

Trabalho Prático 5

Parte I - Introdução ao *Assembly* do MIPS

Objetivos


- Introdução ao simulador MARS
- Tradução dum programa em C para *Assembly*
- Execução e *Debug* dum programa *Assembly*

Guião

1. Panorâmica geral do simulador MARS

O simulador MARS (MIPS Asembler and Runtime Simulator) é um *software* gratuito que permite editar e executar programas em *Assembly*. Essencialmente, é composto por um editor sensível à sintaxe (*syntax highlight*) e por um simulador que permite a execução e *debug*. Apresentamos abaixo uma breve descrição destas duas componentes.

1.1 Janela de edição

A Figura 1 apresenta o aspeto da janela do editor, onde se podem destacar as secções de dados (*.data*) e de código (*.text*) e os comentários (verde). Na secção de código, podemos identificar claramente as mnemónicas *Assembly* (azul), os registos (vermelho) e os *labels* (preto), graças à utilização de diferentes cores. O programa pode ser *assemblado* carregando no botão . Caso o programa não contenha erros de sintaxe, o MARS muda para a janela de execução e de *debug* da Figura 2.

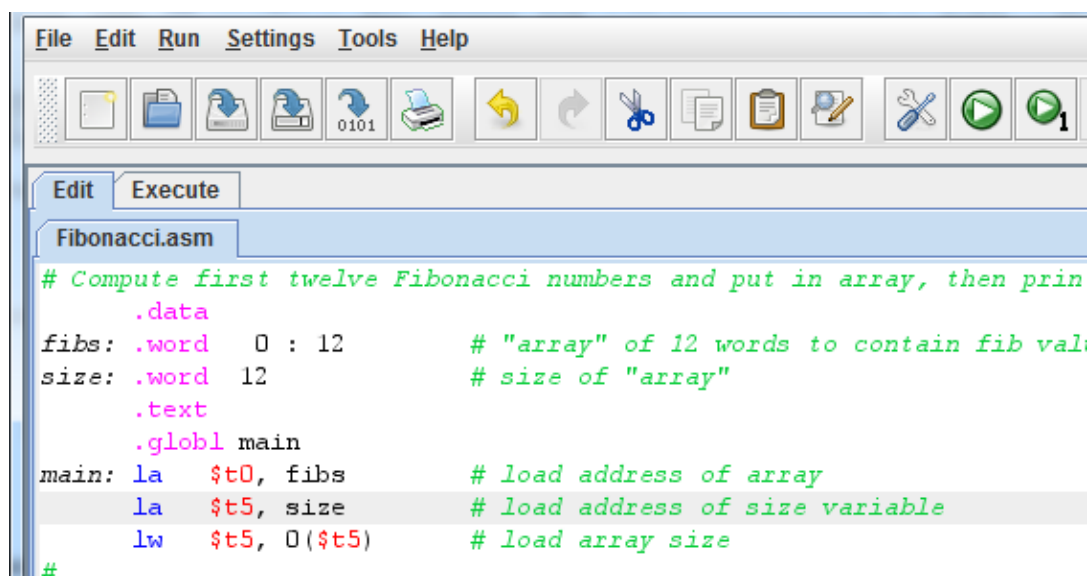




Figura 1 - Janela de edição

1.2 Janela de execução

Nesta janela são apresentados os segmentos de código (.text) e de dados (.data) do programa *Assembly*. No painel do lado direito temos o conjunto dos 32 registos do MIPS, cujo valor pode também ser editado. A consola (*Run I/O*), na parte inferior, permite ao programa interagir com o utilizador, usando chamadas-ao-sistema (*syscalls*) específicas para esse efeito, por exemplo, *print_string*. A execução do programa pode ser feita duma só vez (comando *run* ) , ou em modo passo-a-passo (comando *single step* ). É ainda possível a introdução de pontos-de-paragem (*breakpoints*) para facilitar a deteção e correção de erros (*debug*).

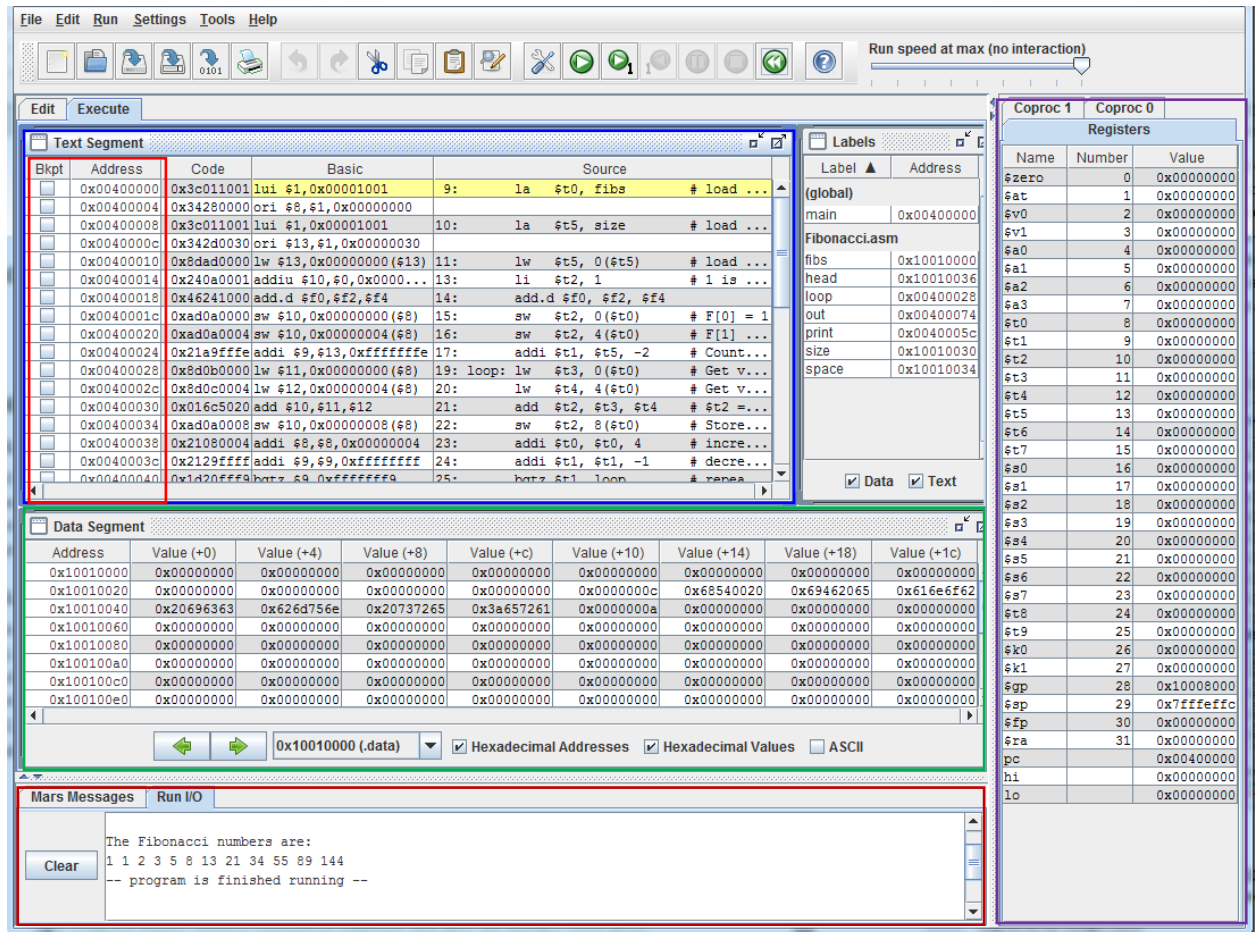


Figura 2 - Janela de execução

- Segmento de código (.text)
- Segmento de dados (.data)
- Registos
- Consola (Run I/O)
- Pontos-de-paragem (Breakpoints)

2. Programação em *Assembly* no MARS

Edite o programa *Assembly* apresentado no Anexo C e guarde-o num ficheiro com a extensão ".asm" (e.g., "trab5_1.asm"). Faça a *assemblagem* do código que editou.

2.1. Segmento de código

Observe o conteúdo do painel *text segment*, em particular a primeira e a última colunas.

- A partir de que endereço foi colocada a 1ª instrução *Assembly* que escreveu?
- Quais as instruções da máquina real que correspondem a essa instrução?
- Qual o código máquina de cada uma das instruções da máquina real que identificou na alínea anterior?
- Qual o valor atual do registo *Program Counter*?

2.2. Segmento de dados

Observe o conteúdo do painel *data segment*. Em que endereços estão colocadas as *strings*: *prompt* e *result*? Se necessário consulte a tabela ASCII de codificação de caracteres.

2.3. Execução e teste

- Run:** Execute o programa e verifique se funciona corretamente.
- Single-Step:** Execute novamente o programa, mas agora passo a passo.
- BreakPoints:** Introduza um *breakpoint* no endereço correspondente à instrução "move \$t0, \$v0", e execute novamente o programa. Execute a parte restante do programa passo a passo, verificando o resultado de todas as instruções. O resultado da execução pode ser visualizado no painel de registos, no segmento de dados e ainda na consola (*Run I/O*).

Anexo A - Programa em Java

```
// Programa em linguagem Java
import java.io.*;
import java.util.Scanner;

public class aula7
{
    public static void main(String[] args) throws NumberFormatException
    {
        String result = "\nO numero em que pensaste e': ";
        String prompt = "1. Pensa num numero!\n2. Adiciona 3\n
                        3. Multiplica o resultado por 2\n
                        4. Subtrai o numero em que pensaste\n\n\t
                        Qual o resultado? ";

        int num;

        System.out.println( prompt );           // print_str( prompt );
        num = new Scanner( System.in ).nextInt(); // num = read_int();
        System.out.print(result);                // print_str( result );
        System.out.println(num-6);               // print_int( num - 6 );
    }
}
```

Anexo B - Programa em C

```
/* Programa correspondente em linguagem C */
```

```

void main(void)
{
    char result[] = "\n0 numero em que pensaste e': ";
    char prompt[] = "1. Pensa num numero!\n2. Adiciona 3\n"
                    "3. Multiplica o resultado por 2\n"
                    "4. Subtrai o numero em que pensaste\n"
                    "\n\t Qual o resultado? ";

    int num;
    print_str( prompt );
    num = read_int();
    print_str( result );
    print_int( num - 6 );

    exit();
}

```

Anexo C - Programa em Assembly

```

# Tradução do programa em linguagem C para assembly do MIPS

.data
result: .asciiz "\n0 numero em que pensaste e': "
prompt: .ascii "1. Pensa num numero!\n"
        .ascii "2. Adiciona 3\n"
        .ascii "3. Multiplica o resultado por 2\n"
        .ascii "4. Subtrai o numero em que pensaste\n"
        .asciiz "\n\tQual o resultado? "
#####
.text
.globl main
# int num;          "num" reside no registo $t0
#
main:
    la    $a0, prompt    # $a0 = prompt
    li    $v0, 4          # $v0 = 4 (syscall "print_str")
    syscall                # print_str( prompt )
    #
    li    $v0, 5          # $v0 = 5 (syscall "read_int")
    syscall                # read_int() (o valor lido é
                          # devolvido no reg. $v0)
    move   $t0, $v0        # $t0 = $v0 ( num = read_int() )
    #
    la    $a0, result     # $a0 = result
    li    $v0, 4          # $v0 = 4 (syscall "print_str")
    syscall                # print_str( result )
    #
    sub    $a0, $t0, 6     # $a0 = $t0 - 6 ( $a0 = num - 6 )
    li    $v0, 1          # $v0 = 1 (syscall "print_int")
    syscall                # print_int( num - 6 )
    #
    li    $v0, 10         #
    syscall                # exit()

```

Parte II – Instruções Lógicas e de Deslocamento

Objetivos

- Instruções lógicas bit-a-bit (*bitwise*).
- Instruções de deslocamento (*shift*) lógico e aritmético.
- Instrução *Syscall* e chamadas ao sistema

Introdução

Algumas operações de uso frequente em programação, tais como mascarar, fazer o *set*, o *reset* ou o *toggle* de um subconjunto dos bits dentro dum registo, são implementadas em *Assembly* do MIPS através das instruções lógicas básicas bit-a-bit (*bitwise*). Outro conjunto de instruções bastante usado, especialmente na manipulação de grandezas binárias, são as instruções de deslocamento (*shift*) aritmético e lógico.

O simulador MARS disponibiliza um vasto conjunto de *system calls* que suportam as operações de entrada e saída de e para a consola (ecrã). Destas seleccionamos algumas cujo conhecimento será essencial na resolução de futuros exercícios mais complexos.

Guião

1. Instruções lógicas

1.1 Codifique um programa em *Assembly* que a partir do valor de dois operandos presentes em \$t0 e \$t1¹ calcule o resultado do AND, OR, NOR e XOR e guarde os resultados em \$t2, \$t3, \$t4 e \$t5, respetivamente.

1.2 Teste o programa e confirme manualmente os resultados para os seguintes pares de valores:

\$t0 = 0x12345678 , \$t1 = 0x0000000F

\$t0 = 0x12345678 , \$t1 = 0x0000F000

\$t0 = 0x12345678 , \$t1 = 0x0000ABCD

1.3 Como poderia implementar a operação de negação bit a bit do registo \$t0 e armazenar o resultado em \$t6?

¹ Ao escrever o programa considere que \$t0 e \$t1 já estão inicializados com os valores previstos. A inicialização pode ser feita editando no próprio simulador o conteúdo do registo antes de correr o programa.

2. Instruções de deslocamento lógico e aritmético

O MIPS disponibiliza três instruções de deslocamento (*shift*): deslocamento à esquerda (lógico) e deslocamento à direita lógico e aritmético².

2.1 Escreva um programa que efectue as três operações de deslocamento considerando como operando o registo \$t0 e a constante **Imm** (número de bits a deslocar). Os resultados devem ser guardados nos registos \$t1, \$t2 e \$t3.

2.2 Teste o programa com os seguintes pares de valores e verifique manualmente os resultados:

- a) 0x12345678, 1
- b) 0x12345678, 4
- c) 0x12345678, 2
- d) 0xF0000003, 4

3. Chamadas ao sistema

3.1 Traduza para *Assembly* e teste a seguinte sequência de código C:

```
print_string( "Introduza dois números :" );
a = read_int();
b = read_int();
print_string( "A soma dos números é: " );
print_int10( a + b )
```

3.2 Teste o programa com valores **a** e **b** positivos e negativos.

3.3 Substitua a instrução **print_int10(a + b)** por **print_intu10(a + b)**. Repita os testes da alínea anterior. O que acontece quando o resultado deveria ser negativo?

4. Usando máscaras e deslocamentos escreva um programa que imprima separadamente no ecrã em hexadecimal cada um dos dígitos hexadecimais do valor armazenado no registo \$t1. Use como exemplo as instruções que se seguem:

```
print_int16((t1 & 0xF0000000) >> 28 ); print_char( ' ' );
print_int16((t1 & 0x0F000000) >> 24 ); print_char( ' ' );
...
```

4.1 Que alterações seriam necessárias para imprimir o valor de \$t1 em base 8?

² A diferença entre o deslocamento à direita lógico e aritmético é que este preserva o sinal do número, isto é, se o número era originalmente positivo continua positivo se era negativo mantém-se também negativo, enquanto o deslocamento lógico ignora o sinal do número.