



## Trabalho Prático 9

### Simulação do *Datapath Single-cycle* do MIPS

#### Objetivos

- Introdução ao simulador do *datapath* do MIPS - ProcSim.
- Análise da execução de programas *Assembly* usando sucessivamente arquitecturas do *datapath* com um conjunto de instruções (ISA) de complexidade crescente.
- Começamos com instruções do tipo-R (Registo); Adicionamos instruções do tipo-I de acesso à memória (lw/sw) e a instrução de salto condicional (beq); Concluímos adicionando ao ISA a instrução do tipo-I (addi) e finalmente a instrução de salto incondicional j, o que representa uma síntese da implementação dos *datapath single-cycle* abordados nas aulas teóricas.

#### Guião

##### 1. O simulador ProcSim, v2.0

O simulador ProcSim (*Processor Simulator*) é um *software* gratuito que executa programas em *Assembly* do MIPS. Durante a execução de cada instrução apresenta graficamente o fluxo de dados no *datapath* e o valor dos sinais de controlo envolvidos. Essencialmente, é composto por um *assembler*, um simulador do MIPS e ainda um visualizador gráfico da actividade no *datapath*.

Este visualizador segue a execução de cada instrução assinalando os caminhos de dados activos, apresentando os valores de entrada e de saída das unidades funcionais, quer sejam dados quer sejam sinais de controlo.

Possibilita a simulação do *datapath single-cycle* do CPU MIPS, com um ISA (*Instruction Set Architecture*) de complexidade variável, através dum ficheiro de configuração da arquitetura, o que do ponto de vista pedagógico se adequa aos objectivos duma disciplina como IAC.

O simulador *ProcSim* é uma programa em Java, o que significa que corre em qualquer plataforma que tenha instalado o *java-runtime* (JRE). Recorde-se que este requisito já foi satisfeito aquando da instalação do simulador *Mars* usado nas aulas práticas de *Assembly*.

### 1.1 Painel de Controlo

O programa arranca apresentando o painel de controlo indicado na **Fig.1**. Este painel dá acesso ao *assembler* (*Assembly Code...*), ao configurador da arquitetura (*Processor Architecture...*) e seguidamente iniciar uma simulação (*Start Simulation*).

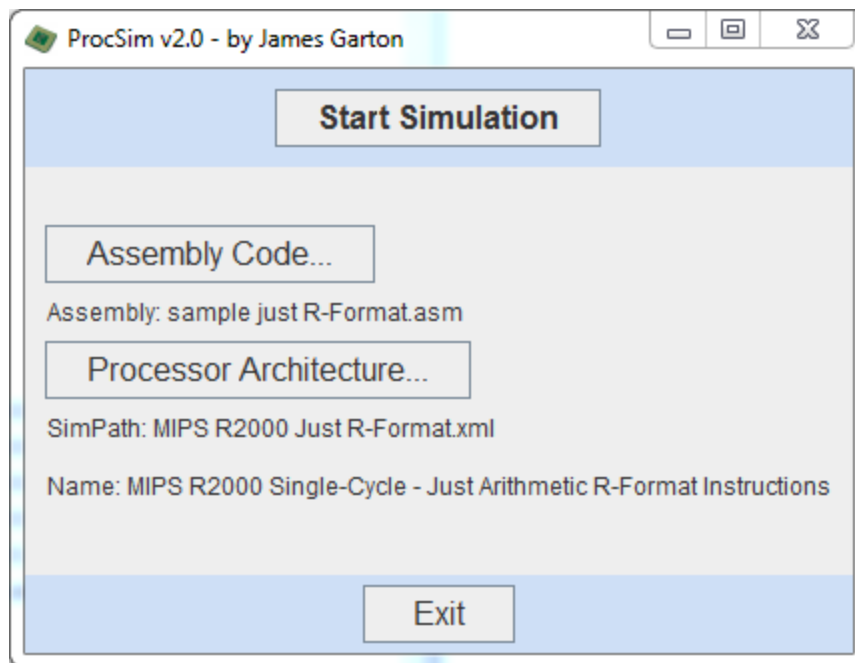


Figura 1 - Painel de Controlo do ProcSim

### 1.2 Configuração Default

Durante a inicialização o ProcSim carrega automaticamente (por *default*) um programa *Assembly* cujo nome é '**sample just R-Format.asm**' (apresentado em **Programa 1**) e um ficheiro de arquitetura do *datapath* '**MIPS R2000 Just R-Format.xml**'.

O objetivo é facilitar a compreensão do funcionamento desta ferramenta apresentando resultados rapidamente.

Para iniciar a simulação pressione o botão *Start Simulation* para aceder à janela de visualização da simulação da **Fig. 3**.

Em funcionamento normal, tanto o ficheiro *Assembly* como o ficheiro de arquitetura do processador podem ser alterados.

### 1.3 Sobre execução do Guião

Devido à sua extensão, poderá haver dificuldades em conseguir executar todos os exercícios sobre as três versões do *datapath single-cyle*. Por este motivo, sugerimos que o preenchimento de algumas tabelas seja relegado para trabalho para casa, privilegiando a visualização e o estudo da simulação dos vários exemplos apresentados.

## 2. Datapath Single-Cycle I - ISA: *and, or, add, sub*

Para simular o **Programa 1** use o *datapath* '1\_MIPS R2000 Just R-Format.xml' com um ISA reduzido, i.e., só com algumas instruções aritméticas e lógicas do tipo-R, apresentado na **Fig. 2**.

### 2.1 Programa Assembly

```
# Example1: Testing R-format arithmetic and logical operators

# init registers
.register $t0 -10
.register $t1 5
.register $t2 3

main: add    $t3, $t1, $t1    # 5+5
      add    $t4, $t1, $t0    # 5+-10
      sub    $t5, $t1, $t2    # 5-3
      sub    $t6, $t2, $t1    # 3-5
      and    $t7, $t2, $t1    # 011 AND 101

exit:
```

Programa 1 - Instruções do tipo-R

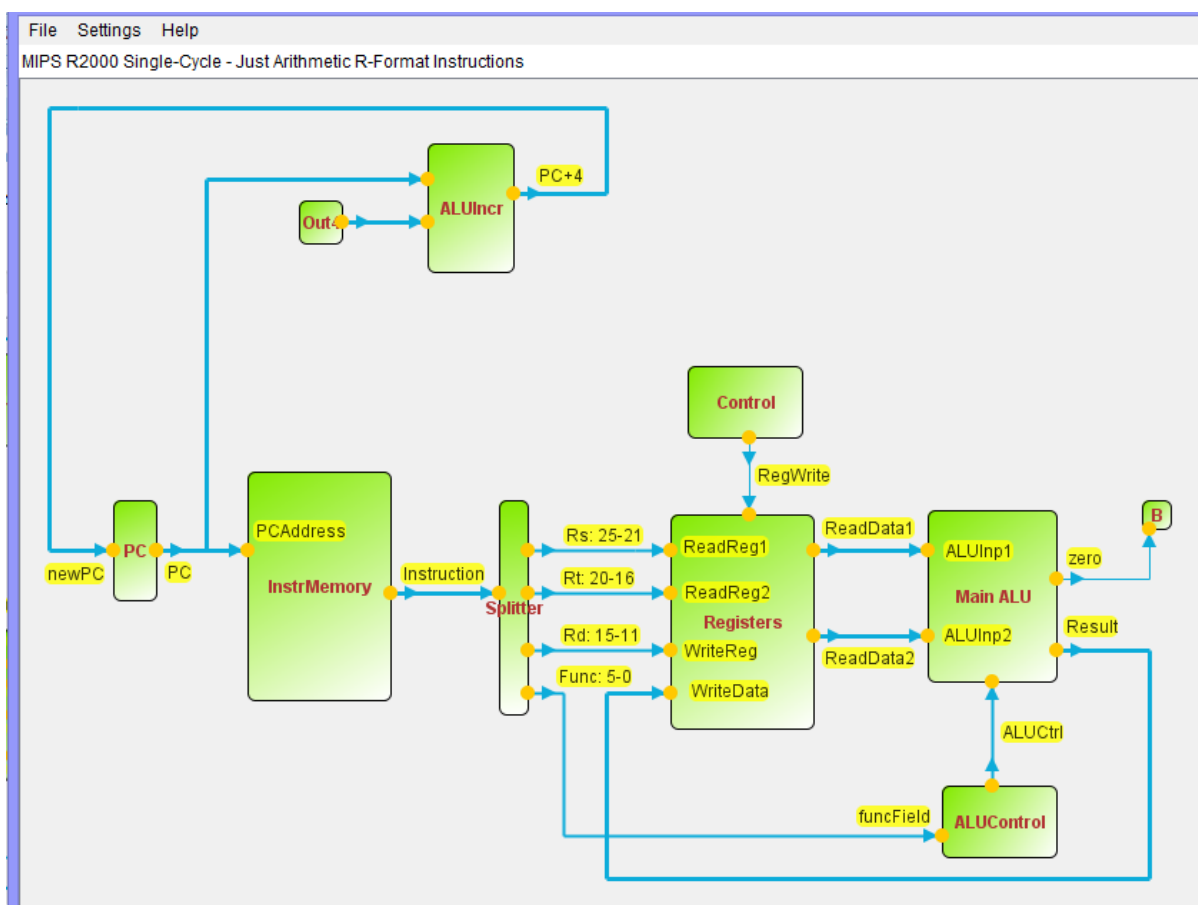


Figura 2 - Datapath Single-Cycle I

## 2.2 Simulação

Para iniciar a simulação da execução do programa carregado na memória de instruções, no *Control Panel* pressiona-se o botão *Start Execution*.

Os botões *Registers*, *Main Memory* (Dados) e *Instruction Memory* permitem aceder ao conteúdo destas unidades funcionais como ilustrado.

A simulação do programa é acompanhada da animação do fluxo de dados nos *buses* e nos sinais de controlo do *datapath* (**Fig. 3**). As linhas activas estão assinaladas a vermelho. Os valores são apresentados em binário e em decimal.

Para uma descrição mais pormenorizada consulte por favor o ficheiro *readme.txt* que acompanha o ProcSim.

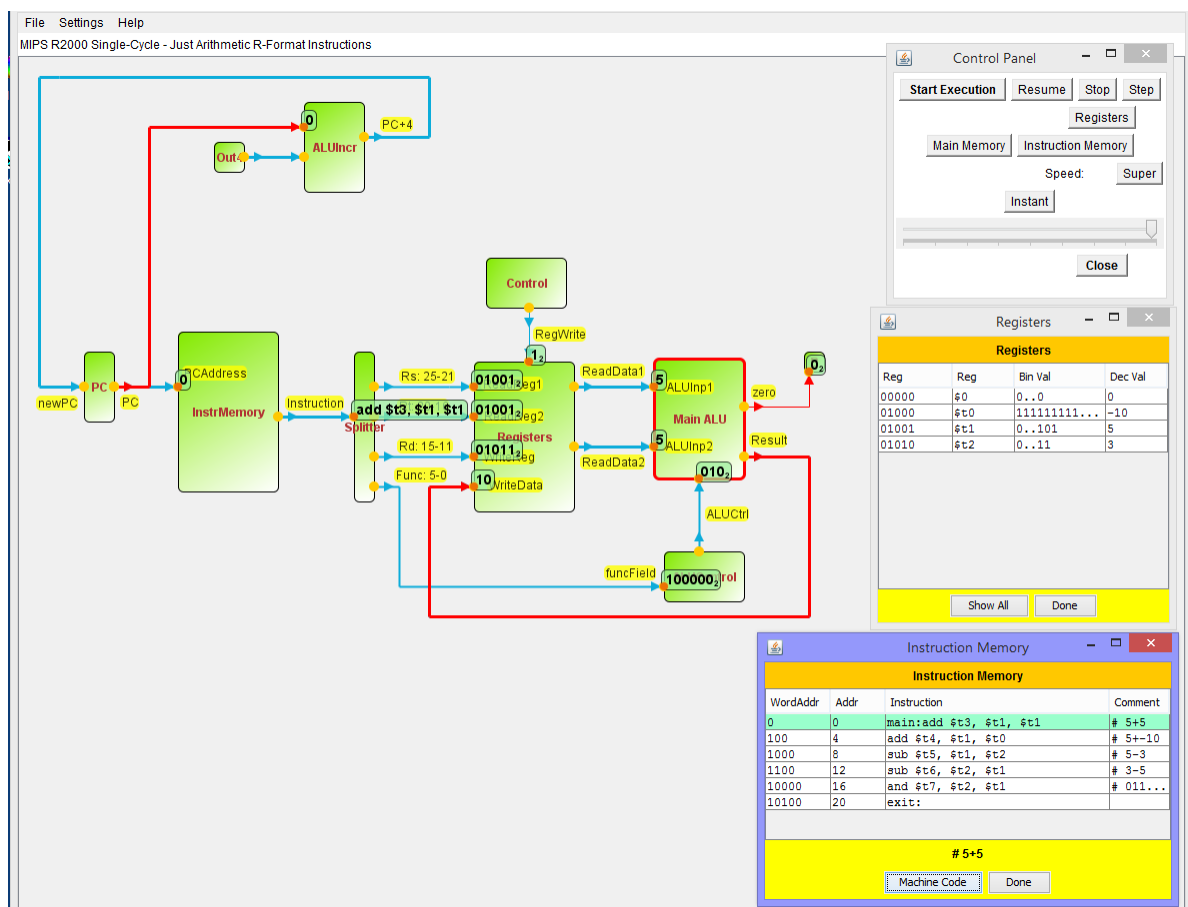


Figura 3 - Datapath Single-Cycle I - Instrução 'add \$t3, \$t1, \$t1'

## 2.3 Número dos registos

Nos exercícios seguintes o número do registo \$t0..\$t7 é o que consta na **Tabela II.1**.

\$t0	\$t1	\$t2	\$t3	\$t4	\$t5	\$t6	\$t7
8	9	10	11	12	13	14	15

Tabela II.1 - Número dos Registos \$t0..\$t7

## 2.4 Trabalho a realizar <sup>1</sup>

Execute a primeira instrução do tipo-R "**add** \$t3, \$t1, \$t1" passo-a-passo (*Step*<sup>2</sup>)

a) Na **Tabela II.2** preencha os campos da instrução, sabendo que o *Funct*<sub>5:0</sub> de **add** é 32.

OpCode 31:26	Rs 25:21	Rt 20:16	Rd 15:11	Shamt 10:6	Funct 5:0

**Tabela II.2 - Tipo-R - Campos da Instrução**

b) Na **Tabela II.3** preencha os os valores de entrada e saída do Banco de Registos e da ALU para cada passo de execução da instrução. Apresente o valor dos dados em decimal e o valor dos sinais de controlo em binário. Os valores em branco (no simulador) assinale-os com um X (*don't care*).

# Step	ReadReg1	ReadReg2	WriteReg	WriteData	ReadData/ ALUInp1	ReadData2/ ALUInp2	ALUCtrl 2:0	Result
4								
6								
7								
8								

**Tabela II.3 - Banco de Registos e ALU**

c) Na **Tabela II.4** preencha os valores do sinal de controlo e os valores dos registos \$t1 e \$t3

# Step	RegWrite	\$t1	\$t3
4			
6			
7			
8			

**Tabela II.4 - Sinal de Controlo e Valores dos Registos**

<sup>1</sup> Sempre que possível, use os valores apresentados pelo simulador ProcSim.

<sup>2</sup> *Step* do ProcSim.

### 3. Datapath Single-Cycle II - ISA: *and, or, add, sub, slt, beq, sw, lw*

Para simular o **Programa 2** use o *datapath* '2\_MIPS R2000 3 arithm slt beq swlw.xml'.

#### 3.1 Programa Assembly

```
# Example2: Initializes an integer array in memory,
# storing the values {2,4,...,18,20}
.register $t0 0      # addr
.register $t1 4      # increm addr
.register $t2 2      # number increm
.register $t3 0      # number val
.register $t4 20     # max number
.register $t5 -1     # lw test

main: add    $t3, $t3, $t2      # increment number val
      sw     $t3, 0($t0)      # store number at the new address
      add    $t0, $t0, $t1     # increment address
      beq    $t3, $t4, load    # check if reached max
      beq    $zero, $zero, main # start again
load: lw     $t5, -4($t0)      # grab last value into $t5
exit:
```

Programa 2 - Inicialização dum *array* de inteiros com instruções do tipo-R, lw/sw e beq

#### 3.2 Trabalho a realizar

a) Para a instrução '**lw** \$t5, -4(\$t0)' preencha a **Tabela III.1**. O  $OpCode_{5:0}$  de **lw** é 35.

31:26 (op)	25:21 (rs)	20:16 (rt)	15:0 (imm16)

Tabela III.1 - Instrução 'lw rt, imm16(rs)';  $rt = Mem[rs + imm32]$ ;  $\langle addr \rangle = rs + imm32$

Determine o valor do **Endereço de Memória** e o valor de **\$t5** após a execução da instrução.

b) Preencha a **Tabela III.2** com os valores os sinais de controlo durante a execução das instruções indicadas.

Instr	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemRead	MemToReg	ALUOp <sub>1:0</sub>	ALUCtrl <sub>2:0</sub>
sw										
lw										
beq										

Tabela III.2 - Sinais de Controlo

#### 4 Datapath Single-Cycle III - ISA: *and, or, add, sub, slt, beq, sw, lw, j, addi*

Para simular o **Programa 3** use o *datapath* '**3\_MIPS R2000 5 all jmp and addi.xml**' .

##### 4.1 Programa Assembly

```
# Example3: Puts the 20 times table into memory starting from -100
# going up to 500, at every word location.
# The memory now acts like an array
```

```
main: addi $t3, $zero, -100 # start val
      addi $t4, $zero, 500 # max number
loop: sw $t3, 0($t0)        # store number at the new address
      addi $t0, $t0, 4      # increment address
      addi $t3, $t3, 20     # increment number val
      beq $t3, $t4, exit   # check if reached max
      j loop               # start again
exit:
```

**Programa 3 - Inicialização dum array de inteiros usando addi e j**

##### 4.2 Trabalho a realizar

a) Para a instrução '**j loop**', preencha a **Tabela IV.1**. O  $OpCode_{5:0}$  de **J** é 2. Determine o valor do **Jump Target Address (JTA)**.

31:26 (op)	25:0 (imm26)

**Tabela IV.1 - Instrução 'j Imm26';  $\langle JTA \rangle = (PC+4)_{31:28} : imm26 \ll 2$**

b) Para as instruções **addi** e **j** preencha na tabela abaixo os valores dos sinais de controlo.

Instr	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	Jump	MemWrite	MemToReg	ALUOp <sub>1:0</sub>	ALUCtrl <sub>2:0</sub>
<b>addi</b>										
<b>j</b>										

**Tabela IV.2 - Control e ALUControl - Sinais de Controlo**

c) Verifique o conteúdo da memória de dados (*Memory*) à maneira que o programa vai sendo executado.

## 5. Exercício adicional - *Bubble Sort*

### 5.1 Programa *Assembly*

Assemble o programa '**4\_bubble\_sort.asm**', usando o *Datapath III* e verifique o seu correcto funcionamento analisando o conteúdo da memória de dados no início e no final do ciclo *do-while*.

```
# Example: bubble-sort
.register $t0 20
.register $t1 17
.register $t2 10
.register $t3 -1
.register $t4 18
.register $t5 -2

# init memory values
main: sw    $t0, 0($zero)    # 20
      sw    $t1, 4($zero)    # 17
      sw    $t2, 8($zero)    # 10
      sw    $t3, 12($zero)   # -1
      sw    $t4, 16($zero)   # 18
      sw    $t5, 20($zero)   # -2

# start the sorting process
# $t2=flag; $t3=i; $t4=j; $t5 = array
      addi   $t5,$zero,0      # $t5 = array
      addi   $t4,$zero,5      # j = SIZE-1
do:    addi   $t2,$zero,1      # do { flag = 1
      add     $t3,$zero,$zero  # i = 0
wh:    slt    $t1,$t3,$t4      # while(i < j)
      beq     $t1,$zero,endif   # {
      add     $t6,$t3,$t3      # $t6 = i+i = 2i
      add     $t6,$t6,$t6      # $t6 = 2i+2i = 4i
      add     $t6,$t6,$t5      # $t6 = array + 4i ->&array[i]
      lw      $t7,0($t6)       # $t7 = array[i]
      lw      $t8,4($t6)       # $t8 = array[i+1]
      slt     $t1,$t8,$t7      #
      beq     $t1,$zero,endif   # if(array[i+1] < array[i]) {
      sw      $t7,4($t6)       # swap values
      sw      $t8,0($t6)       # array[i+1] <-> array[i]
      addi    $t2,$zero,0      # flag = 0 (values swapped)
endif: addi   $t3,$t3,1        # } i++
      j       wh               # }
endw:  addi   $t4,$t4,-1       # j--
      beq     $t2,$zero,do      # } while(flag==0)
exit:                                     # done!
```

Programa 4 - Ordenação de inteiros (*Bubble Sort*)