

µProcessador 3 Banco de Registradores e ULA

O banco de registradores é apenas um bloco onde estão os registradores de uso geral. *Ele se comporta como uma memória*: indicamos o número do registrador a ser lido e o valor guardado neste registrador aparece na saída. A escrita é similar, mas exige um *enable* para sincronização.

Neste capítulo da nossa saga: muita explicação e nem tanta diversão. Ao final, como criar um projeto com vários arquivos fonte *.vhd*.

VHDL Sequencial

Nas práticas anteriores, vimos apenas *portas lógicas*, ou seja, circuitos combinacionais. Para trabalhar com *circuitos sequenciais*, que possuem *flip-flops*, registradores, máquinas de estado e o escambau, os comandos VHDL são diferentes.

Um registrador é um bloco que guarda dados. São apenas vários *flip-flops* em paralelo, portanto ele precisa de um *clock*, deve ter um *reset* (ou *clear*) e é bom que tenha um *write enable*. Sua interface não surpreende:

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

entity reg8bits is
  port( clk      : in std_logic;
        rst      : in std_logic;
        wr_en    : in std_logic;
        data_in  : in unsigned(7 downto 0);
        data_out : out unsigned(7 downto 0)
  );
end entity;
```

Já a arquitetura traz diversas novidades:

```
architecture a_reg8bits of reg8bits is
  signal registro: unsigned(7 downto 0);
begin

  process(clk,rst,wr_en) -- acionado se houver mudança em clk, rst ou wr_en
  begin
    if rst='1' then
      registro <= "00000000";
    elsif wr_en='1' then
      if rising_edge(clk) then
        registro <= data_in;
      end if;
    end if;
  end process;

  data_out <= registro; -- conexao direta, fora do processo
end architecture;
```

O bloco construtor é o processo: ali colocamos a descrição sequencial de um circuito com *flip-flops*. Seguindo a palavra “process” colocamos a *lista de sensibilidade*, ou seja, indicamos quais são os sinais cuja alteração deve ser respondida pelo processo.

Note que usamos aqui uma construção *if-then*. Estes comandos são exclusivos do “process”. A construção *when-else* que usamos anteriormente é proibida nesta seção.


Registrador Padrão

Usamos a *borda de subida* (rampa ou transição de 0 para 1) para habilitar o registrador¹. Para isso, usamos a função “*rising_edge*” no sinal, que detecta a rampa baseado nas suas propriedades.

```
if rising_edge(clk) then
    registro <= data_in;
end if;
-- obrigatoriamente não tem else aqui
```

Mas este *if* não é bem um *if* comum de programação: *ele é um comando para inferir flip-flops!*² As comparações apenas determinam o funcionamento dele.

É necessário chamar a atenção para essa ressalva.

	O <i>if-then</i> só deve ser usado com <i>clock</i>, <i>clock enable</i> ou <i>reset</i>! Até dá pra colocar condições extras (veremos mais adiante), mas o objetivo é sempre construir um registrador ou um <i>flip-flop</i> . Se na sua construção <i>if-then-else</i> não aparecer um “ <i>rising_edge()</i> ”, você provavelmente está fazendo besteira.
---	---


Não é que “não pode,” mas quase todos os alunos que tentam usar cometem alguns erros conceituais e distrações³ que custam dezenas de horas de depuração. Sim, é sério.

Devido à forma como as FPGAs⁴ são construídas, usa-se um *signal de clock global*, ou seja, o circuito inteiro tem um único *clock*, curto-circuitado em todos os componentes, fazendo com que todos transicionem simultaneamente.

Para fazer uma sequência de operações em vários registradores, usamos um *clock enable*. Quando ele estiver em 0, o *clock* será ignorado.

```
elsif wr_en='1' then          -- este é o clock enable
    if rising_edge(clk) then
```

Com ele, podemos dizer exatamente quando o registrador deve ser escrito e quando não deve.

	Faça um registrador de dezesesseis bits e construa um <i>testbench</i> adequado (dicas a seguir).
---	--

Testbench com Clock

No *testbench* de um circuito sequencial são necessários tipicamente um sinal de *clock* e um de *reset* além dos casos de teste nas entradas. Um sinal típico de *clock* é gerado por um processo como:

```
process          -- sinal de clock
begin
    clk <= '0';
    wait for 50 ns;
    clk <= '1';
    wait for 50 ns;
end process;
```

- 1 Os *flip-flops* usados nos circuitos sempre são bordas de subida; ao usar descida ou nível (*latch*), o circuito final provavelmente vai ser gerado com gambiarras que podem ser indesejáveis.
- 2 A inferência de *flip-flops* em VHDL se dá pela especificação incompleta de valores: se o valor muda quando “algo” acontece mas não muda em outras situações, entende-se que nestas ele deve continuar o mesmo, ou seja, deve ser armazenado. Mas você não precisa entender isso, apenas siga a receita de bolo respeitosamente.
- 3 Duas coisas são muito frequentes: esquecer-se de um “*else*” final para os casos omissos; e um encadeamento muito longo de comparações que fica incompleto em certos casos e é difícil de seguir quando se lê o código.
- 4 *Field Programmable Gate Array*, componentes nos quais podemos programar, construindo fisicamente os circuitos especificados em VHDL.

O de *reset* é um pulso inicial e o resto '0', ou seja, um **wait**; final que paralisa o processo. P. ex.:

```
process      -- sinal de reset
begin
    rst <= '1';
    wait for 100 ns;
    rst <= '0';
    wait;
end process;
```

Podemos fazer processos separados para estes sinais, facilitando o reaproveitamento. Mas também temos que cronometrar a execução do nosso teste, usando um processo e um sinal adicionais, similar ao *reset*. Veja a seguir o trecho padrão para implementação disto:

```
architecture ateste_tb of teste_tb is
    component teste is      -- aqui vai seu componente a testar
    [...]
    end component;

    -- 100 ns é o período que escolhi para o clock
    constant period_time : time      := 100 ns;
    signal  finished      : std_logic := '0';
    signal  clk, reset    : std_logic;
begin
    uut: teste port map [...]; -- aqui vai a instância do seu componente

    reset_global: process
    begin
        reset <= '1';
        wait for period_time*2; -- espera 2 clocks, pra garantir
        reset <= '0';
        wait;
    end process;

    sim_time_proc: process
    begin
        wait for 10 us;      -- <== TEMPO TOTAL DA SIMULAÇÃO!!!
        finished <= '1';
        wait;
    end process sim_time_proc;

    clk_proc: process
    begin
        -- gera clock até que sim_time_proc termine
        while finished /= '1' loop
            clk <= '0';
            wait for period_time/2;
            clk <= '1';
            wait for period_time/2;
        end loop;
        wait;
    end process clk_proc;

    process      -- sinais dos casos de teste (p.ex.)
    begin
        wait for 200 ns;
        wr_en <= '0';
        data_in <= "11111111";
        wait for 100 ns;
        data_in <= "10001101";
        [...]
        wait;
    end process;
end architecture ateste_tb;
```

É importante atentar ao *tempo total de simulação*, definido no processo `sim_time_proc`, porque vai ser necessário alterar ele mais à frente no projeto. E nunca esqueça de terminar seus testes com um comando `wait`; final, senão ele fica simulando “para sempre” e nunca acaba.

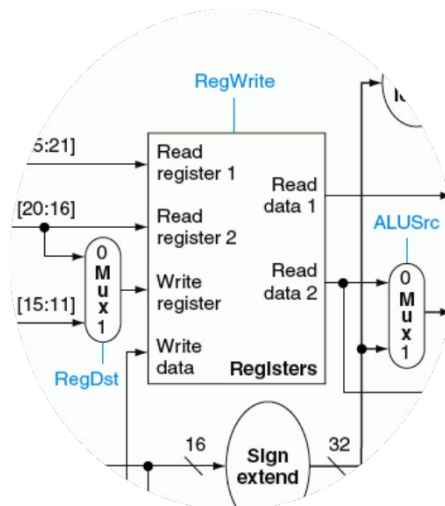
Especificação do Banco de Registradores (Provisório)

O banco *sempre* está fazendo a leitura de dois registradores, portanto temos dois barramentos de entrada dizendo o número dos registradores que desejamos ler. Por exemplo, numa instrução `add $5,$6,$7` iremos ler simultaneamente os registradores \$6 e \$7.

- Entradas (pinos):
 - Seleção de quais registradores serão lidos (2 barramentos)
 - Barramento de dados para escrita (o valor a ser escrito)
 - Seleção de qual registrador será escrito
 - write enable* para habilitar a escrita apenas no momento correto (é o *clock enable* dos registradores)
 - clk* (é o *clock* geral do processador)
 - rst* (*reset*, zera todos os registradores)
- Saídas (pinos):
 - Barramentos com os dados dos registradores lidos (2 barramentos)

Reconhece a figura a seguir?

(Patterson-Hennessy)



Especificações adicionais:

- Faça oito registradores de 16 bits cada;
- O registrador zero possui a constante zero e não pode ser alterado.



Construa o banco de registradores especificado acima e faça um *testbench* adequado.

É obrigatório usar múltiplas fontes VHDL: um arquivo “.vhd” para um registrador e outro arquivo instanciando oito registradores (este é o banco). *Veja o apêndice ao final deste arquivo.*

Ligando o Banco a uma ULA

Faça as ligações como os circuitos vistos no livro, ou seja:

- ligue a saída da ULA na entrada de dados do banco;
- ligue uma saída do banco direto numa das entradas da ULA;
- ligue a outra saída do banco num MUX, e a saída do MUX na ULA;
- ligue um pino de entrada *top level* na outra entrada do MUX (será a entrada de uma

- constante externa);
- ligue pinos de entrada em clk, rst e write enable;
- ligue um pino de saída extra à saída da ULA para poder debugar no top level.

Detalhes:

- Obrigatoriamente criem no mínimo um bloco para a ULA, um bloco para o banco e usem ambos integrados no arquivo *top level* (ou seja, serão *três ou mais* arquivos .vhd, fora os *testbenches*).
- Obrigatoriamente façam um *reset* explícito de todos os registradores/flip-flops no início da simulação.
- Não é necessário testar as outras operações da ULA: supomos que vocês fizeram isso direito na prática anterior.

Apêndice: Múltiplos Arquivos Fonte

Para trabalhar com vários arquivos “.vhd”, siga a ideia do *testbench*: simplesmente declare o que você quer usar como um componente no outro arquivo. No VHDLPlus, basta os arquivos estarem no projeto; no terminal, rode *ghdl -a* para a análise *em cada um dos fontes*. Daí pra frente é igual.

Vamos rever o arquivo (besta) da porta E do lab #1, “porta.vhd”:

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

entity porta is
    port( in_a  : in std_logic;
          in_b  : in std_logic;
          a_e_b : out std_logic
    );
end entity;

architecture a_porta of porta is
begin
    a_e_b <= in_a and in_b;
end architecture;
```

Aí vamos fazer um outro, igualmente besta, que vai usar *três portas E*, uau! Vamos associar três portas de 2 entradas pra fazer uma porta E de 4 entradas. Batizemo-lo de “e_4_entradas.vhd”.

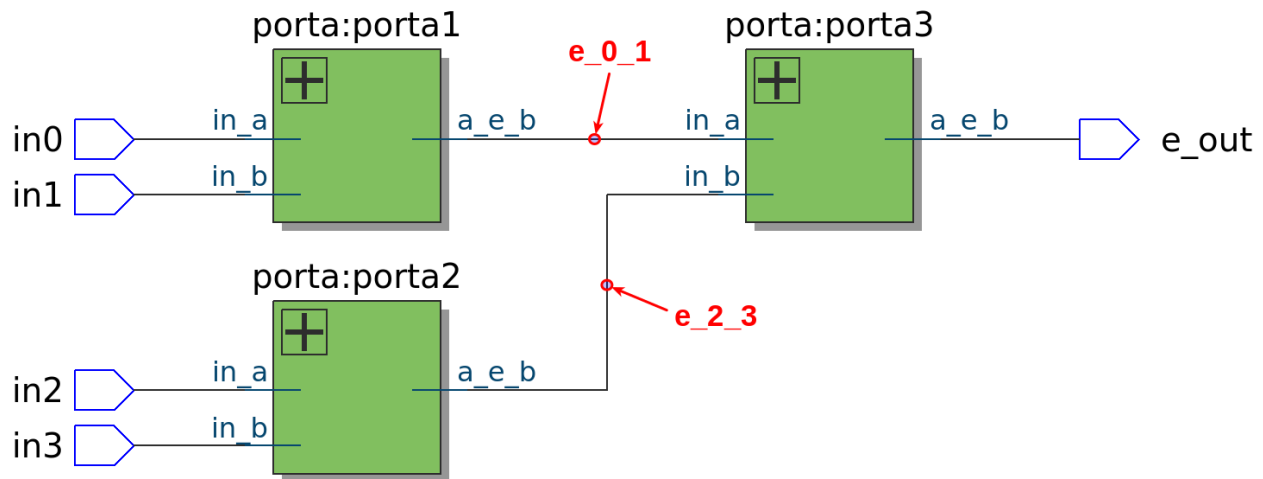
```
library ieee;
use ieee.std_logic_1164.all;

entity e_4_entradas is
    port( in0,in1,in2,in3: in std_logic;
          e_out: out std_logic
    );
end entity;

architecture a_e_4_entradas of e_4_entradas is
    component porta is
        port( in_a  : in std_logic;
              in_b  : in std_logic;
              a_e_b : out std_logic
        );
    end component;
    signal e_0_1, e_2_3: std_logic;
begin
    porta1: porta port map(in_a=>in0,in_b=>in1,a_e_b=>e_0_1);
    porta2: porta port map(in_a=>in2,in_b=>in3,a_e_b=>e_2_3);
    porta3: porta port map(in_a=>e_0_1, in_b=>e_2_3, a_e_b=>e_out);
end architecture;
```

Observe que apenas inserimos a interface do componente “porta” ali na arquitetura e o compilador se encarrega do resto. (Na linha de comando, analise ambos os arquivos, dentro da mesma pasta, e rode a entidade do *testbench* final com, digamos, *ghdl -r e_4_entradas_tb --wave=result.ghw* ou algo assim).

Criei três portas E distintas: porta1, porta2 e porta3. Observe que com o “port map” nós conseguimos fazer a ligação entre os componentes sem gastar uma linha de código explícito. O circuito final é este:



Os sinais indicados em vermelho, **e_0_1** e **e_2_3**, servem como “cola” para ligar as instâncias.

Por fim, lembrem-se de que, se encher o saco ficar selecionando sinais no *gtkwave*, podemos guardar a configuração de uma tela (sinais e zoom, essencialmente) usando o menu *File => Write Save File* para gravar a configuração de visualização (.gtkw).