1 TUTORIAL DE SIMULAÇÃO (VHDL) DO PROCESSADOR MIPS Multiciclo com 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 MIPSmulti_with_BRAMs. Este processador implementa uma organização que pode executar uma boa parte das instruções da arquitetura MIPS R2000. Uma especificação detalhada do MIPSmulti_with_BRAMs 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. Contudo, o acesso desalinhado a byte com instruções **`lb`**, **`lbu`** e **`sb`** é possível. Como o que se pretende é 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 facilmente sintetizadas em FPGAs Xilinx, conforme explicado abaixo. Nos modelos de simulação de processadores vistos nas aulas teóricas, um testbench complexo permite carregar antes da simulação as memórias com um programa qualquer e seus dados. Aqui, se procede 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 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 MIPS_multiciclo_com_BRAMs.

1.1 O ambiente de simulação do processador MIPS_multiciclo_com_BRAMs

O ambiente de simulação/síntese a ser usado neste tutorial encapsula o processador MIPSmulti_with_BRAMs 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.

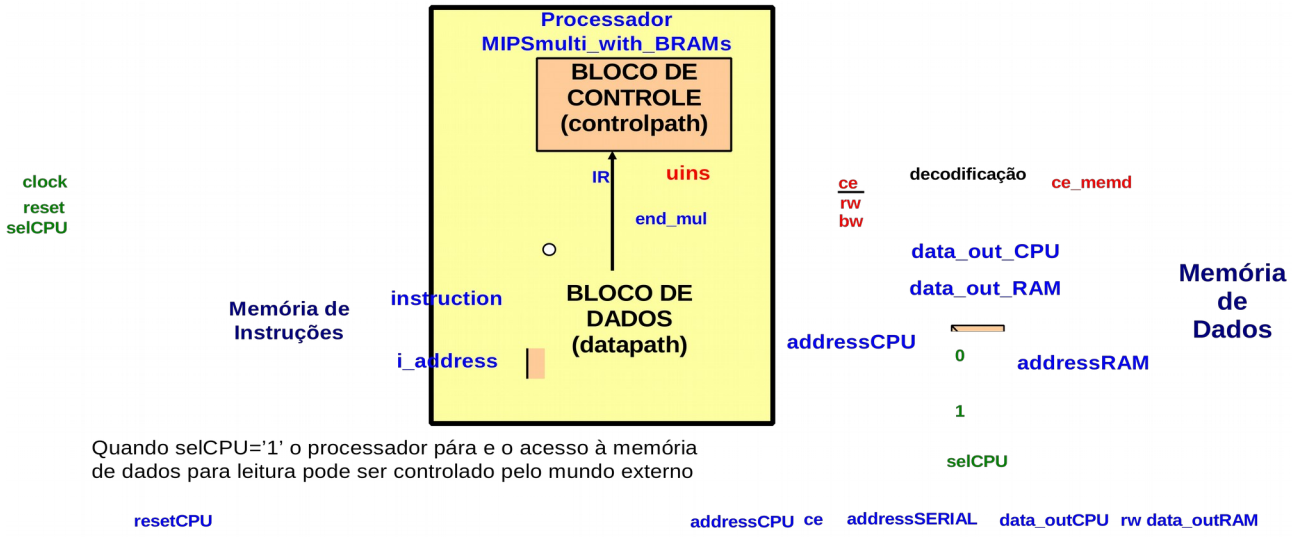

Figura 1 – Diagrama de blocos do ambiente contendo o processador MIPSmulti_with_BRAMs e suas memórias de instruções e dados. O processador MIPSmulti_with_BRAMs em si (retângulo amarelo e seus conteúdos) conecta-se às memórias de instruções e de dados. Vários sinais de interfaces dos blocos foram omitidos 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 MIPS_multiciclo_com_BRAMs, mais uma "lógica de cola" simples para permitir o controle da comunicação entre as diversas partes do ambiente entre si.

```
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;
```

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 MIPSmulti_with_BRAMs faz acesso.

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

[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.

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 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. Obviamente, estes valores podem ser aumentados sem grande 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_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"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 MIPS_multiciclo_com_BRAMs. Note que no

[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).

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).

1.2.2 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***. GERAM-SE DOIS ARQUIVOS – exemplo: *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:

1. **le_mars.c**– código fonte para converter os dois programas de *dump* para o arquivo *memory.vhd*. É tarefa do aluno gerar o executável. Por exemplo " gcc –Wall –o le_mars le_mars.c".
   Para uso no LINUX: *./le_mars prog.txt data.txt* (digitando apenas *./le_mars* tem-se um texto de ajuda).
2. **soma_vet.asm** – arquivo fonte com um programa exemplo em linguagem de montagem do MIPS.

```
3c011000
343d0800
3c011001
34310004
3e310000
3c011001
34320008
3e520000
3c011001
3433000c
3e730000
27bdfff0
afbf0000
afb10004
afb20008
afb3000c
```

```
Uma linha de INIT contém 32 bytes, ou 8 palavras de 32 bits
endereço maior (à esquerda) para o endereço menor (à direit
estas 8 linhas de código geram a seguinte linha de INIT:

8e520000 34320008 3c011001 8e310000 34310004 3c011001 343d(
```

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

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 MIPSmulti_with_BRAMs 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 ***prog.txt*** e ***data.txt***, usando o formato de *dump* ***Text/Data Segment Window***.

2. Executar 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.
3. Criar um projeto no ISIM (o simulador do ISE) adicionando a estes os arquivos:
   i. **memory.vhd** (recém gerado no passo anterior),
   ii. **mrstd.vhd** (presente no diretório MIPS_multiciclo_com_BRAMs), e
   iii. **mrstd_tb.vhd** (presente no diretório MIPS_multiciclo_com_BRAMs).

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. Este problema não deve ocorrer nos laboratórios da FACIN.

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 5. Note que se trata de uma simulação completa do programa (***soma_vet.asm***) de soma de dois vetores de 11 elementos. Algumas das informações perceptíveis na Figura são:

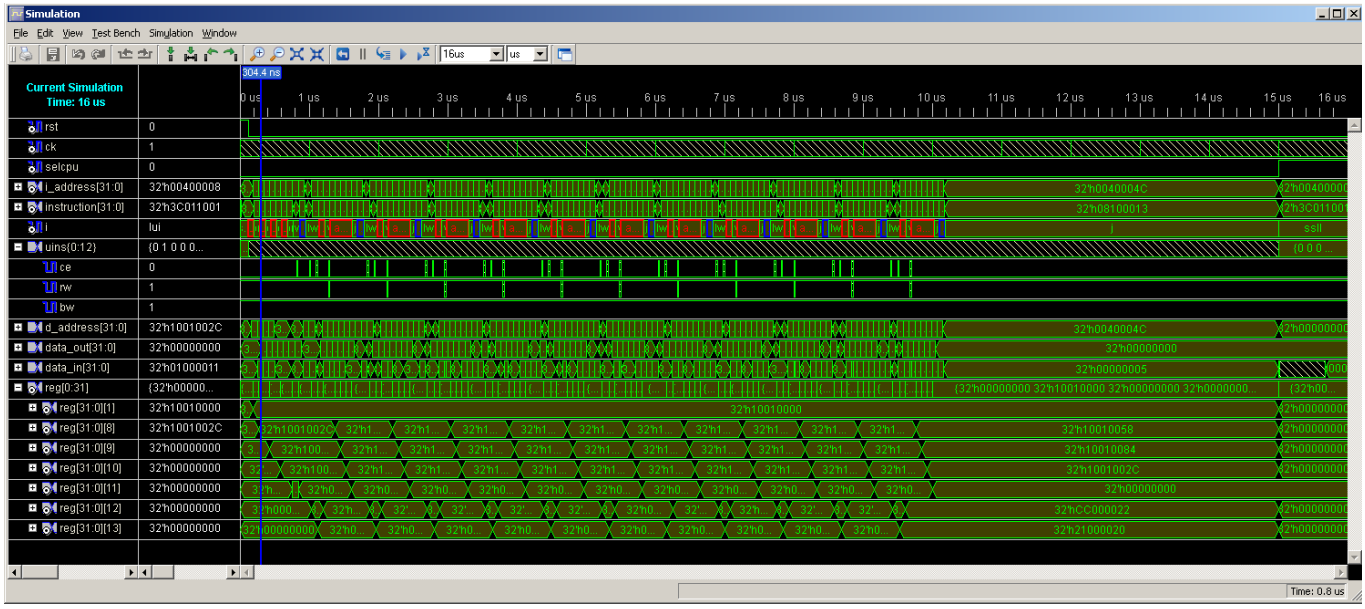

Figura 5 – Simulação completa da execução do programa *soma_vet.asm* no processador MIPS_multiciclo_com_BRAMs.

A. O tempo de simulação do programa completo (10.37 µs, pouco mais de 10 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.
D. O *dump* de memória realizado pelo testbench após o instante 15µs, quando o processador é ressetado. Cuidado com o testbench! Caso o programa demore mais de 15µs 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. Aqui, após o fim do programa, entra-se em um laço eterno de salto para a linha do salto.

3 A Fazer e a Entregar

1. Cada grupo de alunos deverá realizar dois exercícios de programação em linguagem de montagem

do MIPS, usando o subconjunto de instruções ao qual o MIPSmulti_with_BRAMs dá suporte. Durante a aula, o professor atribuirá a cada grupo de dois alunos um de cada conjunto da lista definida na Tabela 1 do Apêndice deste documento. Para cada exercício implementado devem ser **entregues** o **código fonte** em **linguagem de montagem**, juntamente com uma documentação escrita em **linguagem C** ou um pseudo-código de C ou outra linguagem procedural. O objetivo desta última parte serve apenas documentar a estrutura do programa em linguagem de montagem.

2. Após implementar cada aplicação proposta, o grupo deve executar procedimento similar ao descrito na Seção 2 deste documento (Tarefas Acessórias Iniciais) para gerar o código objeto de cada aplicação, bem como o arquivo VHDL **memory.vhd** pertinente. Então, o grupo deve simular a execução completa do programa no simulador ISE, recompilando o ambiente de simulação com o arquivo gerado. O grupo **deve entregar**, para cada aplicação implementada:
    a. O projeto ISE da aplicação, contendo pelo menos uma forma de onda de uma execução completa do programa;
    b. Um documento contendo a explicação da simulação, mostrando o estado inicial de memórias e variáveis mais importantes do programa e o estado final das mesmas memórias e variáveis. Estados intermediários da execução podem ser mostrados opcionalmente. O uso de ilustrações com partes da forma de onda de simulação para mostrar o que aconteceu durante a execução simulada do programa é encorajado.

Apêndice

Tabela 1 - Especificação de grupos de programas.

		Grupos											
		G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12
Conjunto	1	1	2	3	4	5	6	7	8	2	4	6	8
	2	9	10	11	12	13	14	15	16	9	10	12	14

CONJUNTO DE EXERCÍCIOS PROPOSTOS 1:

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: valor(*inicio1*) < valor(*fim1*), valor(*inicio2*) < valor(*fim2*), (*fim1* - *inicio1*) = (*fim2* - *inicio2*) = (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 Fibonnacci, e armazene a sequências 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.

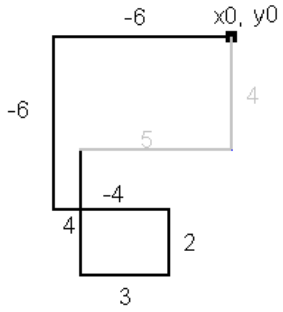

Figura 6 – Representação gráfica da entrada de dados [-6, -6, -4, 2, 3, 4].

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*.