

**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*.

Por favor cuidado com a Internet e com o ChatGPT -- normalmente eles geram código que vocês não dominam, com comportamentos inusitados difíceis de perceber. Se eu ver coisa estranha nos fontes, em especial if-then's, a nota será zero.

**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 registrador<sup>1</sup>. 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!*<sup>2</sup> As comparações apenas determinam o funcionamento dele.

É necessário chamar a atenção para essa ressalva.

	<b>O <i>if-then</i> só deve ser usado nesta disciplina para criar um registrador!</b> Ele sempre deve ter clock, clock enable ou reset, talvez mais alguma condição extra (veremos mais adiante). 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 distrações<sup>3</sup> que custam dezenas de horas de depuração. Sim, é sério.

Devido à forma como as FPGAs<sup>4</sup> 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 <b>dezesesseis</b> 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:

- <sup>1</sup> 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.
- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> *Field Programmable Gate Array*, componentes nos quais podemos programar, construindo fisicamente os circuitos especificados em VHDL.

```

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

```

O sinal de *reset* é um pulso inicial e o resto '0', ou seja, há um **wait** final que paralisa o processo. Por 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";
    [...]                                   -- outros casos
    wait;                                  -- <== OBRIGATÓRIO TERMINAR COM WAIT; !!!
end process;
end architecture ateste_tb;

```

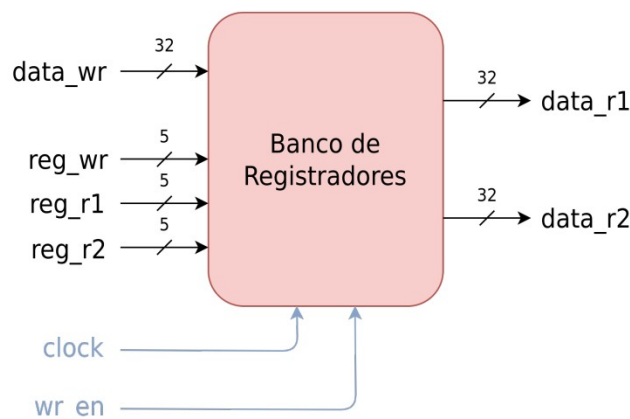
É importante atentar ao *tempo total de simulação*, definido no processo `sim_time_proc`, porque vai ser necessário alterar ele mais à frente no projeto. E nunca esqueça de terminar seus testes com um comando `wait`; final, senão ele fica simulando “para sempre” e nunca acaba.

## Pinagem do Banco de Registradores

Verifique no sorteio enviado por e-mail as características do seu processador. O mais importante é se há ou não um acumulador.

Se não houver acumulador, as instruções são ortogonais e o banco é similar ao visto em sala, como na figura abaixo.

(Do meu PDF de  $\mu P$   
ciclo único)



Neste caso, 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 R5,R6,R7` iremos ler simultaneamente os registradores R6 e R7.

Já no caso de haver acumulador, o banco *sempre* estará fazendo a leitura de apenas um dos seus registradores, uma vez que a soma obrigatoriamente utiliza o acumulador e mais algum registrador: em `ADD A,R3` (que faz  $A \leftarrow A + R3$ ), ambos A e R3 devem ser lidos. O acumulador é um registrador à parte, *fora* do banco.

Portanto a interface é:

- Entradas (pinos):
  - Seleção de qual registrador será lido (1 ou 2 barramentos, lê 1 ou 2 registradores)
  - Seleção de qual registrador será escrito
  - Barramento de dados para escrita (o valor a ser escrito no registrador)
  - *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, também é global como o *clock*)
- Saídas (pinos):
  - Barramentos com os dados dos registradores lidos (2 barramentos ou 1)

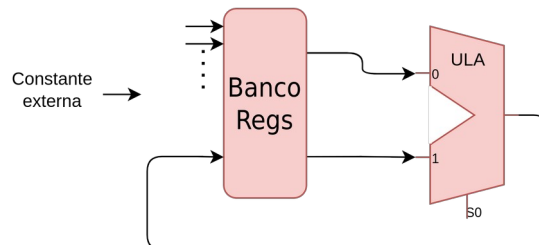


Construa o banco de registradores especificado para a equipe e faça um *testbench* adequado. Cada registrador tem 16 bits.

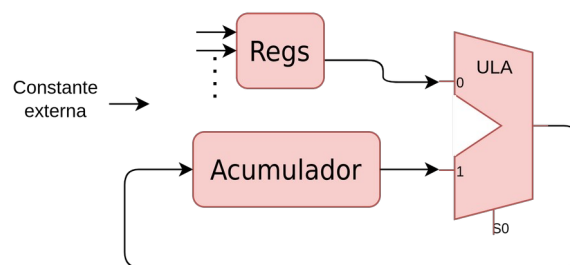
É obrigatório usar múltiplas fontes VHDL: um arquivo “.vhd” para um registrador e outro arquivo instanciando os registradores (este é o banco). Veja o apêndice ao final deste arquivo.

### Ligando os Registradores a uma ULA

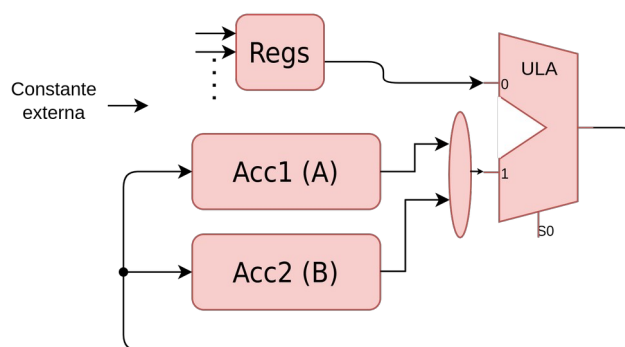
Se não houver acumulador (ISA ortogonal), fica muito parecido com o meu PDF:



Se houver acumulador, há diferenças. O acumulador é um registrador à parte do banco, que é usado para operações com a ULA. Se houver acumulador, o resultado da ULA *sempre* será gravado no acumulador; e o acumulador *sempre* fornecerá um dos valores da operação. Veja a figura abaixo:



Se houver dois acumuladores, isso significa que deveremos especificar qual dos dois será usado, para cada instrução. Ou seja, ou A é uma fonte e o destino de uma operação, ou então B será uma fonte e o destino desta operação. Fica assim:



Observe que há diversos multiplexadores que não estão ilustrados nas figuras e dependem das operações sorteadas: se houver LD de constante, ela deve entrar no banco via um MUX; instruções MOV de cópia de registrador também exigem MUX; etc. Responda às perguntas seguintes e entenda as consequências para o circuito:

- Existe um registrador com constante zero? Isso significa que a carga de uma constante é feita com a soma da constante com zero?
- É vital poder copiar o dado guardado no acumulador para um registrador do banco e vice-versa. Precisamos de MUXes para isso? Se sim, onde?

- Como carregar uma constante para um registrador do banco? Qual a instrução ou instruções assembly? É preciso mudar algo no circuito?

Não esqueça de ligar pinos de entrada em *clk*, *rst* e *write enable*. 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.

## Tarefa para Entregar: Banco + ULA

Desta vez *eu vou querer ver funcionando na sala de aula*, logo depois da entrega, obrigatoriamente com todos os membros da equipe presentes.

## Apêndice: Múltiplos Arquivos Fonte [replay do lab anterior]

Para trabalhar com vários arquivos “.vhd”, siga a ideia do *testbench*: simplesmente declare o que você quer usar como um componente no outro arquivo. Preferencialmente mantenha os arquivos na mesma pasta (e um projeto único por pasta). 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;
```

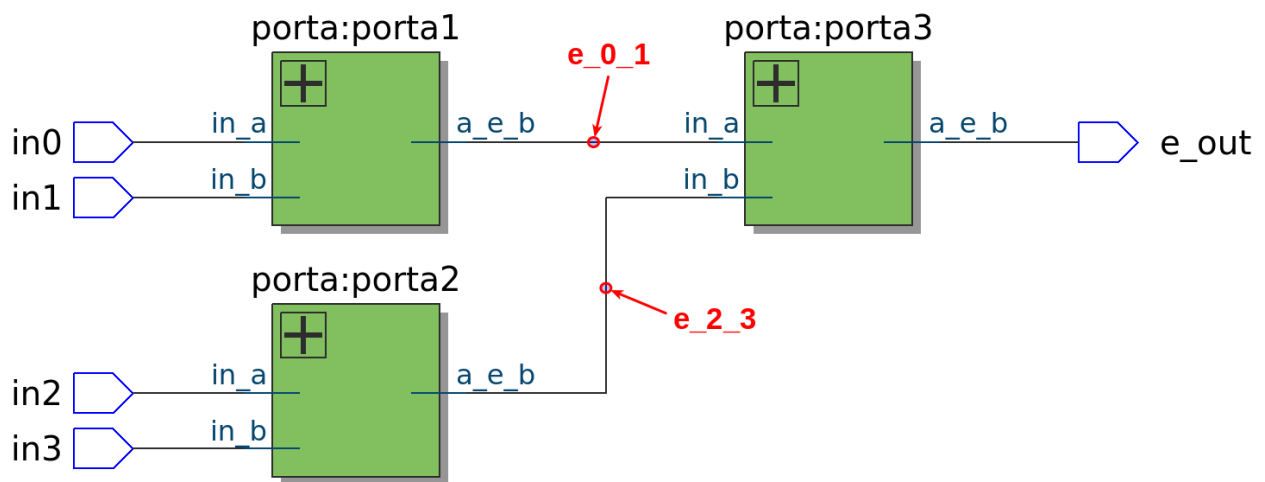
```

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).