Laboratório 1
Assembly MIPS

Giovanni M Guidini, 16/0122660
Vitor F Dullens, 16/0148260
André Cássio B Souza 16/0111943
Henrique P Léo,16/0124379
Thiago V Machado, 16/0146682
Gabriel B Vieira, 16/0120811

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CiC 116394 - Organização e Arquitetura de Computadores - Turma A

1. MARS

1.1. Sort.s:

Estatísticas somente da função sort (implementação de bubble sort) em Assembly MIPS. Retiradas com auxílio das ferramentas *Instruction Counter* e *Instruction Statistics* do MARS.

Baseado em um vetor não ordenado de tamanho n = 10.

- 842 instructions : I - 58%; R - 32%; J - 9%.
- 842 instructions : ALU - 38% ; Jump - 12% ; Branch - 11% ; Memory - 22% ; Other - 17%.
- 1147 bytes de memória.

1.2. Instruções; Tempo de Execução; Gráfico

Ainda no programa sort.s, contamos o número de instruções executadas e calculamos o tempo de execução de cada teste. Os gráficos 1 e 2 resumem estas informações.

Para o tempo de execução foi considerada uma máquina com frequência de clock de 50MHz e CPI médio igual à 1. Os testes foram executados com valores crescentes para o tamanho do vetor a ser ordenado, com este tamanho variando de 1 até 100 números. Aém disso testamos o melhor caso (linha verde) e o pior caso (linha vermelha), para conhecer os extremos do algoritmo.

- Número de instruções x Tamanho do Vetor:
  Note como existe uma diferença tremenda no crescimento dos casos de teste, com o pior caso chegando muito próximo à um milhão de instruções.

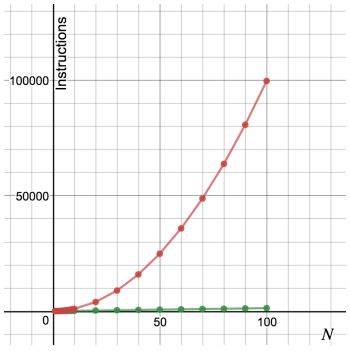

Figure 1. Gráfico Instruções x Tam. Vetor

- Tempo de execução x Tamanho do Vetor:
  Aqui o gráfico ficou bastante parecido com o primeiro, conforme esperado. A medida de tempo está em microssegundos, mas nota-se que o pior caso com tamanho 100 chega a precisar 2s para ser executado.

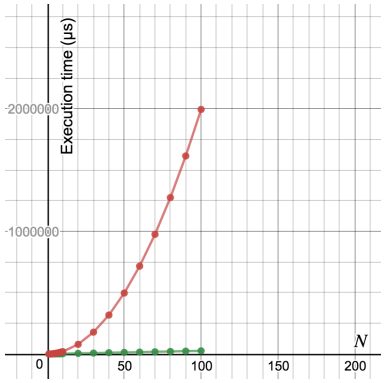

Figure 2. Gráfico Texec x Tam. Vetor

Um aspecto bastante interessante desta questão foi ”concretizar” as noções de análise de algoritmos que desenvolvemos em ED e APC. Por exemplo, sabemos que o BubbleSort possui complexidade de tempo O(n^2), e o gráfico nos mostra claramente que, no pior caso, a complexidade de tempo do algoritmo é, de fato, quadrática.

O valor exato de cada ponto do gráfico, bem como as contas realizadas para obtenção do tempo de execução pode ser verificado através do link `https://pastebin.com/GvK0iRYr` ou nos arquivos na pasta 1 anexados ao relatório.

2. Compilador GCC

2.1.

Comentarios sobre as diretivas em ’sortc.s’, anexado junto ao relatorio, ou em ’directives.txt’, também anexado.

Percebemos que cross compiler faz uso intensivo de diretivas, muitas das quais o MARS não reconhece e/ou ignora. Isso foi realmente um problema para ser resolvido de forma que os arquivos gerados pudessem ser executados no MARS.

O maior problema foi a diretiva .set, que o MARS reconhece como diretiva, mas decide ignorar. Percebemos que o cross compiler faz uso intenso dessa diretiva, por exemplo, para criar macros.

2.2.

Modificações realizadas indicadas em ’sortWorking.s’, anexado junto ao relatório, ou em ’modifications.txt’, também anexado.

Em linhas gerais foi possível perceber que o cross compiler mantém a estrutura do código C inalterada, como a ordem em que as funções aparecem no código, por exemplo. Cada pedaço de código tinham inúmeras diretivas e especificações que, num primeiro momento, atrapalharam o entendimento do código.

2.3.

Após realizarmos os procedimentos propostos, comparamos com o sort.s realizado no item 1.1, nele o programa realizava a mesma tarefa tarefa que o novo sortc.s (derivado da compilação com o cross compiler MIPS GCC). O sort.s se mostrou mais eficiente, possuindo menos intruções e ocupando menos memoria que o outro, mesmo ambos realizando o mesmo procedimento.

Feito essa comparação, compilamos novamente, porem utilizando as diretivas de otimização da compilação -O(0,1,2,3,s). Abaixo está o resultado obtido a partir da nova compilação:

- sort.s (item 1.1)→ 1147 bytes
- sortWorking.s (sortc working on MARS)→ 4126 bytes
- sortc.s → 3614 bytes
- sort-O0 → 3318 bytes
- sort-O1 → 3358 bytes
- sort-O2 → 3182 bytes
- sort-O3 → 3168 bytes
- sort-Os → 2992 bytes

Claramente uma diferença considerável. Pelo menos 300 bytes de programa puderam ser otimizados com as diretivas. Isso faz uma diferença razoável na performance do programa, em especial se tiver que ser executado repetidas vezes. Atenção especial para a diretiva -Os cuja objetivo é criar o menor arquivo possível, e de fato é consideravelmente menor que todos os outros. Ainda sim, o ”mais otimizado” em relação à tempo de execução provavelmente seria o da diretiva -O3, que permite toda e qualquer otimização.

3. Sprites

O código completo e devidamente comentado se encontra nos arquivos em anexo na pasta 3. Tenha em mente que o código disponibilizado é uma versão de teste, então muitas coisas que foram *hardcoded* (por exemplo, o tamanho dos sprites e o nome do arquivo, serão gerados dinamicamente no código do projeto final. Além disso, das 380 linhas do arquivo, pouco mais de 250 efetivamente mostram o sprite na tela e verificam colisões.

Um resumo do funcionamento da função pode ser visto na figura 3. As partes mais importantes da função e as decisões de design serão descutidas a seguir, mas o esudo do código será deixado ao leitor que tenha interesse em abri-lo nos anexos.

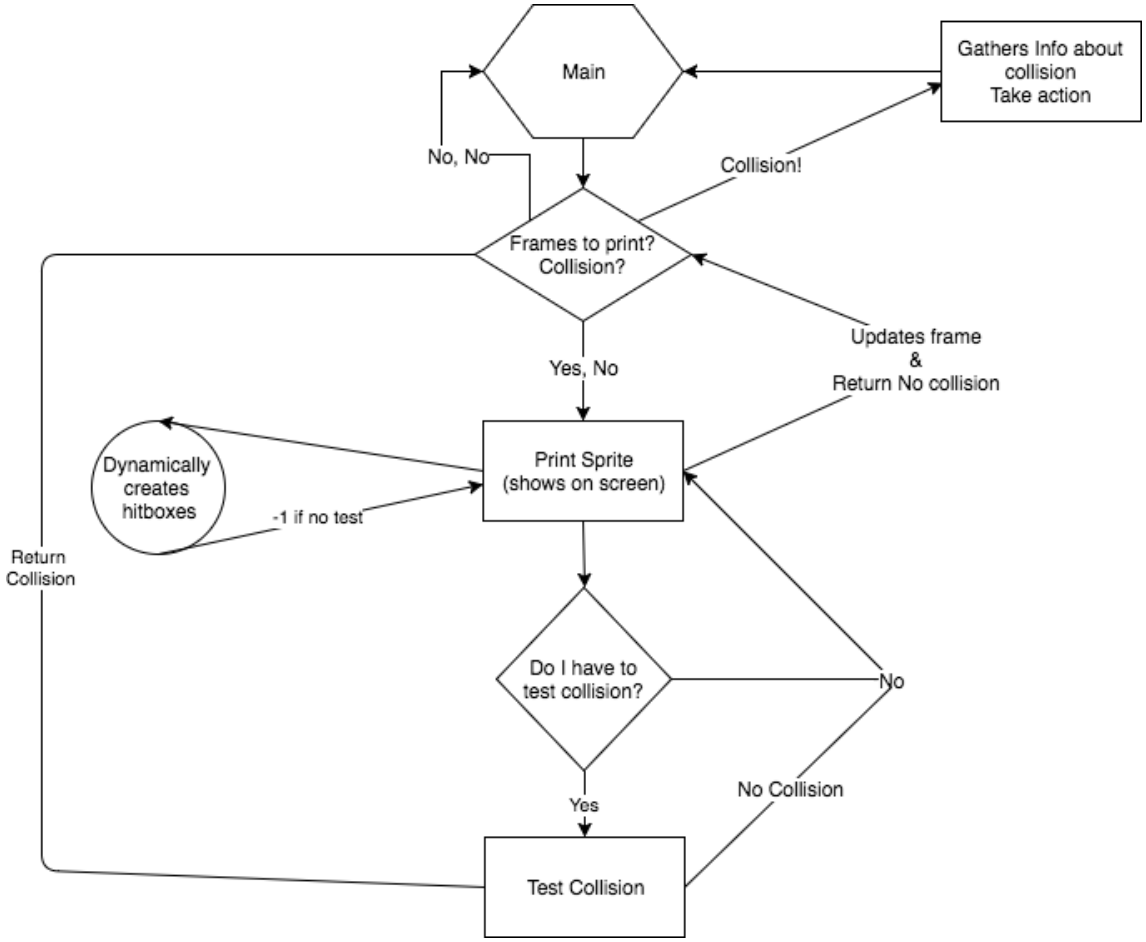

Figure 3. Flowchart do funcionamento da função printSprite

A função printSprite trabalha com uma spritesheet que compreenda um único movimento completo de um personagem. Por exemplo, na ação de socar o arquivo contém 3 ou 4 frames que serão mostrados na sequência correta. Esta sequência está definida num vetor de abel `frameSequence` na região `.data` da main. Além disso printSprite recebe um intent (ação), posição x e y, e as diretivas do frame - largura e altura de cada frame, número de frames no arquivo, e qual a posição atual na sequência de frames a ser mostrada.

O arquivo com a imagem propriamente dita é carregado numa posição da memória RAM com a label buffer. Ele só precisa ser carregado uma vez antes da primeira chamada da função printSprite, contendo todos os frames. Decidimos fazer desta forma para poder printar vários frames em sequência sem precisar ficar abrindo vários arquivos, e acessando mais mémoria RAM do que HD, acelerando bastatante o processo de mostrar uma imagem na tela.

Com todas estas informações mostrar o sprite na tela é bastante simples. Os cálculos do tamanho do arquivo é realizado, o início do frame correto encontrado e as informações movidas do buffer para a memória VGA.

Após mostrar a imagem na tela a função printSprite chama outra função (interna, ou seja, só será chamada pela função print sprite), para criar as hitboxes dinamicamente.

Como toda alteração no sprite altera suas hitboxes elas são refeitas odas as vezes.

Existem 2 vetores de hitboxes, uma para o player 1 e outra para o player 2. A função de criar hitboxes sabe quem está se mexendo atualmente pelo intent passado à função printSprite: se for positivo é o player 1, se for negativo é o player 2. Feito isso ele pode carregar a label correta para inserir as informações da hitbox.

Dado o intent, fizemos uma espécie de switch case na função de criar hitboxes. Cada intent gera uma quantidade pré-determinada de hitboxes com tamanho também pré-determinado e baseado na posição atual e tamanho do sprite que acabou de ser desenhado. Para o teste implementamos os movimentos *passive* - ficar parado - e *punchM* - soco no meio.

As informações geradas de hitboxes são armazenadas nos devidos vetores e a função retorna para printSprite indicando se é necessário verificar colisão ou não. Somente ações ”ativas” (como os ataques e especiais) precisam ser verificadas. Isso acelera o programa, já que não tem que verificar colisões todo o tempo.

A função de verificação de colisões (`colisionTest`). Ela, em verdade, verifica se não houve colisão, através de testes relacionados com as extremidades superior esquerda e inferior direita dos retângulos que formam as hitboxes.

Havendo um hit ela retorna os tipos de hitboxes que se chocaram. O retorno (na main) consegue inferir quem acertou quem e tomar as devidas ações. Isso para o loop de renderização da sequência de frames que estava em andamento.

Em suma, até agora, a função é capaz de renderizar qualquer sequência de frames em arquivos de imagem que não excedam o tamanho alocado em buffer, gerar dinamicamente hitboxes apropriadas para ações passivas e de soco, e corretamente identificar colisões.

Veja no apêndice A fotos do desenvolvimento da função e comentários quanto aos estágios de desenvolvimento. Nos arquivos anexados temos um gif também mostrando o funcionamento do programa.

4. Raizes equação 2 grau

Os procedimentos/codigo fonte do programa pedido segue no anexo devidamente comentado;

Executando a rotina baskara obtivemos os seguintes resultados, a partir das entradas abaixo:

1. 1, 0, -9.86960440 → R1 = 4.791542E-21; R2 = -4.791542E-21
2. 1, 0, 0 → R1 = 0.0; R2 = -0.0
3. 1, 99, 2459 → R1 = -49.5 + (-49.5) i; R2 = -49.5 - (-49.5) i
4. 1, -2468, 33762440 → R1 = 1234.0 + (1234.0) i; R2 = 1234.0 - (1234.0) i
5. 0, 10, 100 → R1 = Infinity; R2 = Infinity

Com os testes realizados acima, calculamos o tempo de execução de cada um dos itens, respectivamente:

1. 1, 0, -9.86960440 → 338 ns
2. 1, 0, 0 → 338 ns

3. 1, 99, 2459 → 392 ns
4. 1, -2468, 33762440 → 392 ns
5. 0, 10, 100 → 343 ns

5. Trajetória

5.1. Código

Usamos as fórmulas de lançamento oblíquo e manipulação algébrica chegamos às seguintes equações

$$x = V_x t \Rightarrow t = x/V_x$$
$$y = V_y t + gt^2/2 \Rightarrow$$
$$y = V_y(x/V_x) - g/2 * (x/V_x)^2$$

Com essas equeções foi possível calcular um valor y para cada um dos possíveis 320 x, criando uma simulação do lançamento baseado nos componentes V_x e V_y.

Nos arquivos anexados estão alguns gifs com o funcionamento do programa, bem como o código em si.

Link para o video demonstrativo do exercicio `https://youtu.be/5DiJ23gcBpU`.

5.2.

Empiricamente, descobrimos que os valores $V_y = 21.7$ e $V_x = 7.206$ geram um lançamento que atinge a altura máxima (y = 240) e a largura máxima ($x \approx 320$)

A. Print Sprite Tests

A figura 4 mostra a primeira tentativa de renderização. Claramente não foi bem sucedida. Faltava acertar os critérios de parada de cada linha do arquivo.

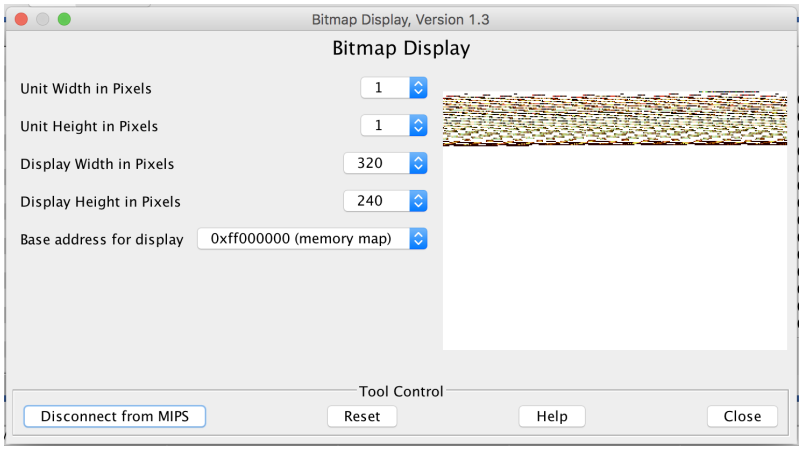

Figure 4. Primeira tentativa de renderização de sprites

Depois de acertar os critérios de parada, o problema que encontramos foi muito mais com os arquivos das imagens do que com o algoritmo de mostrá-las na tela. A

figura 5 exemplifica isso: note que a imagem é renderizada como um retângulo, mas está ”shiftada”, porque o arquivo não teve as especificações corretas passadas.

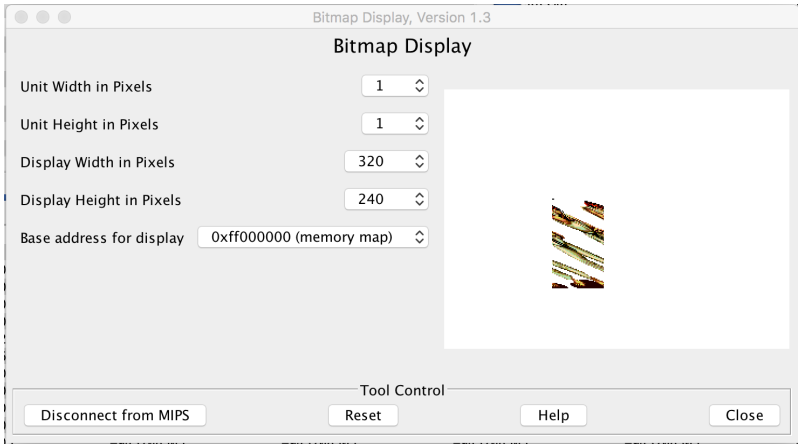

Figure 5. Depuração de renderização de sprites

Neste estágio já estávamos testando as colisões. Repare que já era possível printar diferentes personagens em diferentes posiçÕes. Eles se mexiam também. O arquivo que não foi printado corretamente tinha um erro nas especificações de tamanho, causando este erro, mas as colisões foram corretamente detectadas.

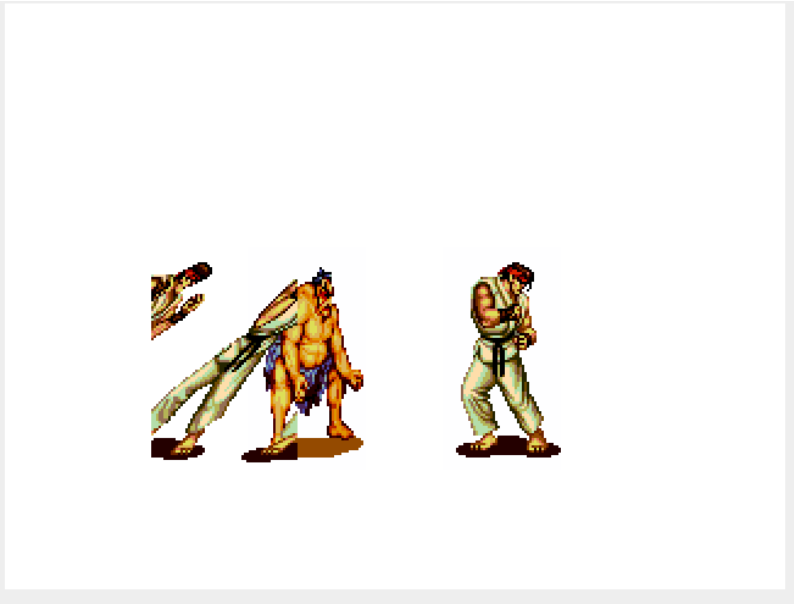

Figure 6. Testando print de múltiplos sprites simultaneamente