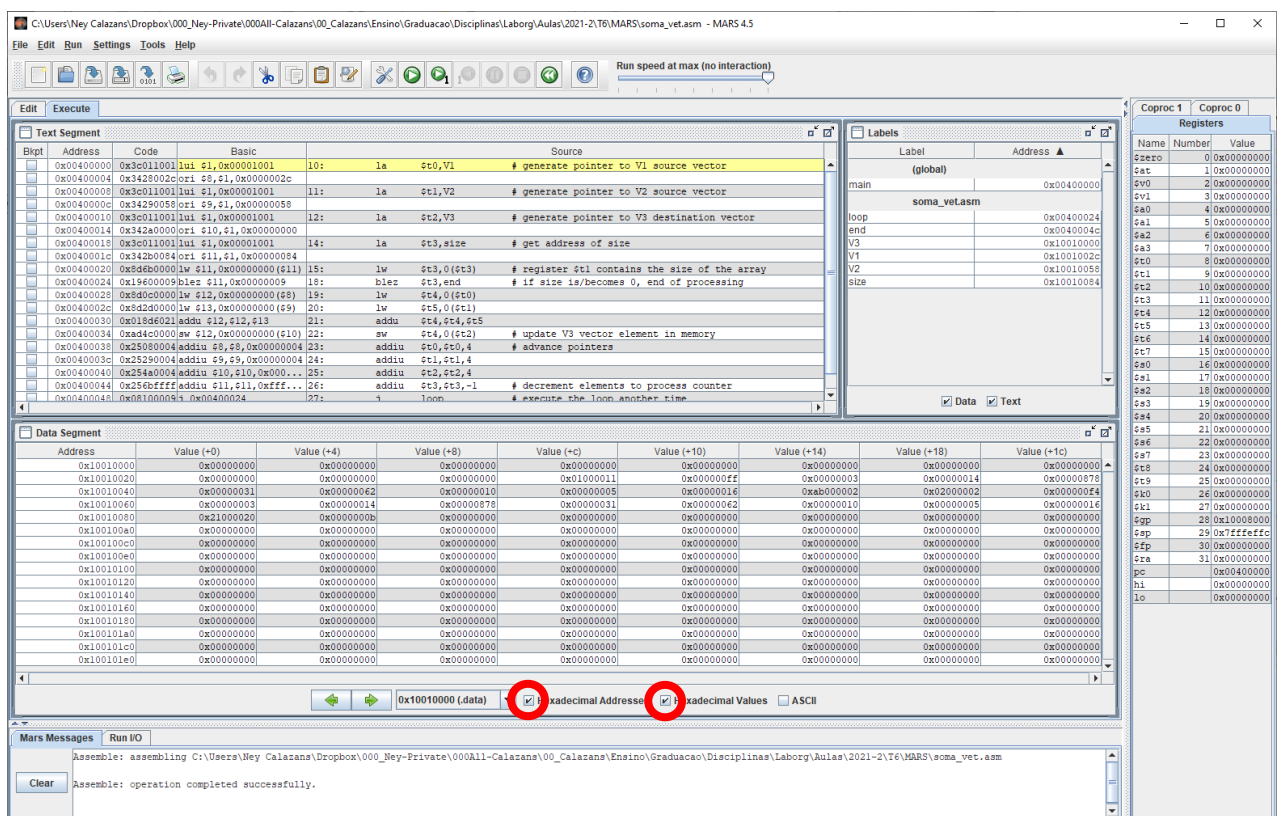
2. **soma_vet.asm** – arquivo fonte com um programa exemplo em linguagem de montagem do MIPS.

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 MIPS_S_withBRAMs executando tal programa.

1. Abra o arquivo **soma_vet.asm** no simulador MARS, monte o código objeto (tecla **F3** dentro do MARS). 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 **prog.txt** e **data.txt**, usando o formato de *dump Text/Data Segment Window*.

Atenção: Para que o *dump* de memória do segmento de dados seja produzido na forma esperada pelo software que vai tratá-lo, é necessário que endereços e dados estejam mostrados na tela em hexadecimal, veja os boxes marcados na Figura abaixo:



2. Execute o programa **le_mars** no diretório onde foram gravados os arquivos de *dump*. A execução do **le_mars** gera o arquivo VHDL **memory.vhd**, contendo o código objeto mapeado nas memórias de instruções e de dados. Esta execução deve ser executada em uma janela textual de comandos, seja no Windows, seja no Linux.
3. Crie um projeto no ISIM (o simulador do ISE) adicionando os seguintes arquivos:
 - i. **memory.vhd** (gerado no passo 2 acima),
 - ii. **MIPS_S_withBRAMs.vhd** (presente no diretório VHDL), e

- iii. **mips_sem_perif_tb.vhd** (presente no diretório VHDL).

Observação: Caso a biblioteca UNISIM não esteja disponível, contacte o professor para resolver o problema. Isto não deve ocorrer em geral.

4. Realize a simulação VHDL do sistema criado por um período de $5.15\mu s$ (usa-se um clock de 100Mhz). Isto pode ser feito no VIVADO lançando a simulação do testbench `mips_sem_perif_tb.vhd` e executando o seguinte comando na Tcl Console da simulação: `restart; run 5.15us`. Em uma janela do tipo forma de onda inserir os sinais conforme mostrado na Figura 5 (Pode-se carregar o arquivo que define esta disposição de formas de ondas, `CPU_tb_behav.wcfg`, disponível no material de apoio ao trabalho). Note que se trata de uma simulação completa do programa (*soma_vet.asm*) que soma de dois vetores (V1 e V2) de 11 inteiros e escreve o resultado em um terceiro vetor de mesmo tamanho (V3). Algumas das informações perceptíveis na Figura são:

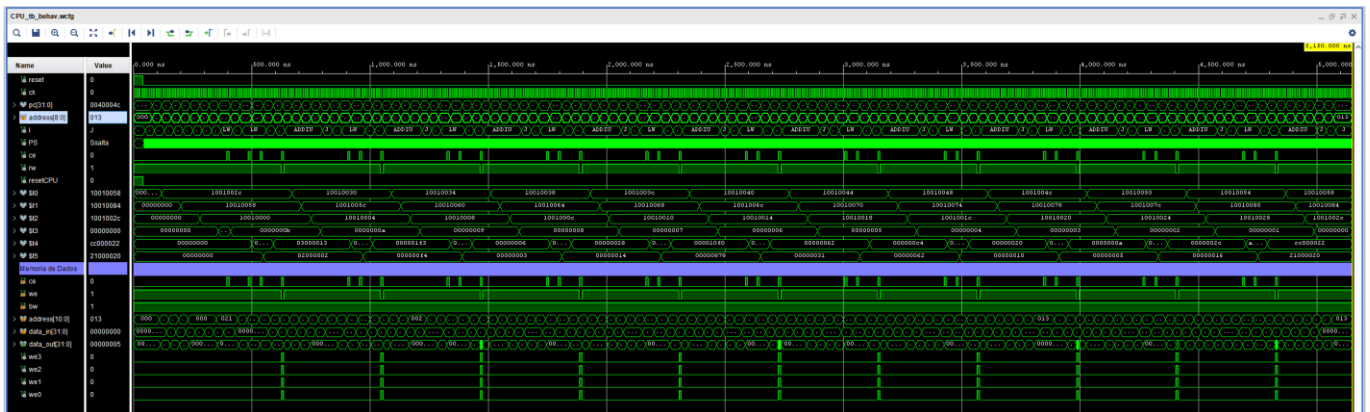


Figura 5 – Simulação completa da execução do programa *soma_vet.asm* no processador MIPS_S_withBRAMs.

- O tempo de simulação do programa completo (carca de 5,15 μ s), assumindo o uso de um clock de 100MHz;
- 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;
- Os momentos de escrita na memória, quando os sinais uins.ce='1' e uins.rw='0'. Isto ocorre exatamente 11 vezes na Figura (os vetores somados têm 11 elementos).
- O *dump* de memória realizado pelo testbench após o instante 5,15 μ s, quando o processador é ressetado. Cuidado com o testbench! Caso o programa demore mais de 15 μ s para executar o momento em que o sinal sel_CPU sobe (para gerar reset no processador e permitir o *dump* de memória) deve ser alterado. Notem que, após o fim do programa, entra-se em um laço eterno de salto para a mesma linha, usando uma instrução “end: j end”.

Apêndice

CONJUNTO DE EXERCÍCIOS PROPOSTOS 1 (para gerar exemplos de programas a executar via simulação no ambiente acima descrito):

1. Subtrair uma constante a um vetor armazenado a partir da posição de memória cujo rótulo é *end1*. O número de elementos está armazenado na posição de memória cujo rótulo é *end2*.
2. Fazer um algoritmo que lê um vetor, calcula o número de elementos pares e ímpares e informa qual é o maior par e qual é o maior ímpar.
3. 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: $\text{valor}(\text{inicio1}) < \text{valor}(\text{fim1})$, $\text{valor}(\text{inicio2}) < \text{valor}(\text{fim2})$, $(\text{fim1} - \text{inicio1}) = (\text{fim2} - \text{inicio2}) = (\text{tamanho do vetor} - 1)$.

4. Contar o número de posições de memória com conteúdo igual a uma constante, armazenada em uma posição de memória cujo rótulo é CTE, no vetor armazenado entre dois endereços marcados com os rótulos *inicio* e *fim*.
5. Dados dois inteiros, A e B, armazenados, respectivamente, nas posições de memória cujos rótulos são *n1* e *n2*, armazenar na posição de memória com rótulo *max* o $\max(A,B)$ (valor máximo entre A e B) e na posição de memória com rótulo *min* o $\min(A,B)$ (definido de forma similar ao *max*), levar em consideração números positivos e negativos.
6. Descobrir se um número *n* é múltiplo de 4.
7. Descobrir se um número *n* é múltiplo de um número *m* qualquer. Pressupor que *n* possa ser um número positivo ou negativo.
8. Faça um algoritmo que calcula a divisão de dois números de 8 bits (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*.

CONJUNTO DE EXERCÍCIOS PROPOSTOS 2:

9. 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.
10. Fazer um programa que gere os *n* primeiros números da sequência de Fibonacci, e armazene a sequência em endereços consecutivos a partir da posição de memória com rótulo *idx*.
11. A transmissão de dados binários é o que viabiliza a existência de tecnologias como a Internet. A transmissão de dados a longas distâncias é uma tarefa muito propensa a erros, devido a efeitos ambientais externos (raios, interferências eletromagnéticas nas linhas de transmissão devidas a equipamentos elétricos, raios cósmicos ou manchas solares que interferem nas transmissões de satélites etc.). Uma maneira de detectar erros de transmissão é acrescentar bits de controle ao dado transmitido, de forma que o receptor possa verificar a validade dos dados transmitidos. Um esquema simples de detecção de erros são as técnicas de *bit de paridade*. A idéia consiste em contar o número de bits de um determinado valor e acrescentar um bit na mensagem que diz se a contagem destes valores na mensagem é par ou ímpar. Assim, podem-se imaginar alguns tipos básicos de cálculo de paridade: paridade de 0s ou paridade de 1s, paridade par ou paridade ímpar, paridade ativa em 0 ou paridade ativa em 1. Implemente um programa que recebe *n*, o número de bits de uma mensagem; *msg*, a mensagem de tamanho *n*; e produz uma nova mensagem *msgp* com *n* + 1 bits, onde o bit mais significativo é a paridade da mensagem.
12. 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*.

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

13. Seja dado um ponto inicial (*x0*, *y0*) e uma sequência de *n* valores inteiros (o valor de *n* deve estar armazenado em uma posição de memória com rótulo *n*, e a sequência de valores deve estar armazenada como um vetor iniciando na posição de memória com rótulo *seq*). Implemente um algoritmo que desloque o ponto inicial alternadamente na horizontal e na vertical (iniciando com um deslocamento horizontal). Suponha que cada valor da sequência dá a magnitude do deslocamento seguinte. Considere que os deslocamentos são tais que o deslocamento atual e o imediatamente anterior circunscrevem um arco que avança no sentido anti-horário se o deslocamento atual for

negativo, e no sentido horário se o deslocamento atual for positivo. Note que o deslocamento inicial pode ser em qualquer sentido horizontal, pois não há deslocamento anterior. Documente sua solução para indicar a escolha feita pelo seu programa neste caso.

Exemplo: dados de entrada [-6, -6, -4, 2, 3, 4] (ver Figura 6). Resultado -5 para x e -4 para y.
Decisão: primeiro deslocamento para a esquerda se negativo, para a direita se positivo.

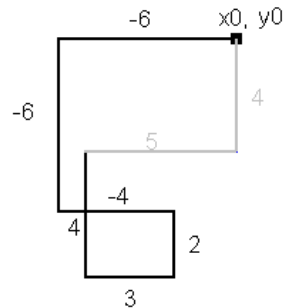


Figura 6 – Representação gráfica da entrada de dados [-6, -6, -4, 2, 3, 4].

14. 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.
15. Faça um programa para ordenar 20 valores hexadecimais que estão armazenados em 10 bytes. Sendo que dentro de cada byte, o *nibble* (conjunto de 4 bits) menos significativo deve conter o menor valor.
Exemplo: Vetor de entrada: 0x34, 0xFA, 0xBC, 0x77, 0x1C
Vetor ordenado: 0x13, 0x47, 0x7A, 0xBC, 0xCF
16. Escrever um algoritmo que calcule os primeiros *n* números primos e os armazene sequencialmente, a partir da posição de memória cujo rótulo é *nprimos*.