

Laboratório 1 - Assembly MIPS –

Objetivos:

- Familiarizar o aluno com o Simulador/Montador MARS;
- Desenvolver a capacidade de codificação de algoritmos em linguagem Assembly MIPS;
- Desenvolver a capacidade de análise de desempenho de algoritmos em Assembly;

(1.0) 1) Simulador/Montador MARS

Instale em sua máquina o simulador/montador MARS v.4.5 Custom 7 disponível no Moodle.

(0.0) 1.1) Dado o programa `sort.s` e o vetor: $V[10]=\{5,8,3,4,7,6,8,0,1,9\}$, ordená-lo em ordem crescente e contar o número de instruções por tipo, por estatística e o número total exigido pelo algoritmo. Qual o tamanho em bytes do código executável?

(1.0) 1.2) Considere a execução deste algoritmo em um processador MIPS com frequência de *clock* de 50MHz que necessita 1 ciclo de *clock* para a execução de cada instrução ($CPI=1$). Para os vetores de entrada de n elementos já ordenados $v_o[n]=\{1,2,3,4,...,n\}$ e ordenados inversamente $v_i[n]=\{n, n-1, n-2,...,2,1\}$, obtenha o número de instruções, calcule o tempo de execução para $n=\{1,2,3,4,5,6,7,8,9,10,20,30,40,50,60,70,80,90,100\}$ e plote esses dados em um mesmo gráfico $n \times t_{exec}$. Comente os resultados obtidos.

(2.0) 2) Compilador GCC

Instale na sua máquina o *cross compiler* MIPS GCC disponível no Moodle.

Forma de utilização: **mips-sde-elf-gcc -S teste.c** #diretiva -S para gerar o arquivo Assembly teste.s

Inicialmente, teste com programas triviais em C para entender a convenção utilizada para a geração do código Assembly.

(0.5) 2.1) Dado o programa `sortc.c`, compile-o e comente o código em Assembly obtido indicando a função de cada uma das diretivas do montador usadas no código Assembly (`.file` `.section` `.mdebug` `.previous` `.nan` `.gnu_attribute` `.globl` `.data` `.align` `.type` `.size` `.word` `.rdata` `.ascii` `.text` `.ent` `.frame` `.mask` `.fmask` `.set`).

(0.5) 2.2) Indique as modificações necessárias no código Assembly gerado pelo gcc para poder ser executado no Mars.

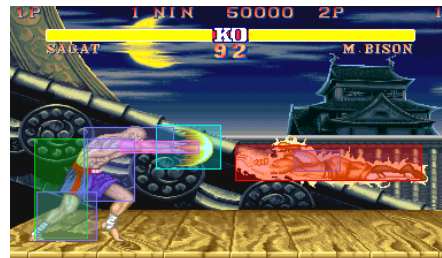
(1.0) 2.3) Compile novamente o programa `sortc.c` e com a ajuda do Mars compare o número de instruções executadas e o tamanho em bytes dos códigos obtidos com os dados do item 1.1) para cada diretiva de otimização da compilação `{-O0, -O1, -O2, -O3, -Os}`.

(2.0) 3) Sprites:

Hoje em dia existem diversas bibliotecas nas mais diversas linguagens de programação de alto nível que permitem realizar, de maneira fácil, o desenho de um sprite em uma determinada posição da tela e detectar a colisão do mesmo com o cenário existente e mesmo outros sprites presentes através de *hitboxes*. Crie um procedimento (ou conjunto de procedimentos) que permita desenhar sprites na tela gráfica do Mars.

Considere que um Sprite é definido em um endereço (32 bits) na memória de dados que contém a estrutura:

- Tamanho h em pixels (1 byte)
- Tamanho w em pixels (1 byte)
- Bloco de $h*w$ bytes contendo um retângulo que define as cores de cada pixel da imagem do Sprite
- lista de NumH vértices, tamanhos e tipos (todos de 1 byte), que definem as *hitboxes* do sprite



(2.0) 3.1) `int PrintSprite (int EnderecoSprite, int x, int y);`

Imprime o sprite presente no endereço EnderecoSprite na posição (x,y) da tela e retorna 0 se não houver colisão ou o 'tipo' da *hitbox* que apresentou colisão.

Grave vídeos demonstrativos e coloque os links no relatório.

A tela gráfica do Mars, acessível pelo BitMap Display Tool, possui resolução 320x240 e 8 bits/pixels para a codificação das cores. O pixel na posição (x,y) pode ser plotado através do comando `sb $t0,0($t1)`:

$\$t1 = 0xFF000000 + 320*y + x$ é o endereço do pixel na memória de vídeo VGA

$\$t0 = 1$ byte que define a cor no formato {BBGGRR}

Obs.: A cor 0xC7 (ou R=255 G=0 B=255) é considerada transparente pelo Mips no FPGA e pelo Mars Custom 7.

(3.0) 4) Cálculo das raízes da equação de segundo grau:

Dada a equação de segundo grau: $a.x^2 + b.x + c = 0$

(1.0) 4.1) Escreva um procedimento int baskara(float a, float b, float c) que retorne 1 caso as raízes sejam reais e 2 caso as raízes sejam complexas conjugadas, e coloque na pilha os valores das raízes.

(0.5) 4.2) Escreva um procedimento void show(int t) que receba o tipo (t=1 raízes reais, t=2 raízes complexas), retire as raízes da pilha e as apresente na tela, conforme os modelos abaixo:

Para raízes reais:

R(1)=1234.0000

R(2)=5678.0000

Para raízes complexas:

R(1)=1234.0000 + 5678.0000 i

R(2)=1234.0000 – 5678.0000 i

(0.5) 4.3) Escreva um programa main que leia do teclado os valores float de a, b e c, execute as rotinas baskara e show e volte a ler outros valores.

(1.0) 4.4) Escreva as saídas obtidas para os seguintes polinômios [a, b, c] e, considerando um processador MIPS de 1GHz, onde instruções tipo-J são executadas em 1 ciclo, tipo-R em 2 ciclos, tipo-I em 3 ciclos, tipo-FR e FI em 4 ciclos de clock, calcule os tempos de execução da sua rotina baskara (otimizada).

a) [1, 0, -9.86960440]

b) [1, 0, 0]

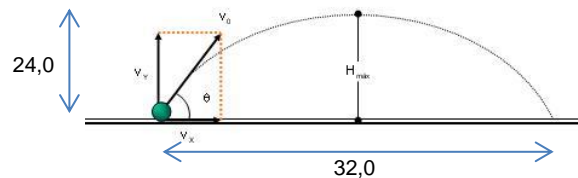
c) [1, 99, 2459]

d) [1, -2468, 33762440]

e) [0, 10, 100]

(2.0) 5) Trajetórias:

Uma constante na área de simulação computacional é o cálculo de trajetória de pontos/objetos considerando as leis da física. Dada a aceleração da gravidade 9.8 m/s^2 , que o ar não tenha resistência (sem atrito) e que a tela possui um tamanho de 32,0 x 24,0 metros conforme o desenho abaixo:



(1.5) 5.1) Realize a simulação do lançamento de uma bola de canhão de tamanho 1 pixel, desenhando a sua trajetória em tempo real com base na velocidade inicial \vec{v}_0 (com componentes vertical V_{0y} , horizontal V_{0x} e ângulo θ).

Leia os valores necessários do teclado e faça uma animação gráfica do disparo.

Considere que a bola possa sair da região visível da tela.

Grave vídeos demonstrativos com diversos casos e coloque os links no relatório.

(0.5) 5.2) Indique os valores de θ e $|\vec{v}_0|$ para que um lançamento atinja os limites superior e lateral direito da tela.

Obs.: Os pulos do Dhalsim não seguem a física....