µProcessador 4 Unidade de Controle Rudimentar

Hoje é o dia do esqueleto! Vamos fazer um programa armazenado em ROM ser percorrido por um PC e executar *jumps* incondicionais. Outra hora adicionaremos ULA e Banco de Registradores.

ROM em VHDL

Para fazer uma ROM em VHDL vamos usar um modelo como o que segue. Neste caso, é uma ROM de 128 endereços, com dados de 12 bits em cada endereço.

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

entity rom is
   port( clk      : in std_logic;
         endereco : in unsigned(6 downto 0);
         dado     : out unsigned(11 downto 0)
   );
end entity;

architecture a_rom of rom is
   type mem is array (0 to 127) of unsigned(11 downto 0);
   constant conteudo_rom : mem := (
      -- caso endereco => conteudo
      0  => "000000000010",
      1  => "100000000000",
      2  => "000000000000",
      3  => "000000000000",
      4  => "100000000000",
      5  => "000000000010",
      6  => "111100000011",
      7  => "000000000010",
      8  => "000000000010",
      9  => "000000000000",
      10 => "000000000000",
      -- abaixo: casos omissos => (zero em todos os bits)
      others => (others=>'0')
   );
begin
   process(clk)
   begin
      if(rising_edge(clk)) then
         dado <= conteudo_rom(to_integer(endereco));
      end if;
   end process;
end architecture;
```

Os dados tabelados em "conteudo_rom" são apenas ilustrativos.

Note que *esta ROM é síncrona!*[1] Isto significa que é preciso dar um *clock* nela para que ela leia os dados, ou seja, só quando houver rampa de subida no *clock* é que teremos uma resposta à saída. Para fazer uma ROM assíncrona, basta retirar o *clock* e o *process* e deixar a arquitetura como:

```
begin
         dado <= conteudo_rom(to_integer(endereco));
end architecture;
```

1 Dentro duma FPGA as memórias são tipicamente síncronas.

▶	Verifique no "*Sorteio de Microprocessadores para as Equipes*" a largura dos dados da ROM especificadas para a sua equipe de laboratório. Verifique também se a ROM é síncrona ou assíncrona. Construa uma ROM de acordo. Faça um *testbench* simples e se assegure de que está tudo ok.

Máquina de Estados

Vamos evitar problemas? Vamos evitar problemas sim.

Então vamos já começar com uma máquina de dois estados. O primeiro *clock* vai fazer *fetch*, o segundo vai fazer *decode/execute* e é isso aí. Depois a gente incrementa a coisa e faz mais estados.

Mesmo neste laboratório com circuito ainda pequeno, seria complicado fazer as coisas em ciclo único (leia-se: só sairia com chuncho, desorganização e coisa feia), especialmente por conta da atualização correta do PC.

▶	Faça uma máquina de estados com dois estados. Para isso, use o esquema já visto de um registrador e adapte-o para usar só 1 bit (ver logo abaixo). Faça mais um *testbench* simples só para este ".vhd" e se assegure novamente de que está tudo ok.

Pra fazer isso, use um simples *flip-flop* T, ou seja, aquele que troca de estado a cada *clock*. Veja o trecho da arquitetura, alterada a partir de um registrador de 1 bit:

```
elsif rising_edge(clk) then
   estado <= not estado;
end if;
```

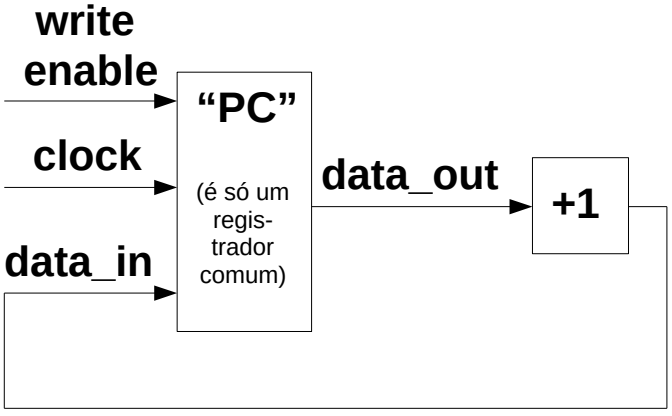
Não esqueça do *reset*. Teste separadamente esse *flip-flop* T pois essa é uma atitude saudável.

Dica: se os nomes iguais estiverem se trombando, é comum usar os sufixos _i, _o e _s para diferenciar pinos de entrada, saída e sinais internos, como "dado_i" e "dado_o" para pinos do bloco e "dado_s" para o *signal* interno, por exemplo.

Contador de Programa

O nosso PC *(Program Counter)* vai ser só um registrador, sem contagem interna, assim como nos circuitos vistos no livro. A "contagem" vai ser dada por um circuito externo somador com 1.

Eu sei que tem na Internet os esquemas para fazer um contador em VHDL mas... **Não use um contador típico VHDL.** Estes circuitos incrementam o contador a cada *clock* e isso não vai prestar para a gente. Ao invés disso, construa um VHDL que faz como na figura:



▶	Faça o PC funcionar apenas contando para a frente. O PC é só um registrador comum, igual ao do laboratório #3. Não precisa ainda usar a máquina de estados. Crie outro módulo (uma proto-unidade de controle) que simplesmente adiciona 1 no valor de saída do PC e conecta o resultado desta soma de volta à entrada do PC. Por enquanto pode deixar *wr_en=1* sempre, então todo *clock* vai incrementar. Faça outro *testbench* simples só para este PC e veja se está tudo ok, como já virou um hábito. Se você está pulando estes *testbenches*, acho que você pode se arrepender logo logo.

E mais:

▶	Conecte o PC à ROM! Basta ligar a saída do PC direto na entrada de endereços da ROM. Coloque a saída da ROM num pino do *top-level* e veja os conteúdos da memória serem apresentados em ordem, a partir do endereço zero, na tela do gtkwave. Sorria, isso já mostra o básico de um processador!

Unidade de Controle com Jump

Okay, chega de brincadeira.

▶	Inclua a máquina de estados de 1 bit no módulo da unidade de controle. O circuito deverá fazer a leitura da ROM no estado 0 (*fetch*) e a atualização do PC no estado 1 (*decode/execute*). Ele vai ignorar solenemente as instruções por enquanto. Faça um *testbench* e olhe bem para as formas de onda resultantes. Pense um pouco.

Lembre-se: *não use if-then,* a não ser para criar um *flip-flop* ou registro (cf. lab #3). Se você está colocando mais condições dentro do *if*, é provável que esteja fazendo besteira. Use *when-else* para construir qualquer lógica, que deve ser colocada fora do "process."

Agora vamos implementar o *nop* e o *jump*.

▶	Crie uma codificação para a instrução *jump*. Pode ser parecida com a do formato J do RISC-V se você quiser. Implemente (veja abaixo uma sugestão). Restrição: deve ser usado endereço absoluto (a constante dentro da instrução nos dá o endereço destino do salto) e não endereço relativo (como no *branch* do RISC-V, que dá um valor a ser somado no PC). Agora faça um conjunto decente de testes. Faça a codificação das instruções em binário e coloque no "rom.vhd", de preferência saltando alguns endereços para a frente e também fazendo um *loop*. Confira se as operações ocorrem no estado esperado.

Para decodificar instruções, basta separar os bits do opcode em sinais parciais e fazer comparações simples. Veja o trecho que usei:

```
architecture a_un_controle of un_controle is
   signal opcode: unsigned(3 downto 0);
begin
   -- coloquei o opcode nos 4 bits MSB
   opcode <= instr(11 downto 8);
   -- meu jump: opcode 1111
   jump_en <=  '1' when opcode="1111" else
               '0';
```

Não custa lembrar: use *when-else*, não use *if-then* porque *if-then* costuma dar treta, além de ser

usado só em blocos “process”.

Se quiser ser chique, verifique se a instrução é desconhecida (diferente de *jump* e *nop*) e neste caso paralise o PC e gere um sinal de erro (exceção de *opcode*). Mas só se você quiser.

▶	Estime o número de *clocks* necessários para seu programa. Por exemplo, se uma instrução leva dois clocks para ser executada, então ele deverá executar dez instruções em vinte *clocks*. Confira isso como teste de sanidade.

Algo que pode ajudar bastante os próximos labs: desenhe no caderno, *sem olhar no gtkwave,* as formas de onda esperadas para a execução de três instruções seguidas, sendo a segunda um jump. Coloque clock, os write enables e os dados de saída de PC e ROM. Compare com o gtkwave -- se estiver diferente, encontre a explicação.

Para entrega, me interessam apenas os arquivos fontes VHDL. Entregue-os dentro de um arquivo .zip; pode ser apenas a última versão deste lab, incluindo *jumps* e testes com um programa simples. Este laboratório não vai ser visto presencialmente pelo professor, fica apenas com a entrega eletrônica.