

## AULAS PRÁTICAS N.º 11 e 12

### Objetivos

- Completar a montagem dos elementos operativos/funcionais de uma versão simplificada do MIPS com arquitetura *multi-cycle*, sobre uma versão incompleta previamente fornecida.
- Realizar testes de funcionamento do processador, executando pequenos programas de teste.

### Introdução

Esta sequência de aulas práticas vai usar como referência o *datapath multi-cycle* que foi apresentado nas aulas teóricas, com suporte para a execução das seguintes instruções: **ADD, SUB, AND, OR, NOR, XOR, SLT, ADDI, SLTI, LW, SW, BEQ e J**. É fornecida a descrição VHDL incompleta de um módulo, designado por "**MIPS\_MultiCycle**", que implementa a funcionalidade de um CPU MIPS capaz de executar as instruções atrás mencionadas e que disponibiliza, na sua interface, os barramentos de endereços e dados, os sinais *Read* e *Write* e ainda o *Reset* e o sinal de relógio (ver Figura 1). O *top-level* do projeto terá assim que instanciar o módulo "**MIPS\_MultiCycle**" e uma memória RAM (para o armazenamento de instruções e dados) e fazer a interligação dos dois através dos barramentos e sinais disponibilizados.

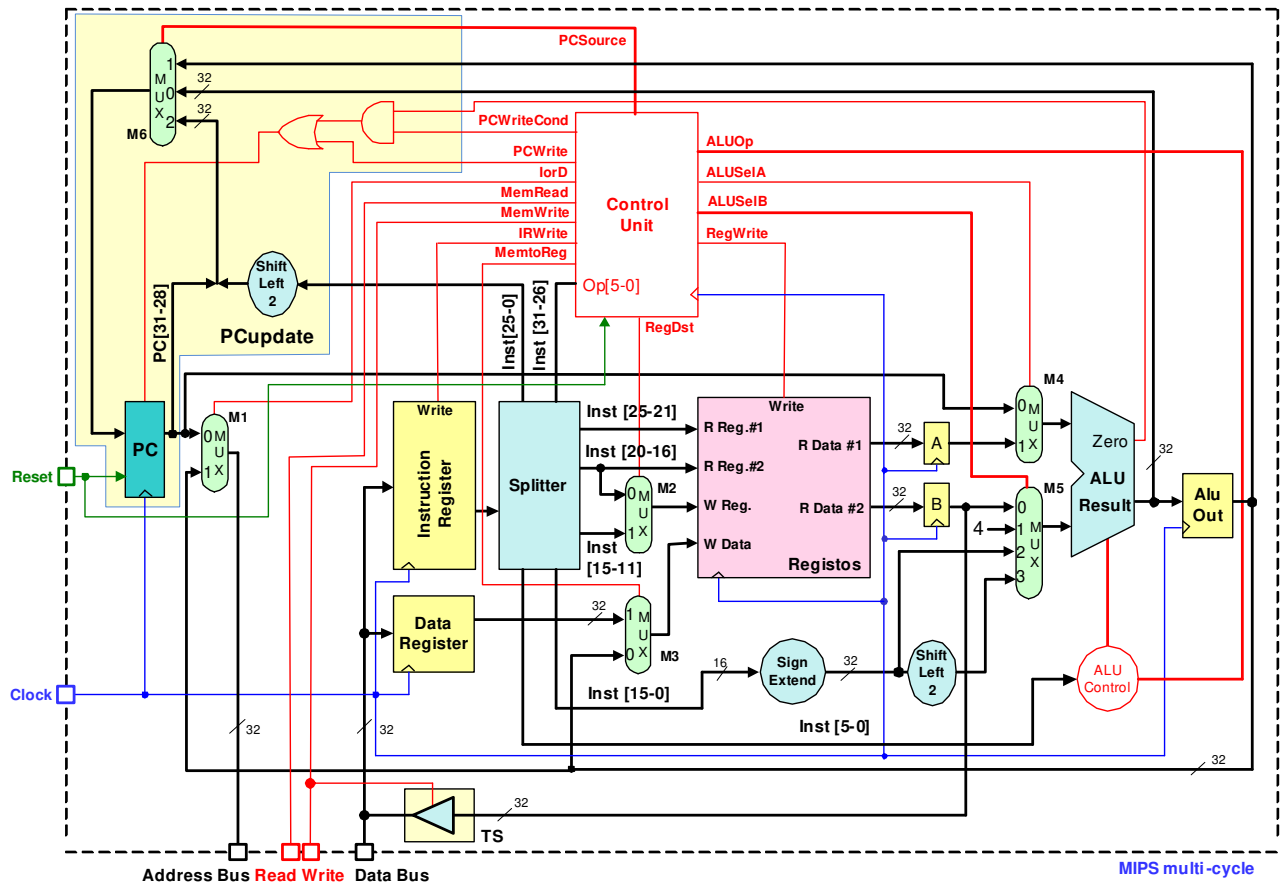


Figura 1. MIPS simplificado com arquitetura *multi-cycle*.

O barramento de dados, através do qual o CPU comunica com o exterior, é um barramento bidirecional, ou seja, permite enviar (escrita) ou receber (leitura) informação de elementos funcionais externos, nomeadamente da memória de dados e instruções. Esta alteração, relativamente à abordagem feita nas aulas teóricas, obriga à inclusão de portas *tri-state* (TS na Figura 1) à saída do registo B, cujo *enable* é controlado pelo sinal *MemWrite* gerado pela unidade de controlo. Deste modo, o barramento de dados, no que depende do processador, estará em baixa

impedância apenas durante a fase de acesso à memória numa instrução *store word* (**SW**) e em alta impedância em todas as restantes instruções e fases.

A maioria dos módulos implementados para a versão *single-cycle* do *datapath* é reaproveitada para a implementação da versão *multi-cycle*. Os módulos específicos para esta versão do *datapath* são: Unidade de Controlo, registo de 32 bits com *enable*; *multiplexer* 4x1 genérico; módulo de atualização do *Program Counter* e *Left Shifter* de 2 bits. Faz-se, de seguida, uma breve descrição destes módulos:

### Módulo de atualização do *Program Counter*

Relativamente à versão *single-cycle*, este módulo tem que ser alterado, uma vez que agora quer o incremento do *Program Counter* (PC) quer o cálculo do *Branch Target Address* (BTA) são feitos na ALU (na primeira e na segunda fase de execução da instrução, respetivamente). Deverá, no entanto, continuar a calcular o *Jump Target Address* (JTA). Assim, este módulo tem as seguintes entradas (ver Figura 2):

- os 26 bits menos significativos do código máquina da instrução, necessários para calcular o JTA;
- a saída da ALU, necessária para obter o valor de PC+4, calculado na primeira fase de execução da instrução;
- a saída do registo AluOut, necessária para obter o valor do BTA, calculado na segunda fase de execução da instrução;
- o sinal "**Zero**" gerado pela ALU;
- os sinais gerados pela Unidade de Controlo, "**PCSource**", "**PCWrite**" e "**PCWriteCond**".

Para além destes é ainda incluído o sinal de "**Reset**" necessário para repor a zero o valor do PC.

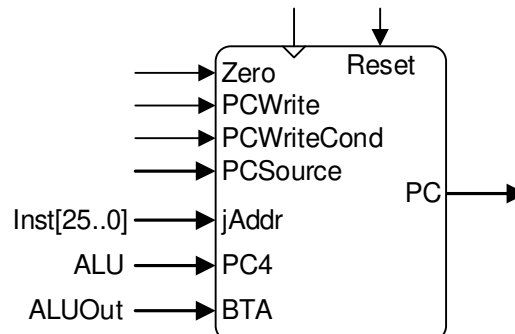


Figura 2. Módulo de atualização do PC.

A tabela seguinte apresenta o valor que a saída do módulo de atualização do PC irá tomar na próxima transição ativa do relógio, em função dos sinais gerados pela unidade de controlo ("**PCSource**", "**PCWrite**" e "**PCWriteCond**") e pela saída "**Zero**" da ALU.

Tabela 1. Valor que a saída do módulo de atualização do PC vai tomar na próxima transição ativa do relógio.

PCWrite	PCWriteCond	Zero	PCSource	PC
0	0	X	XX	PC (não alterado)
0	1	0	XX	PC (não alterado)
0	1	1	01	BTA
1	X	X	00	PC4
1	X	X	10	JTA

### Multiplexers

O *multiplexer* M6 está incluído na descrição comportamental do módulo de atualização do *Program Counter*. Para o *multiplexer* M5 é usado um novo módulo com 4 entradas de N bits e 2 bits de seleção.

### Left Shifter

O *Left Shifter* recupera os 2 bits menos significativos do *offset* da instrução de *branch* que são desprezados no momento da codificação da instrução, isto é, converte o *offset* em termos de número de instruções num *offset* em termos de endereços.

### Registo de 32 bits

Na saída de cada elemento operativo é colocado um registo de 32 bits. Este registo armazena o valor calculado durante o ciclo de relógio corrente, deixando-o disponível para ser usado, noutra elemento operativo, no ciclo de relógio seguinte. Neste módulo é incluído um sinal de *enable*, que é colocado a '1' em todas as instâncias em que a escrita é incondicional (Data Register, A, B e AluOut).

### Unidade de controlo

No *datapath multi-cycle*, cada instrução é decomposta num conjunto de ciclos de execução, correspondendo cada um destes a um período de relógio distinto. Na versão simplificada do *datapath* considerado nesta implementação, o número de ciclos de relógio necessário para executar cada uma das instruções é o seguinte: 3 ciclos de relógio para as instruções de *branch* e *jump*; 4 ciclos de relógio para as instruções tipo R, imediatas e de escrita na memória; 5 ciclos de relógio para a leitura da memória.

A geração dos sinais de controlo ao longo do conjunto de ciclos em que é decomposta cada instrução depende da instrução particular que está a ser executada. A decomposição da execução da instrução num conjunto de fases torna necessário o recurso a uma máquina de estados síncrona para a geração de todos os sinais de controlo.

A Figura 3 apresenta o diagrama de estados completo, modelo de Moore, da unidade de controlo do *datapath multi-cycle*. De salientar que, nessa figura, os sinais de saída não explicitados em cada estado ou são irrelevantes (por exemplo os de seleção dos *multiplexers*) ou encontram-se no estado não ativo, isto é, tomam o valor lógico '0' (sinais de controlo de elementos de estado).

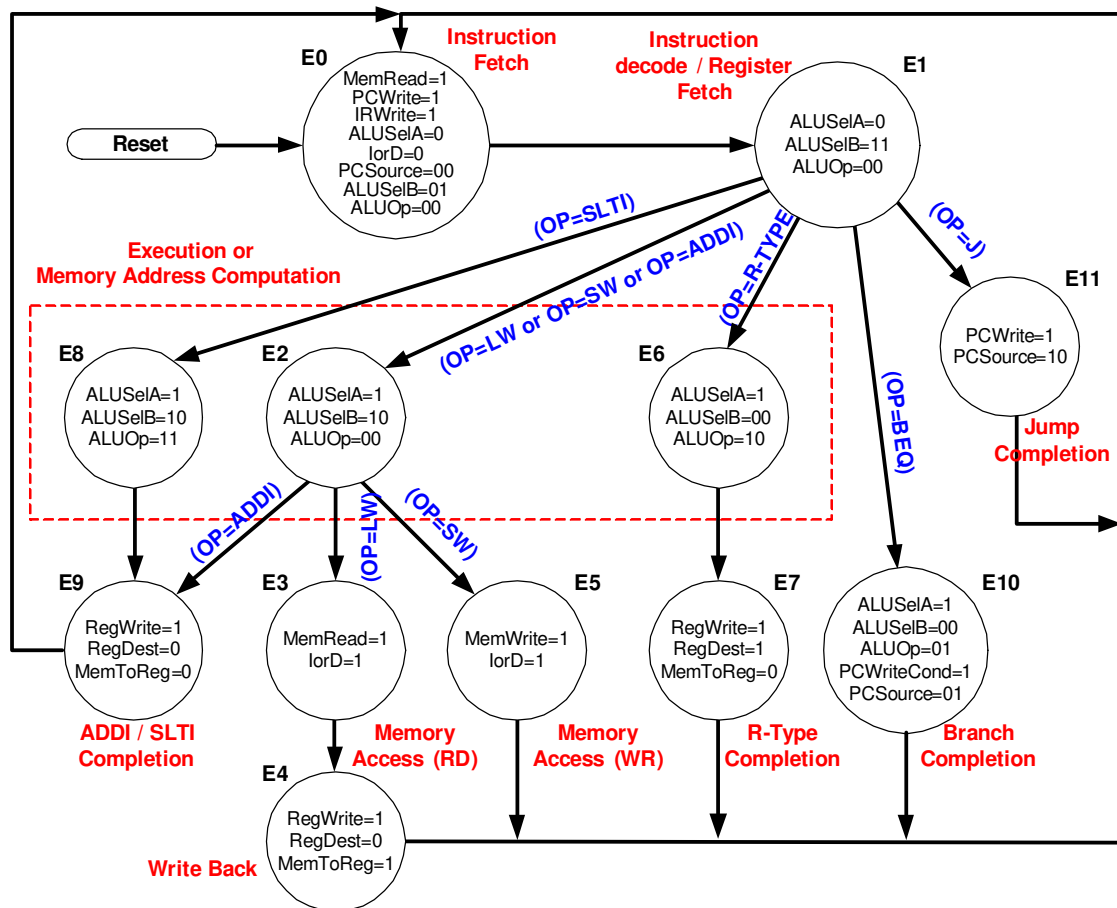


Figura 3. Diagrama de estados da unidade de controlo do *datapath multi-cycle*.

## Sistema computacional mínimo com CPU e memória

Para a implementação de um sistema computacional mínimo será necessário juntar o bloco CPU disponibilizado ("**MIPS\_MultiCycle**") à memória RAM. A interligação entre os dois é feita através dos barramentos de endereços e de dados e dos sinais de controlo *Read* e *Write*, tal como ilustrado na Figura 4.

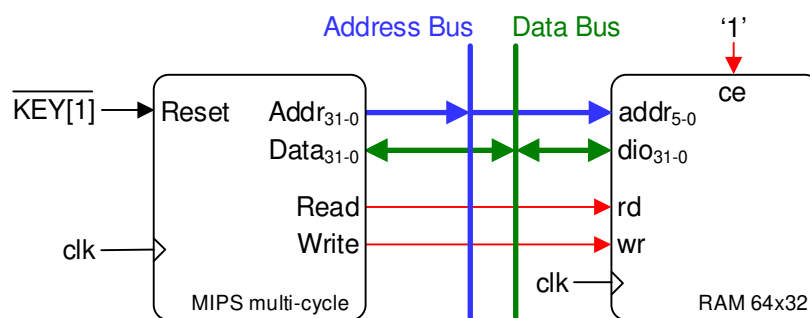


Figura 4. Blocos funcionais do sistema computacional, a instanciar no *top-level* do projeto.

Para além da memória, o sistema pode ainda incluir unidades de entrada/saída ou outros dispositivos, todos interligados através de um barramento de dados comum, ou seja, um barramento de dados partilhado por todos. A ligação a um barramento de dados partilhado requer que cada um desses dispositivos tenha a capacidade de colocar a sua saída para o barramento em alta impedância, exceto quando estiver a ocorrer uma operação de leitura especificamente sobre esse dispositivo. Podendo haver vários dispositivos ligados ao mesmo barramento é, assim, necessário que cada um tenha um sinal de *enable* (**CE** de *chip enable*) que tem o objetivo de o habilitar especificamente a realizar operações de escrita ou leitura e, por essa via, controlar também

o estado de alta impedância do seu barramento de dados: com o sinal **CE** inativo, não é possível realizar qualquer operação sobre o dispositivo e o barramento é colocado em alta impedância. Tal como indicado na Figura 4, o sinal **CE** da memória está, por agora, sempre ativo.

### Memória de dados e instruções

A memória RAM armazena os dados e as instruções do programa e deverá ter uma capacidade de armazenamento de 64 posições de 32 bits cada (256 bytes) (na implementação deste módulo pode tomar como referência o código VHDL disponibilizado na aula teórica n.º 2).

Sendo o espaço de armazenamento partilhado por dados e instruções, será necessário especificar a divisão lógica desse espaço. Uma das formas possíveis consiste em dividir o espaço disponível ao meio, de modo a que as primeiras 32 posições de memória de 32 bits (endereços **0x00** a **0x7F**, considerando a memória *byte-addressable*) sejam utilizadas para o armazenamento do código máquina das instruções do programa e a capacidade restante (endereços **0x80** a **0xFF**) para o armazenamento de dados.

A FPGA usada permite a inicialização de uma RAM, de modo similar ao que se faz para uma memória ROM. O código seguinte exemplifica o procedimento:

```
entity RAM is
    generic( ADDR_BUS_SIZE : positive;
             DATA_BUS_SIZE : positive);
    port (clk      : in std_logic;
          addr     : in std_logic_vector(ADDR_BUS_SIZE - 1 downto 0);
          ce       : in std_logic;  -- chip enable
          wr       : in std_logic;  -- write
          rd       : in std_logic;  -- read
          dio      : inout std_logic_vector(DATA_BUS_SIZE - 1 downto 0));
end RAM;
architecture Behavioral of RAM is
    (...)
    signal s_memory : TMemory := ( X"8C010000",  -- lw      $1,0x0000($0)
                                     X"20210004",  -- addi    $1,$1,4
                                     X"AC010004",  -- sw      $1,4($0)
                                     others => X"00000000");
begin
    (...)
end Behavioral;
```

O endereço de acesso à memória está disponível no barramento de endereços do CPU (é proveniente do módulo de atualização do PC num acesso durante a fase *Instruction Fetch*, ou é o resultado do valor calculado na ALU na fase *Execute* da instrução num acesso durante a fase *Memory*) e tem uma dimensão de 32 bits.

O ISA do MIPS especifica que a memória tem uma organização do tipo *byte-addressable* (a cada endereço está associado o armazenamento de 1 byte). No entanto, a memória que se está a implementar tem uma organização *word-addressable* em que cada endereço corresponde ao armazenamento de 1 *word* de 32 bits. A compatibilização física destes dois modelos faz-se ignorando os 2 bits menos significativos do barramento de endereços o que faz com que o endereço 0 do MIPS corresponda ao endereço 0 da memória, o endereço 4 do MIPS (...0100) corresponda ao endereço 1 da memória, etc. Assim, e uma vez que vão ser necessários 6 bits para aceder a todas as posições da memória ( $2^6=64$  *words* de 32 bits) na ligação do CPU ao barramento de endereços da memória ignoram-se os 2 bits menos significativos e utilizam-se os 6 seguintes, ou seja, utilizam-se os bits A<sub>7</sub> a A<sub>2</sub> do CPU para ligar aos bits A<sub>5</sub> a A<sub>0</sub> da memória.

## Guião

### Parte I

1. Crie no Quartus um projeto, selecionando como FPGA o dispositivo Altera Cyclone IV EP4CE115F29C7. Designe o projeto e a entidade *top-level* por "**MIPS\_System**".
2. Implemente em VHDL o módulo de atualização do *Program Counter*. Poderá designar este módulo por "**PCupdate.vhd**". Selecione este ficheiro como o *top-level* do projeto.
3. Usando o simulador *University Program VWF*, simule funcionalmente o módulo "**PCupdate.vhd**". Verifique o valor de saída do módulo nos seguintes casos (selecione, para os sinais correspondentes a barramentos, a opção "radix=hexadecimal"):
  - a) **PCWrite='1'; PCWriteCond='0'; PCSource="00"**
  - b) **PCWrite='1'; PCWriteCond='0'; PCSource="10"; instr[25..0]=0x05**
  - c) **PCWrite='0'; PCWriteCond='1'; Zero='0'; PCSource="01"; BTA=0x0C**
  - d) **PCWrite='0'; PCWriteCond='1'; Zero='1'; PCSource="01"; BTA=0x0C**
4. Implemente em VHDL um *multiplexer* 4x1 genérico. Designe este módulo por "**Mux41\_N.vhd**".
5. Implemente em VHDL o *Left Shifter* de 2 bits. Designe este módulo por "**LeftShifter2.vhd**".
6. Implemente em VHDL um registo de 32 bits com *enable*, isto é, a escrita só é realizada se, quando ocorrer uma transição de '0' para '1' no relógio, o sinal de *enable* estiver ativo. Designe este módulo por "**Register\_N.vhd**".
7. Implemente em VHDL a memória RAM 64x32 com escrita síncrona e leitura assíncrona, com um barramento de dados bidirecional e sinal de seleção (*chip enable*). Designe este módulo por "**RAM.vhd**" (veja a secção "Memória de dados e instruções" para mais informação). Usando o simulador *University Program VWF*, simule funcionalmente o seu funcionamento.

### Parte II

1. Preencha a tabela abaixo com o nome de cada uma das fases de execução e com o valor que tomam, em cada uma delas, os sinais de controlo aí indicados, para todas as instruções que o *datapath multi-cycle* suporta, isto é, **r-type**, **addi**, **slli**, **lw**, **sw**, **beq** e **j** (uma tabela para cada instrução).

Tabela 2. Sinais de controlo e fases de execução das instruções.

	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
<b>Nome da fase</b>					
<b>PCWriteCond</b>					
<b>PCWrite</b>					
<b>MemWrite</b>					
<b>MemRead</b>					
<b>IRWrite</b>					
<b>RegWrite</b>					
<b>RegDst</b>					
<b>ALUOp</b>					
<b>ALUSelA</b>					
<b>ALUSelB</b>					
<b>MemtoReg</b>					
<b>PCSource</b>					
<b>IorD</b>					

2. Preencha a tabela seguinte com o nome de cada uma das fases de execução da instrução **sw \$7, 0x20(\$6)** e com o valor que tomam, em cada uma delas, os sinais e os valores do *datapath* nos elementos funcionais/operativos ali indicados. Considere que os registos, no instante em que vai iniciar-se o *instruction fetch*, têm os seguintes valores: **\$7=0x12F4C6A8**, **\$6=0x00000004** e **\$PC=0x00000008**.

Tabela 3. Fases de execução e valores do *datapath*.

	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
<b>Nome da fase</b>					
<b>PC</b>					
<b>Instr. Register</b>					
<b>Data Register</b>					
<b>A</b>					
<b>B</b>					
<b>ALU Result</b>					
<b>ALU Out</b>					
<b>ALU Zero</b>					

### Parte III

1. Descarregue o arquivo "**MIPS\_System.zip**" disponível no moodle de AC1. Esse arquivo inclui:
  - *top-level* ("**MIPS\_System.vhd**") com a instanciação dos seguintes módulos: "**MIPSMultiCycle.vhd**" (implementação do *datapath multi-cycle* da arquitetura MIPS); "**RAM**" (memória RAM para dados e instruções, parametrizável); "**DebounceAndClock.vhd**" (*debouncer* e geração de sinal para simular *clock*); "**DisplayUnit.vhd**" (módulo de visualização); "**DivFreq.vhd**" (divisor de frequência); note que alguns destes módulos estão apenas parcialmente instanciados.
  - módulo "**DebounceAndClock.vhd**" com alterações relativamente à versão anteriormente usada que permite a geração de um pulso ou de múltiplos pulsos, dependendo da duração temporal do sinal de entrada.
  - módulo "**MIPS\_MultiCycle.vhd**" com a implementação incompleta do *datapath multi-cycle*.
  - módulo "**ControlUnit.vhd**" com a implementação incompleta da unidade de controlo.
2. Acrescente ao projeto os módulos VHDL necessários para a implementação do *datapath multi-cycle* (desenvolvidos no ponto anterior e/ou reaproveitados da implementação *single-cycle*). Complete a implementação do módulo "**MIPS\_MultiCycle.vhd**" (note que todos os módulos necessários estão parcialmente instanciados, pelo que deve verificar a coerência dos nomes dos módulos desenvolvidos anteriormente).
3. A unidade de controlo ("**ControlUnit.vhd**") está parcialmente implementada. Analise o código VHDL fornecido e complete-o (de acordo com os valores com que preencheu a Tabela 2).
4. O sinal de *clock* deve ser ligado, através do *debouncer*, a uma tecla (**KEY[0]**) da placa de desenvolvimento. O sinal de *reset* deve ser ligado à tecla **KEY[1]** (não se esqueça de inverter este sinal).

5. Verifique as ligações dos sinais globais do módulo de visualização, de modo a poder observar:
  - o valor à saída do módulo de atualização do PC (sinal "DU\_PC");
  - o conteúdo dos registos do *Register File* (sinais "DU\_RFdata" e "DU\_RFaddr");
  - o conteúdo da memória (sinais "DU\_DMdata" e "DU\_DMaddr");
  - o valor à saída da ALU (sinal "DU\_IMdata" anteriormente usado para observar a memória de instruções e que agora será usado para visualizar o resultado que está a ser calculado pela ALU);
  - o estado atual da máquina de estados da unidade de controlo (sinal "DU\_CState").
6. Efetue a síntese e implementação do projeto e elimine todos os erros que forem detetados pela ferramenta.

### Parte IV

Com o *datapath multi-cycle* concluído podem agora ser feitos testes de funcionamento, executando pequenos programas que permitam verificar o correto funcionamento do processador. Assim, para cada um dos exemplos que se seguem, deve inicializar a memória RAM com o código máquina das instruções, fazer a síntese e implementação do projeto e reprogramar a FPGA. Note que o espaço de armazenamento em RAM disponível terá agora que ser partilhado por instruções e dados. A divisão poderá ser a seguinte: instruções (**text** segment) entre o endereço **0x00** e o endereço **0x7F**; dados (**data** segment) entre os endereços **0x80** e **0xFF**.

O sinal de relógio do *datapath* é gerado manualmente premindo a tecla **KEY[0]** da placa de desenvolvimento (o módulo de *debouncing* gera um pulso, com a duração de 1 ciclo do seu relógio de referência, sempre que é premida a tecla **KEY[0]**, ou múltiplos pulsos com a duração de 1 ciclo de relógio cada, sempre que a tecla é premida de forma mais prolongada). Desse modo é possível observar a execução de cada instrução, ciclo a ciclo. No módulo de visualização pode acompanhar a evolução da máquina de estados na execução de cada uma das instruções do programa.

#### Programa de teste 1:

Address	Code	Instr	Comment
0x00000000		main: addi \$3,\$0,0x55	
0x00000004		sw \$3,0x80(\$0)	
0x00000008		lw \$4,0x80(\$0)	
0x0000000C		w1: j w1	while(1);

- a) Quantos ciclos de relógio demora a execução do programa, desde o *instruction fetch* da primeira instrução até à conclusão da instrução "lw \$4,0x80(\$0)"?

#### Programa de teste 2:

Address	Code	Instr	Comment
		main: addi \$3,\$0,0x8F	
		addi \$4,\$0,0x36	
		slt \$1,\$3,\$4	
		beq \$1,\$0,else	
		sw \$3,0x80(\$0)	
		j endif	
		else: sw \$4,0x80(\$0)	
		endif: j endif	while(1);



- Quantos ciclos de relógio demora a execução do programa, desde o *instruction fetch* da primeira instrução até ao instante em que vai iniciar-se o *instruction fetch* da instrução referenciada pelo label "**endif**"?
- Pretende-se agora usar como relógio do *datapath* um sinal com uma frequência de 8 Hz, obtido por divisão de frequência do relógio de referência da placa de desenvolvimento (**CLOCK\_50**, 50 MHz). Para isso ligue o sinal de saída do divisor de frequência (já instanciado no *top-level* do projeto) ao sinal de relógio do CPU e da memória (desligue a saída do *debouncer*: "**pulsedOut** => **open**")<sup>1</sup>.
- Teste novamente o funcionamento dos programas de teste; note que fazendo *reset* reinicia a execução do programa.

### Programa de teste 3:

```
void main(void)
{
    int i, soma;
    static int array[5]; // Considera-se que o "Data segment"
                        // tem início no endereço 0x00000080

    int *p = array;
    for(i=4, soma=0; i > 0; i--)
    {
        *p = 2 * i;
        soma = soma + 2 * i;
        p++;
    }
    *p = soma;
}
```

### # Mapa de registos

```
$5: i
$6: p
$7: soma
$8: aux
```

Address	Code	Instr	Comment
		main: addi \$5,\$0,4	i = 4;
		addi \$6,\$0,0x80	p = 0x80;
		add \$7,\$0,\$0	soma = 0;
		for: slt \$1,\$0,\$5	while(i > 0)
		beq \$1,\$0,endif	{
		add \$8,\$5,\$5	
		sw \$8,0(\$6)	*p = 2 * i;
		add \$7,\$7,\$8	soma += 2 * i;
		addi \$6,\$6,4	p++;
		addi \$5,\$5,-1	i--;
		j for	}
		endif: sw \$7,0(\$6)	*p = soma;
		w1: beq \$0,\$0,w1	while(1);
		nop	

- Observe o conteúdo da memória, no segmento de dados (endereço **0x80**).
- Quantos ciclos de relógio demora a execução completa do programa?

<sup>1</sup> Note que os programas de teste fornecidos terminam com um ciclo infinito de modo a impedir que a execução continue para além da última instrução válida do programa.

**Programa de teste 4:**

O objetivo deste programa é ordenar um *array* de 6 *words*, armazenado no segmento de dados da memória RAM a partir do endereço **0x80**. Para executar e testar este programa terá que adicionar à inicialização da memória as 6 *words* com os valores iniciais do *array*, a partir da posição **0x20** (a que corresponde o endereço **0x80**). As posições não ocupadas entre o final do programa e o início dos dados terão que ser preenchidas (por exemplo com **x"00000000"**), de modo a garantir que os dados ficam armazenados nos endereços pretendidos.

Sugerem-se os seguintes valores iniciais para o *array*: **20, 17, -2, 25, 5, -1**. O trecho de código VHDL seguinte (incompleto) mostra a inicialização da memória com estes valores.

```

subtype TDataWord is std_logic_vector(31 downto 0);
type TMemory is array(0 to 31) of TDataWord;
signal s_memory : TMemory := (
    --
    .text (address = 0x00)
    X"20050080", -- addi $5,$0,0x80
    (...)
    X"00000000",
    (...)
    X"00000000",
    --
    .data (address = 0x80)
    -- array:
    .word 20, 17, -2, 25, 5, -1
    X"00000014", -- .word 20
    X"00000011", -- .word 17
    X"FFFFFFFE", -- .word -2
    X"00000019", -- .word 25
    X"00000005", -- .word 5
    X"FFFFFFF", -- .word -1
    others => X"00000000");

```

```

#define SIZE 6
#define TRUE 1
#define FALSE 0

void main(void)
{
    static int array[]={20, 17, -2, 25, 5, -1};
    int sorted, i, j, aux;
    j = SIZE - 1;
    do {
        sorted = TRUE;
        for (i=0; i < j; i++) {
            if (array[i] > array[i+1])
            {
                aux = array[i];
                array[i] = array[i+1];
                array[i+1] = aux;
                sorted = FALSE;
            }
        }
        j--;
    } while (sorted == FALSE);
}

```

# Mapa de registros:

\$2 - flag

\$3 - i

\$4 - j

\$5 - &amp;array[0]

Address	Code	Instr	Comment
			.text
0x00		main: addi \$5,\$0,0x80	\$5=&array[0]
0x04		addi \$4,\$0,5	j = SIZE - 1
		do:	do {
0x08		addi \$2,\$0,1	sorted = 1
0x0C		addi \$3,\$0,0	i = 0
0x10		for: slt \$1,\$3,\$4	while(i < j)
0x14		beq \$1,\$0,endif	{
0x18		add \$6,\$3,\$3	
0x1C		add \$6,\$6,\$6	aux = i << 2
0x20		add \$6,\$6,\$5	\$6 = array + i
0x24		lw \$7,0(\$6)	\$7=array[i]
0x28		lw \$8,4(\$6)	\$8=array[i+1]
0x2C		slt \$1,\$8,\$7	if(\$7 > \$8)
0x30		beq \$1,\$0,endif	{
0x34		sw \$7,4(\$6)	array[i+1]=\$7
0x38		sw \$8,0(\$6)	array[i]=\$8
0x3C		addi \$2,\$0,0	sorted = 0
		endif:	}
0x40		addi \$3,\$3,1	i++
0x44		j for	}
0x48		endif: addi \$4,\$4,-1	j--
0x4C		beq \$2,\$0,do	} while(sorted == 0)
0x50		nop	
0x54		w1: beq \$0,\$0,w1	while(1);
0x58		nop	
(...)		nop	
0x7C		nop	
			.data
0x80	0x00000014		.word 20
0x84	0x00000011		.word 17
0x88	0xFFFFFFFFE		.word -2
0x8C	0x00000019		.word 25
0x90	0x00000005		.word 5
0x94	0xFFFFFFFFF		.word -1
0x98	0x00000000		
(...)	0x00000000		
0xFC	0x00000000		

- Observe o conteúdo da memória, no segmento de dados, e verifique se o *array* ficou corretamente ordenado.
- Quantos ciclos de relógio demora a execução completa do programa, considerando que o *array* já está ordenado por ordem crescente?

PDF gerado em 17/10/2019