AULA PRÁTICA N.º 13

Objetivos

- Implementar em VHDL um porto de entrada e um porto de saída de 32 bits.
- Implementar em VHDL o módulo de descodificação de endereços para a memória e para os portos de entrada e de saída.
- Instanciar e interligar os módulos anteriores no *top-level* do projeto iniciado na aula anterior.
- Realizar testes de funcionamento através da execução de programas de teste.

Introdução

A comunicação do computador com o exterior é feita através de circuitos especializados, vulgarmente designados por periféricos ou unidades de entrada/saída. Na sua versão mais simples podemos definir dois tipos de periféricos: periféricos de entrada que só suportam operações de leitura (designados habitualmente por portos de entrada) e periféricos de saída que só suportam operações de escrita (também designados por portos de saída).

Um porto de saída memoriza o valor escrito pelo processador e os bits correspondentes ao valor memorizado podem ser ligados a dispositivos externos, como por exemplo um LED ou o controlo ON/OFF de um motor. Por outro lado, um porto de entrada não tem memória e, quando lido, limita-se a transferir para o barramento de dados do processador o valor dos bits produzidos pelos dispositivos externos a ele ligados, como por exemplo um *switch*, tal como esquematizado na Figura 1.

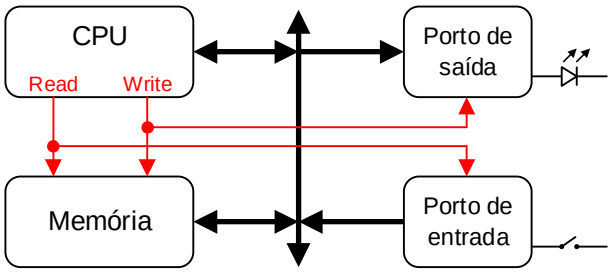

Figura 1. Interação do computador com o exterior através de portos de entrada e de saída.

Os sinais de controlo **`Read`** e **`Write`** gerados pelo CPU definem o tipo de operação a realizar sobre a memória ou sobre os periféricos: a memória suporta operações de leitura e escrita; o porto de entrada só suporta operações de leitura (i.e., só usa o sinal **`Read`**); o porto de saída só suporta operações de escrita (i.e., só usa o sinal **`Write`**). Se só fossem usados estes dois sinais para controlar as operações de transferência de informação, ocorreriam dois problemas: i) numa operação de escrita seriam escritos simultaneamente o porto de saída e a memória, o que não pode acontecer e ii) numa operação de leitura a memória e o porto de entrada colocariam os seus valores nas mesmas linhas do barramento de dados, o que se traduziria num conflito elétrico que levaria, com grande probabilidade, à destruição física dos circuitos.

Para que os problemas anteriormente apontados não aconteçam, todos os dispositivos têm de ter, na sua interface, para além dos sinais **`Read`** e/ou **`Write`**, um sinal que permita a ativação individualizada apenas quando necessário, ou seja, quando o CPU pretender aceder a algum deles. Esse sinal é habitualmente designado por sinal de seleção e é usada a sigla **`CS`** (*Chip Select*) ou **`CE`** (*Chip Enable*). A existência deste sinal permite que apenas um de todos os dispositivos presentes no sistema seja selecionado para a troca de informação. Adicionalmente, os dispositivos que podem ser lidos (e.g. a memória ou o porto de entrada), quando o seu sinal de seleção não estiver ativo, colocam as suas saídas de dados em alta impedância.

Portos de entrada e saída

A descrição comportamental em VHDL de um porto de saída é apresentada de seguida. De notar que a escrita é síncrona e que apenas ocorre se os sinais de seleção (**ce**) e de escrita (**wr**) estiverem simultaneamente ativos.

```
entity OutPort is
     port(clk    : in std_logic;
          ce     : in std_logic;  -- chip enable
          wr     : in std_logic;  -- write
          dataIn : in std_logic_vector(31 downto 0);
          dataOut: out std_logic_vector(31 downto 0));
end OutPort;

architecture behav of OutPort is
begin
   process(clk)
   begin
      if(rising_edge(clk)) then
         if(ce = '1' and wr = '1') then
            dataOut <= dataIn;
         end if;
      end if;
   end process;
end behav;
```

Apresenta-se a seguir a descrição comportamental em VHDL de um porto de entrada. Neste caso a leitura é assíncrona e só ocorre quando os sinais de seleção (**ce**) e de leitura (**rd**) estiverem simultaneamente ativos. Quando isso não acontece a saída de dados do porto está em alta impedância.

```
entity InPort is
   port(rd     : in  std_logic;  -- read
        ce     : in  std_logic;  -- chip enable
        dataIn : in  std_logic_vector(31 downto 0);
        dataOut: out std_logic_vector(31 downto 0));
end InPort;

architecture behav of InPort is
begin
   process(rd, ce, dataIn)
   begin
      if(ce = '1' and rd = '1') then
         dataOut <= dataIn;
      else
         dataOut <= (others => 'Z');
      end if;
   end process;
end behav;
```

Espaço de endereçamento e mapa de endereços

O espaço de endereçamento é a coleção de todos os endereços que o CPU pode gerar e é dependente do número de bits do barramento de endereços. Se o barramento de endereços tiver N bits, o CPU pode gerar 2^N endereços, i.e., desde 0 a 2^N-1. No caso do MIPS o barramento de endereços tem 32 bits, logo podem ser gerados endereços desde **0x00000000** a **0xFFFFFFFF**.

Cada dispositivo do sistema tem obrigatoriamente atribuídos um ou uma gama contígua de endereços, sendo que numa organização do tipo *byte-addressable* a cada endereço corresponde um dispositivo com capacidade para armazenar/produzir uma quantidade de informação de 1 byte. Dependendo da complexidade do sistema, o espaço de endereçamento pode estar apenas parcialmente ocupado, ou seja, poderá haver endereços que não estejam atribuídos a qualquer

dispositivo físico. A tabela que especifica que dispositivos estão disponíveis e quais os endereços que lhe estão atribuídos designa-se por mapa de endereços. Um exemplo de um mapa de endereços, para o sistema que estamos a construir, está apresentado na Tabela 1.

Tabela 1. Mapa de endereços para um sistema com uma memória de 256 bytes (64 words de 32 bits) e um porto de entrada e um porto de saída de 32 bits.

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº de bits de endereço
Memória RAM	256	0x00000000	0x000000FF	8 (A_7-A_0)
Porto de Saída de 32 bits	4	0x00000100	0x00000103	2 (A_1-A_0)
Porto de Entrada de 32 bits	4	0x00000100	0x00000103	2 (A_1-A_0)

De notar que o ISA do MIPS estabelece para o espaço de endereçamento uma organização do tipo *byte-addressable*, razão pelo qual são atribuídos 4 endereços a um porto de 32 bits. Por outro lado, o acesso a informação no exterior, na implementação feita nestas aulas, é feito com as instruções **`lw`** e **`sw`** que transferem 32 bits de informação a partir de endereços múltiplos de 4.

O mapa de endereços tem que garantir que não há dois dispositivos mapeados no mesmo endereço. Então, porque razão se pode atribuir a mesma gama de endereços ao porto de saída e ao porto de entrada, tal como é apresentado na Tabela 1?

Descodificação de endereços

Tendo especificado o mapa de endereços, coloca-se então a questão de como o implementar, isto é, como determinar qual o dispositivo a ativar quando o CPU gera um endereço. A solução geral é a implementação de um circuito combinatório, independente do CPU e dos periféricos, que compare o endereço gerado pelo CPU com os endereços especificados no mapa e, caso esteja numa gama válida, ative o sinal de seleção correspondente ao dispositivo. Ou seja, um circuito que, a partir dos bits do barramento de endereços, gere um conjunto de sinais de seleção de 1 bit, um por cada dispositivo do sistema. Este processo é designado por descodificação de endereços e o circuito que a implementa como descodificador de endereços.

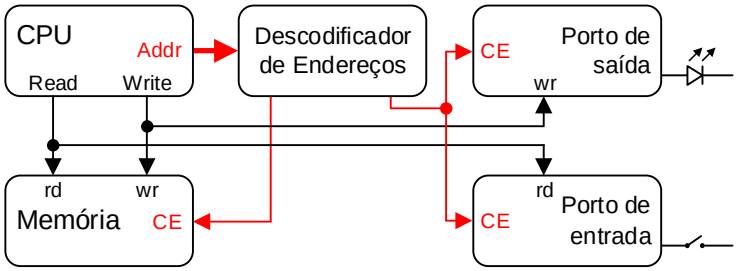

Figura 2. Inclusão do descodificador de endereços para gerar linhas de seleção para a memória e para os portos de entrada/saída.

Retomando a questão colocada no ponto anterior, a atribuição do mesmo endereço ao porto de entrada e ao porto de saída não significa que ambos estejam ativos ao mesmo tempo, uma vez que a operação a realizar depende do estado dos sinais **`Read`** e **`Write`**, que nunca estão ativos simultaneamente. Se o sinal de seleção dos portos estiver ativo e o sinal **`Write`** também estiver ativo, então só o porto de saída está efetivamente selecionado; por outro lado, se, com o mesmo sinal de seleção ativo, o sinal **`Read`** estiver ativo só o porto de entrada realiza a operação.

A implementação do descodificador de endereços pode ser mais ou menos complexa, dependente do número de dispositivos do sistema e das restrições que possam existir relativamente à utilização do espaço de endereçamento. Observando a Tabela 1, verifica-se que, para os endereços especificados, o bit **A_8** permite distinguir entre acesso à memória (**A_8=0**) e acesso aos portos de entrada/saída (**A_8=1**). O descodificador de endereços poderia então ser o que está apresentado na Figura 3 (que é a implementação de *decoder* 1:2).

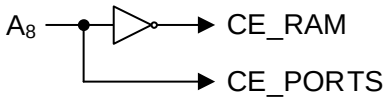

Figura 3. Possível descodificador para o mapa de endereços da Tabela 1.

Com este circuito simplificado garante-se que os endereços especificados na Tabela 1 estão efetivamente atribuídos aos dispositivos do sistema. Não se garante, contudo, que haja um único endereço possível para aceder a cada um dos dispositivos. Por exemplo, os portos também podem ser acedidos nos endereços **0x104** ou **0x108** ou **0x300**, etc. Ou seja, para cada dispositivo há mais do que um endereço possível para o acesso. A descodificação simplificada que conduz a situações destas designa-se por descodificação parcial e é, sempre que possível, usada porque conduz a circuitos descodificadores mais simples.

Sistema computacional completo com CPU, memória e portos

Com a inclusão dos portos de entrada e de saída o sistema computacional que estamos a desenvolver passa a poder interagir com os recursos da placa de desenvolvimento da FPGA. O porto de saída pode ligar aos LEDs vermelhos da placa (LEDR[17..0]) e o porto de entrada aos *switches* SW (SW[17..0]), tal como esquematizado na Figura 4.

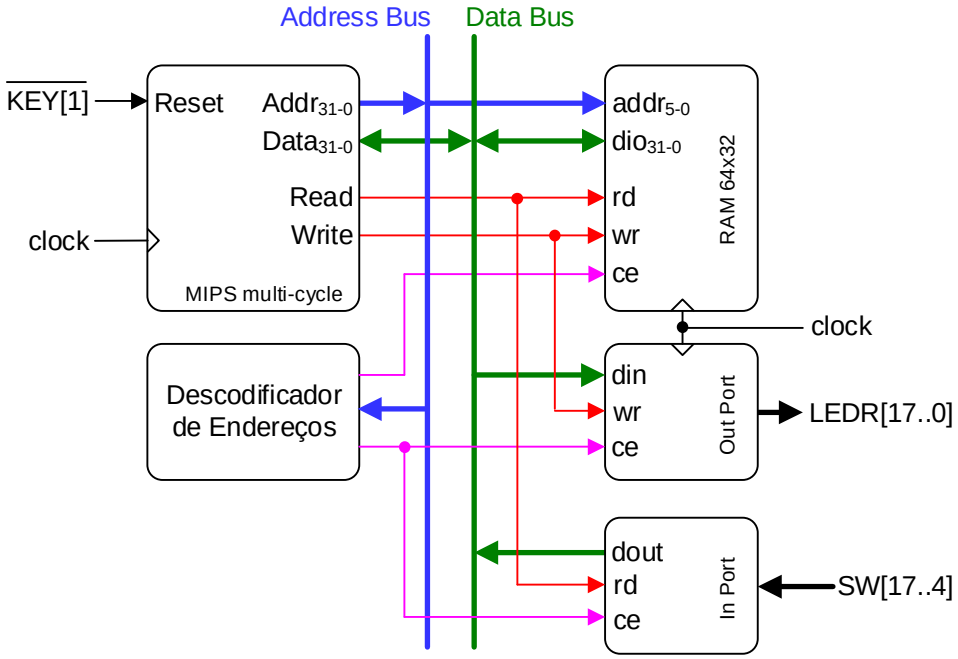

Figura 4. Blocos funcionais do sistema computacional, a instanciar no *top-level* do projeto, com interligação a recursos da placa FPGA (**LEDR** e **SW**).

Guião

Parte I

1. Retome o projeto iniciado na aula anterior e acrescente-lhe os módulos VHDL para o porto de entrada e o porto de saída.

2. Usando o simulador *University Program VWF*, simule funcionalmente os módulos correspondentes ao porto de entrada e ao porto de saída.

3. Implemente em VHDL o módulo decodificador de endereços, de acordo com a descrição feita anteriormente (apesar de ser um bloco muito simples, e por uma questão de organização concetual, sugere-se que acrescente o módulo descodificador e que não implemente diretamente a sua função no *top-level*).

4. Faça todas as ligações dos três módulos descritos nos pontos anteriores no *top-level* do projeto.

5. Com o *top-level* do projeto concluído podem agora ser feitos testes de funcionamento, executando pequenos programas que permitam verificar o correto funcionamento do sistema. Para isso, traduza para *Assembly*, codifique e teste os seguintes programas:

a) Enviar para os **LEDR** da placa o padrão: **1100110011001100**.

```
void main(void)
{
  int *outPort = 0x100;

  *outPort = ???; // Completar
  while(1);
}
```

b) Ler continuamente o valor dos **SW** e enviar o resultado para os **LEDR**.

```
void main(void)
{
  int *inPort  = 0x100;
  int *outPort = 0x100;

  while(1)
  {
    *outPort = ???;   // Completar
  }
}
```

c) Ler continuamente o valor do **SW[4]** e, se o valor lido for **0** apagar todos os **LEDR**; caso contrário acender todos os **LEDR**.

```
void main(void)
{
  int *inPort  = 0x100;
  int *outPort = 0x100;

  while(1)
  {
    if(*inPort & ???) // Completar
      *outPort = ???; // Completar
    else
      *outPort = ???; // Completar
  }
}
```

Parte II

Com pequenas alterações na estrutura existente podem ser incluídos outros periféricos. Nesta parte do trabalho pretende-se incluir um contador de 32 bits, que possa ser usado para a contagem de tempo.

1. Implemente em VHDL um módulo contador de 32 bits que possa ser interligado como um periférico ao sistema implementado na parte 1. Este módulo deve apresentar os seguintes sinais de interface: **`clk`** (clock), **`ce`** (chip enable), **`wr`** (write), **`rd`** (read) e **`data`** (barramento de dados de 32 bits). O contador deve estar em contagem permanente e os sinais de interface devem ter o seguinte comportamento: **`ce`** e **`wr`** ativos simultaneamente, faz o *reset* síncrono do contador; **`ce`** e **`rd`** ativos simultaneamente, coloca no barramento de dados de saída o valor atual do contador; quando não está a ser feita uma operação de leitura, a saída de dados do módulo deve estar em alta impedância.

2. Usando o simulador *University Program VWF*, simule funcionalmente o módulo contador que implementou no ponto anterior.

3. A inclusão do módulo contador obriga a alterar o descodificador de endereços implementado anteriormente, uma vez que é necessário atribuir um endereço a esse módulo. A Tabela 2 mostra o novo mapa de endereços.

Tabela 2. Novo mapa de endereços com a inclusão do módulo contador.

Dispositivo	Dimensão (bytes)	Endereço Inicial	Endereço Final	Nº de bits de endereço
Memória RAM	256	0x00000000	0x000000FF	8 (A_7-A_0)
Porto de Saída	4	0x00000100	0x00000103	2 (A_1-A_0)
Porto de Entrada	4	0x00000100	0x00000103	2 (A_1-A_0)
Contador	4	0x00000200	0x00000203	2 (A_1-A_0)

Altere o descodificador de endereços de modo a gerar mais um sinal de seleção para ligar ao módulo contador (sugestão, use 2 bits do barramento de endereços (**A_9** e **A_8**) e implemente como descodificador de endereços um *decoder* 2:4).

4. O programa seguinte permite verificar o funcionamento do módulo contador.

```
void main(void)
{
  int *outPort = ???;  // Completar
  int *counter = ???;  // Completar

  while(1)
  {
    *counter = 0;
    while(*counter < 1000);
    *outPort = ~(*outPort);
  }
}
```

Supondo que o *clock* do sistema é 1 kHz, qual o resultado que espera observar nos LEDs vermelhos que estão ligados ao porto de saída do sistema?

5. Traduza o programa anterior para *assembly*, codifique-o e teste-o no sistema (não se esqueça de alterar a instanciação do divisor de frequência de modo a ter um *clock* no sistema de 1 kHz).

PDF gerado em 08/11/2018