AULA PRÁTICA N.º 8

Objetivos

- Iniciar a montagem dos elementos operativos/funcionais de um *datapath single-cycle*.
- Implementar em VHDL e testar os seguintes módulos do *datapath*: atualização do *Program Counter*, memória de instruções (ROM), extensor de sinal e separação dos campos da instrução.

Não faça *copy/paste* do código que foi disponibilizado nas aulas teóricas. Escrever o código VHDL ajuda-o a entender a estrutura e o funcionamento do *datapath*.

Introdução

Nesta sequência de aulas práticas, vai ser implementado o *datapath single-cycle* que foi apresentado nas aulas teóricas. Este *datapath* terá suporte para a execução das seguintes instruções: **`ADD`**, **`SUB`**, **`AND`**, **`OR`**, **`NOR`**, **`XOR`**, **`SLT`**, **`ADDI`**, **`SLTI`**, **`LW`**, **`SW`**, **`BEQ`** e **`J`**.

Nesta aula, faremos a implementação de quatro módulos: módulo de atualização do *Program Counter*, memória ROM para armazenamento do código máquina das instruções, extensor de sinal e módulo de separação dos campos da instrução (*splitter*). Para a implementação destes módulos tome como referência o código apresentado nas aulas teóricas.

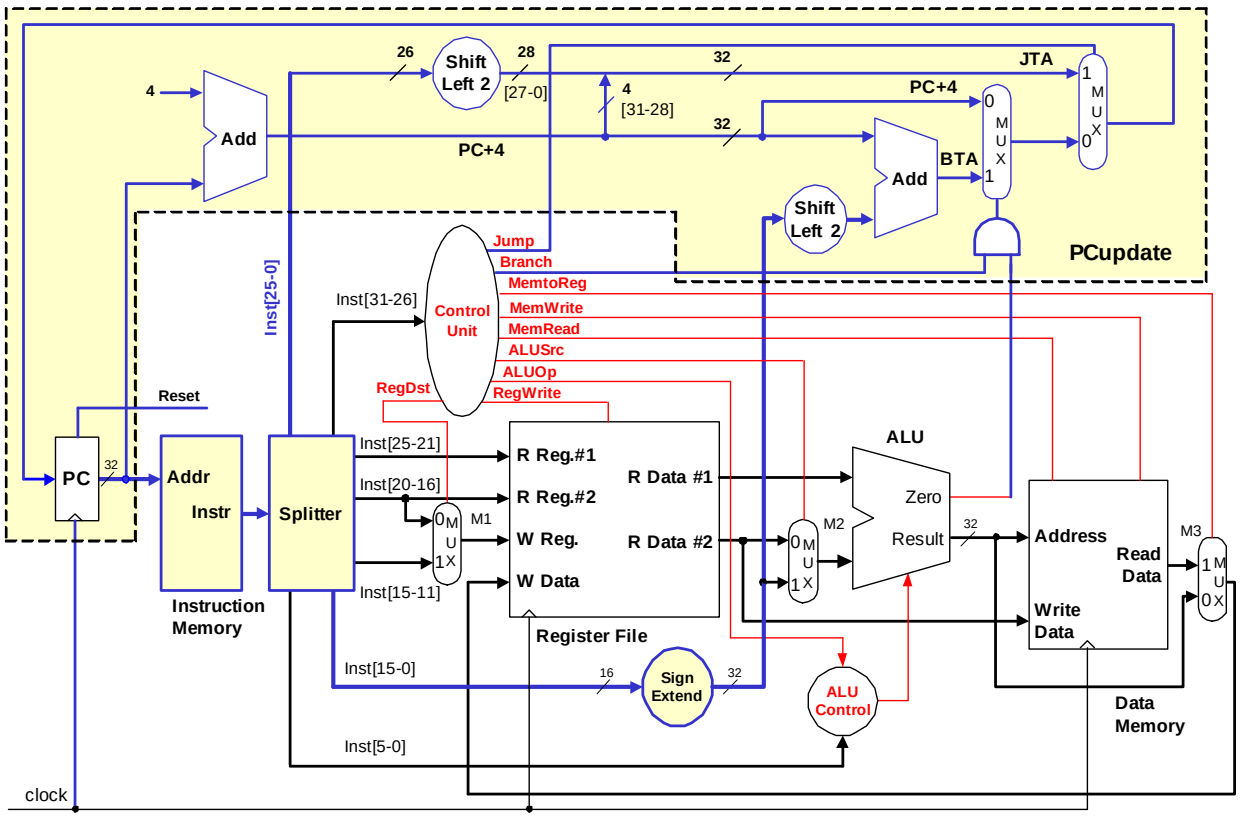

Figura 1. *Datapath single-cycle* com os módulos a implementar nesta aula desenhados com fundo colorido e as ligações a azul.

Módulo de atualização do *Program Counter*

O módulo que se pretende implementar está representado na Figura 2 e tem como objetivo a atualização do registo *Program Counter* (PC) em função de um conjunto de sinais provenientes da unidade de controlo. Este módulo deve apresentar o mesmo comportamento lógico que o conjunto de blocos representado na Figura 1 na zona sombreada (delimitada pela linha tracejada). Recorde-se que o registo *Program Counter* fornece o endereço da instrução a ser lida da memória de instruções (memória ROM).

A saída do módulo de atualização do PC disponibiliza o valor atual do Program Counter. O PC, quando atualizado, pode tomar um de 4 valores: 1) o valor anterior do PC adicionado à constante 4, no caso de a instrução não ser nem um *jump* nem um *branch*, ou, sendo um *branch*, a condição de igualdade ser falsa; 2) o *Branch Target Address*, no caso de a instrução ser um *branch* e a condição de igualdade ser verdadeira; 3) o *Jump Target Address*, no caso de a instrução ser um *jump*; 4) o valor **`0x00000000`**, no caso de a entrada de *reset* estar ativa.

De modo a simplificar o desenvolvimento, este módulo deve ainda calcular o *Branch Target Address* e o *Jump Target Address*.

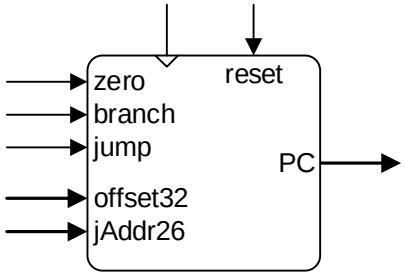

Figura 2. Módulo de atualização do PC.

Na entrada "**`offset32`**" deverá ser introduzido o valor do *offset* do código máquina da instrução, extendido para 32 bits (proveniente do módulo extensor de sinal), necessário para o cálculo do *Branch Target Address*. Por seu lado, na entrada "**`jAddr26`**" devem ser ligados os 26 bits menos significativos do código máquina da instrução (provenientes do módulo *splitter*), necessários para o cálculo do *Jump Target Address*.

O sinal de "**`reset`**" coloca, de forma síncrona com o relógio, o valor **`0x00000000`** no registo PC, o que permite reiniciar a execução do programa a partir da primeira instrução.

A saída deste módulo é um valor de 32 bits dos quais, como se verá na secção seguinte, apenas 6 serão aproveitados.

Memória de instruções

A memória de instruções (ROM) armazena o código máquina de cada uma das instruções do programa a executar no *datapath*. Nesta implementação a capacidade da ROM será limitada a 64 posições de 32 bits cada (6 bits de endereço), ou seja, os programas em teste neste *datapath* poderão ter, no máximo, 64 instruções (que ocuparão 256 bytes).

O endereço gerado pelo módulo de atualização do PC tem, como visto anteriormente, 32 bits. Contudo, para aceder às 64 posições de memória são necessários apenas 6 bits. Quais serão então os bits a considerar desse conjunto de 32?

O ISA do MIPS estabelece que a memória é *byte addressable*, ou seja, cada endereço armazena 1 byte. Por outro lado, na memória que se pretende implementar, cada posição de memória armazena 1 *word* de 32 bits, i.e., 4 bytes (2^2). Esta aparente contradição é facilmente resolvida ignorando os 2 bits menos significativos do PC e usando os 6 seguintes (A_7 a A_2). Desta forma, em cada acesso é lida uma *word* de 32 bits que está armazenada em memória a partir de um endereço múltiplo de 4.

A dimensão do barramento de endereços das memórias de dados e instruções (que determina o número de *words* de 32 bits que cada uma pode armazenar) é definida na *package* "**MIPS_pkg.vhd**" que deverá ser incluída na entidade onde instanciar as memórias. Essa *package* inclui, assim, as seguintes declarações:

```
constant ROM_ADDR_SIZE : positive := 6;
constant RAM_ADDR_SIZE : positive := 6;
```

Estas contantes deverão ser usadas para definir as contantes genéricas de parametrização do número de bits do barramento de endereços das memórias de instruções e de dados.

Extensor de sinal

O extensor de sinal estende para 32 bits os 16 bits menos significativos da instrução, preservando o sinal da quantidade original, isto é, o bit 15 é replicado nos 16 bits mais significativos do valor de saída.

Separação dos campos da instrução (*splitter*)

Este módulo limita-se a fazer a subdivisão dos 32 bits do código máquina da instrução nos campos que são necessários nos diversos pontos do *datapath* como, por exemplo, os 6 bits correspondentes ao "**opcode**", ou os 5 bits correspondentes a cada um dos campos "**rs**", "**rt**" e "**rd**". A utilização deste módulo tem apenas como objetivo simplificar as interligações entre a saída da memória de instruções e os restantes elementos do *datapath*.

```
entity InstrSplitter is
  port(instruction   : in  std_logic_vector(31 downto 0);
       opcode        : out std_logic_vector(5  downto 0);
       rs            : out std_logic_vector(4  downto 0);
       rt            : out std_logic_vector(4  downto 0);
       rd            : out std_logic_vector(4  downto 0);
       shamt         : out std_logic_vector(4  downto 0);
       funct         : out std_logic_vector(5  downto 0);
       imm           : out std_logic_vector(15 downto 0);
       jAddr         : out std_logic_vector(25 downto 0));
end InstrSplitter;

architecture Behavioral of InstrSplitter is
begin
       opcode    <= instruction(31 downto 26);
       rs        <= instruction(25 downto 21);
       rt        <= instruction(20 downto 16);
       rd        <= instruction(15 downto 11);
       shamt     <= instruction(10 downto  6);
       funct     <= instruction( 5 downto  0);
       imm       <= instruction(15 downto  0);
       jAddr     <= instruction(25 downto  0);
end Behavioral;
```

Módulo de visualização

O módulo de visualização (ver Figura 3) permite observar os valores presentes em alguns pontos-chave do *datapath*, não interferindo de nenhum modo com o seu funcionamento. Esses valores são apresentados, em hexadecimal, nos 8 *displays* da placa de desenvolvimento.

Os pontos de visualização que estão definidos são: valor atual do *Program Counter*, conteúdo da memória de instruções (ROM), conteúdo dos registos do *Register File* e conteúdo da memória de

dados (RAM). A seleção de qual a entrada a visualizar é feita através da entrada "**InputSel**", de acordo com a Tabela 1.

O endereço de memória ou o número do registo a visualizar pode ser incrementado/decrementado pulsando a entrada "**NextAddr**". O modo de atualização é escolhido através da entrada "**Dir**". Se **Dir='1'**, a próxima ativação de "**NextAddr**" incrementa o endereço/registo; se **Dir='0'**, decrementa. Este módulo inclui já um *debouncer* para a entrada "**NextAddr**". A ativação prolongada desta entrada permite o incremento/decremento rápido do endereço.

Tabela 1. Seleção da informação a mostrar no módulo de visualização.

InputSel	Entrada selecionada	Módulo	Símbolo em HEX7 e HEX6 quando DispMode = '1'
00	PC	Program Counter	≡P
01	IM	Instruction Memory	≡C
10	RF	Register File	≡F
11	DM	Data memory	≡d

A entrada "**DispMode**" permite escolher entre visualizar os 32 bits do valor (8 dígitos hexadecimais) ou visualizar apenas os 16 bits menos significativos (4 dígitos hexadecimais). No segundo caso (**DispMode = '1'**), aproveitam-se os restantes 4 *displays* para mostrar o endereço do registo/posição de memória que está a ser visualizado/a e qual a entrada que está selecionada para visualização (P, C, F ou d). No caso em que a entrada selecionada é P (isto é PC) o campo de endereço é usado para mostrar o estado corrente da máquina de estados da unidade de controlo do *datapath multi-cycle* (a ver mais tarde).

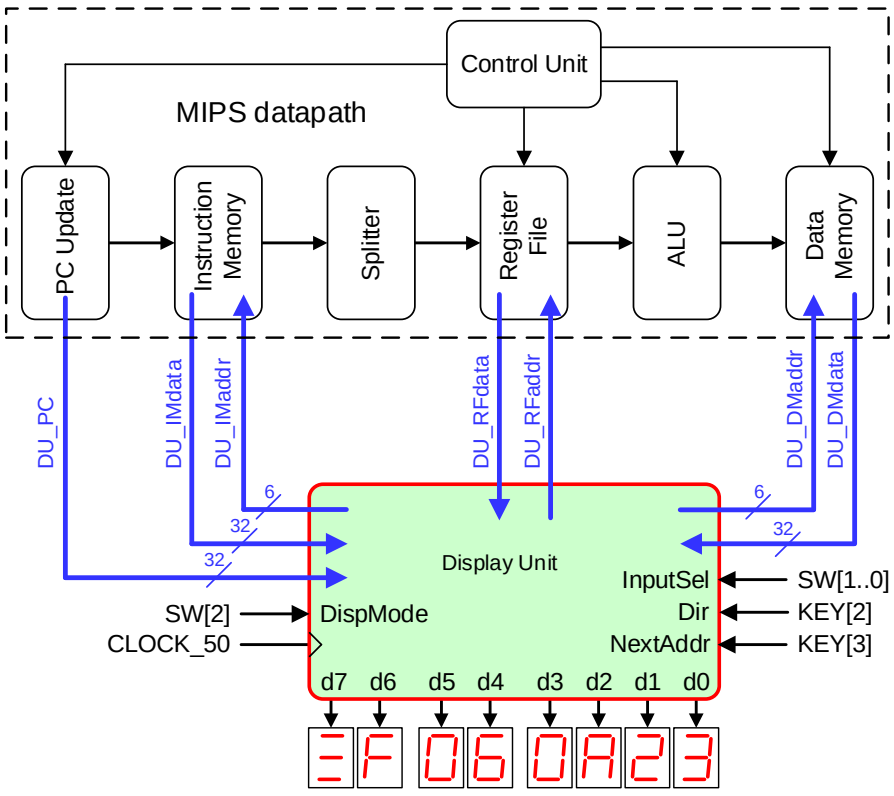

Figura 3. Visão simplificada do *datapath* mostrando a ligação do módulo de visualização aos pontos de observação através de sinais globais (ligações a azul).

Este módulo deve ser instanciado no *top-level* do projeto e ligado a portos de entrada e de saída da placa de desenvolvimento (*switches*, *keys*, *displays*) e ao relógio padrão da placa (**`CLOCK_50`**). Estas ligações estão representadas a preto (traço mais fino) no diagrama da Figura 3. As ligações a azul (traço mais grosso) correspondem às ligações aos pontos de visualização do *datapath*.

As ligações aos pontos de visualização são feitas através de sinais globais, isto é, sinais que permitem a comunicação entre vários módulos sem a necessidade de definir explicitamente portos na interface desses módulos. A utilização desta técnica, não recomendável na modelação *standard* de um sistema digital através de VHDL, visa, neste caso, simplificar as ligações entre o módulo de visualização e o *datapath*.

A ligação dos pontos de observação aos sinais globais obriga à implementação de pontos de leitura assíncrona nas memórias e no banco de registos (funcionam como portos adicionais de leitura sem, contudo, aparecerem explicitamente definidos na interface desses módulos).
Os sinais globais estão declarados numa *package* (**`DisplayUnit_pkg.vhd`**), com os nomes que estão representados na Figura 3, que deve ser incluída nos módulos para os quais se requer um ponto de observação.

O código VHDL do módulo de visualização está disponível no moodle de AC1.

Guião

Parte I

1. Crie no Quartus um projeto, selecionando como FPGA o dispositivo Altera Cyclone IV EP4CE115F29C7. Pode designar o projeto e a entidade *top-level* por "mips_single_cycle".
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 (não se esqueça de selecionar, para os sinais correspondentes a barramentos, a opção "radix=hexadecimal"):
   a) **branch='0'; jump='0'**
   b) **branch='0'; jump='1'; instr[25..0]=0x0000005**
   c) **branch='1'; jump='0'; zero='0'; instr[15..0]=0x0014**
   d) **branch='1'; jump='0'; zero='1'; instr[15..0]=0xFFFC;**
      valor atual do PC: 0x0000001C
   e) **branch='1'; jump='0'; zero='1'; instr[15..0]=0x0014**
4. Implemente em VHDL a memória ROM de 64 posições de 32 bits (64x32). Poderá designar este módulo por "**InstructionMemory.vhd**". Inicialize a memória com o código máquina das seguintes instruções (codifique manualmente as instruções ou recorra ao *assembler* do MARS):

Posição na ROM	Address	Código máquina	Instrução	opcode	Funct
0	0x00000000	0x2002001A	addi $2,$0,0x1A	0x08	-
1	0x00000004		addi $3,$0,-7	0x08	-
2			add $4,$2,$3	0x00	0x20
3			sub $5,$2,$3	0x00	0x22
4			and $6,$2,$3	0x00	0x24
5			or $7,$2,$3	0x00	0x25
6			nor $8,$2,$3	0x00	0x27
7			xor $9,$2,$3	0x00	0x26
8			slt $10,$2,$3	0x00	0x2A
9			slti $11,$7,-2	0x0A	-
10			slti $12,$9,-25	0x0A	-
11		0x00000000	nop	0x00	0x00

5. Simule funcionalmente a memória de instruções, verificando o valor de saída para os endereços 0 a 11 (não se esqueça de selecionar este ficheiro como o *top-level* do projeto).
6. Implemente na memória ROM ("**InstructionMemory.vhd**") um ponto de leitura assíncrona que deve ser ligado aos sinais globais "**DU_IMaddr**" e "**DU_IMdata**" do módulo de visualização. O código VHDL que se apresenta a seguir ilustra o procedimento:

```
library work;
use work.DisplayUnit_pkg.all;

entity InstructionMemory is
  generic(ADDR_BUS_SIZE : positive := 6);
  port(address     : in  std_logic_vector(ADDR_BUS_SIZE-1 downto 0);
       readData    : out std_logic_vector(31 downto 0));
end InstructionMemory;
```

```
(...)
-- Porto de leitura da memória, definido na interface do módulo
readData <= s_memory(to_integer(unsigned(address)));

-- Ponto de leitura para efeitos de visualização (ligado ao módulo
-- de visualização através dos sinais globais DU_IMaddr e DU_IMdata)
DU_IMdata <= s_memory(to_integer(unsigned(DU_IMaddr)));
```

Este ponto de leitura permite observar o conteúdo de cada uma das 64 posições de memória de forma independente do funcionamento do *datapath*.

7. Implemente em VHDL o módulo extensor de sinal. Poderá designar este módulo por "**SignExtend.vhd**".
8. Selecione o ficheiro "**SignExtend.vhd**" como o *top-level* do projeto, e simule o seu funcionamento, observando a sua saída para os seguintes valores de entrada: **0x1234**, **0x835A**.
9. Acrescente ao projeto o módulo de separação dos campos da instrução abordado anteriormente ("**InstrSplitter.vhd**").

Parte II

Após o desenvolvimento e teste dos 4 módulos descritos, passamos agora ao início da construção do *datapth single-cycle*, tomando como referência o diagrama da Figura 1 e o diagrama simplificado da Figura 4. Para tal, terá que ser criado um novo ficheiro onde se fará a instanciação e a interligação desses módulos, e que constituirá o *top-level* do projeto. Para a implementação deste módulo deverá fazer a instanciação e a interligação usando VHDL.

1. Crie um novo ficheiro VHDL ("**mips_single_cycle.vhd**") onde deverá instanciar e interligar a memória de instruções, o módulo de atualização do PC, o extensor de sinal e o módulo de separação dos campos da instrução (ver Figura 4). O sinal de *clock* deve ser ligado, através do *debouncer*, a um pulsador (**KEY[0]**) da placa de desenvolvimento. O sinal de *reset* deve ser ligado à tecla **KEY[1]** (não se esqueça de inverter este sinal).

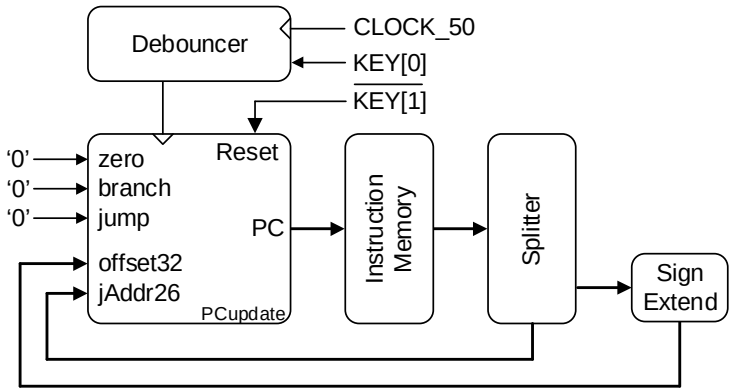

Figura 4. Diagrama mostrando os blocos, e respetivas interligações, a implementar nesta aula.

2. Acrescente ao diagrama o módulo de visualização. Ligue a entrada "**NextAddr**" a **KEY[3]**, a entrada "**Dir**" a **KEY[2]**, a entrada "**InputSel**" a **SW[1..0]** e "**DispMode**" a **SW[2]**. Ligue, igualmente, as saídas "**disp0**" a "**disp7**" aos *displays* **HEX0** a **HEX7**.

Apresenta-se, de seguida, um esboço incompleto do *top-level* em VHDL, com a instanciação completa do *debouncer* (necessário para se poder usar uma tecla da placa, **KEY**, como *clock*) e do módulo de visualização.

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.MIPS_pkg.all;
use work.DisplayUnit_pkg.all;

entity mips_single_cycle is
  port( CLOCK_50: in  std_logic;
        SW      : in  std_logic_vector(2  downto 0);
        KEY     : in  std_logic_vector(3  downto 0);
        HEX0    : out std_logic_vector(6  downto 0);
       (...)
end mips_single_cycle;

architecture Structural of mips_single_cycle is
  signal s_clk : std_logic;
begin
pcupdt:  entity work.PCupdate(Behavioral)
         (...)
instmem: entity work.InstructionMemory(Behavioral)
         generic map(ADDR_BUS_SIZE => ROM_ADDR_SIZE)
         (...)
splitter:entity work.InstrSplitter(Behavioral)
         (...)
signext: entity work.SignExtend(Behavioral)
         (...)
-- Debouncer
debnc: entity work.DebounceUnit(Behavioral)
         generic map(mSecMinInWidth  => 100,
                     inPolarity      => '0',
                     outPolarity     => '1')
         port map( refClk      => CLOCK_50,
                   dirtyIn     => KEY(0),
                   pulsedOut   => s_clk);
-- Display module
displ: entity work.DisplayUnit(Behavioral)
         generic map(dataPathType => SINGLE_CYCLE_DP,
                     IM_ADDR_SIZE => ROM_ADDR_SIZE,
                     DM_ADDR_SIZE => RAM_ADDR_SIZE)
         port map( RefClk  => CLOCK_50,
                   InputSel=> SW(1 downto 0),
                   DispMode=> SW(2),
                   NextAddr=> KEY(3),
                   Dir     => KEY(2),
                   disp0   => HEX0,
                   (...)
                   disp7   => HEX7);
end Structural;
```

3. Ligue aos sinais globais os pontos de observação do *Program Counter* e da memória de instruções (ver ponto 6 da parte 1). Poderá ainda usar o sinal global "**`DU_RFdata`**" para observar o valor do código máquina da instrução à saída da memória de instruções.

4. Selecione o ficheiro "**`mips_single_cycle.vhd`**" como o *top-level*, efetue a síntese e a implementação do projeto e programe a FPGA (não se esqueça de importar o ficheiro de *assignments*, "**`master.qsf`**", disponível no moodle da UC).

5. Usando o módulo de visualização, observe os valores armazenados na memória de instruções (selecione a entrada 1, i.e., **`SW[1..0]=`**"**`01`**", e pulse a entrada "**`NextAddr`**" para avançar para o endereço múltiplo de 4 seguinte).

6. Pulse sucessivamente a tecla **`KEY[0]`** (que simula o relógio do sistema) e observe o valor do PC e o valor à saída da memória de instruções. Faça o *reset* ao sistema (relembre que o *reset* é síncrono) e observe de novo o funcionamento.

Anexo

Regras gerais para utilização do software Quartus da Altera.

1. Para uma melhor organização do trabalho deverá ser criada uma pasta para cada guião (e.g. Aula8, Aula9,..., Aula13).

2. Cada ficheiro apenas deverá conter uma entidade VHDL. O nome da entidade VHDL e do ficheiro onde está armazenada têm que ser rigorosamente iguais.

3. Não devem ser utilizados espaços nem caracteres especiais (por exemplo, acentos) nos *paths* dos projetos e nos nomes dos ficheiros. Isso significa que não se devem gravar os projetos em sub-diretórios do tipo "Ambiente de Trabalho", "Os meus documentos" ou "Aulas\1ºsemestre\". Se o nome de utilizador já tiver um espaço (por exemplo "Shy Guy") terá que ser criada uma pasta na raiz do sistema de ficheiros (por exemplo "**`c:\AC1\Aula8`**").

4. Na utilização do computador da sala de aula:
   a) Devido a limitações de velocidade da rede no acesso à área de trabalho na ARCA (arca.ua.pt) recomenda-se a utilização de uma *pen-drive* ou de um disco externo para armazenamento dos ficheiros e dos projetos.
   b) No caso de se utilizar a ARCA deve especificar-se a drive "**`z:`**" para indicar raiz da estrutura de pastas. Por exemplo, se se pretender criar um projeto na pasta AC1 da ARCA, deve indicar-se o diretório de trabalho como "**`z:\AC1`**" (e nunca como arca.ua.pt\AC1).

PDF gerado em 23/10/2018