Hoje é o dia do esqueleto! Vamos fazer um programa armazenado em ROM ser percorrido por um PC e executar *jumps* incondicionais. No próximo Lab adicionaremos a ULA e o 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 8 bits em cada endereço (os dados tabelados em "***conteudo_rom***" são apenas ilustrativos). Os dados desta memória são as instruções do programa.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity rom is
   port( clk      : in std_logic;
         endereco : in unsigned(15 downto 0);
         dado     : out unsigned(7 downto 0)
   );
end entity;

architecture a_rom of rom is
   type mem is array (0 to 127) of unsigned(7 downto 0);
   constant conteudo_rom : mem := (
      -- caso endereco => conteudo (apenas exemplo. Altere para refletir o
      -- programa solicitado)
      0  => "00000010",
      1  => "10000000",
      2  => "00001111",
      3  => "00000000",
      4  => "10000000",
      5  => "00000010",
      6  => "10100011",
      7  => "00000010",
      8  => "00000010",
      9  => "00011110",
      10 => "00000000",
      -- abaixo: casos omissos => (zero em todos os bits a decodificação do pc
      --é parcial dos 7 least significat 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;
```

Note que diferente do que estamos considerando no livro texto, *esta ROM é sincrona!*[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.

1 Dentro duma FPGA as memórias são tipicamente síncronas.

▶	Compare as características importantes da ROM para o caso da ISA MIPS (as larguras dos barramentos de dados e de endereços). Construa uma ROM de acordo, levando em conta que 128 instruções é mais do que suficiente para o nível de complexidade do algoritmo que teremos tempo de alcançar na disciplina. Ou seja, a decodificação da instrução pode ser similar ao tipo J. Para mais detalhes, verifique o material das aulas teóricas. 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 sai 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 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;
```

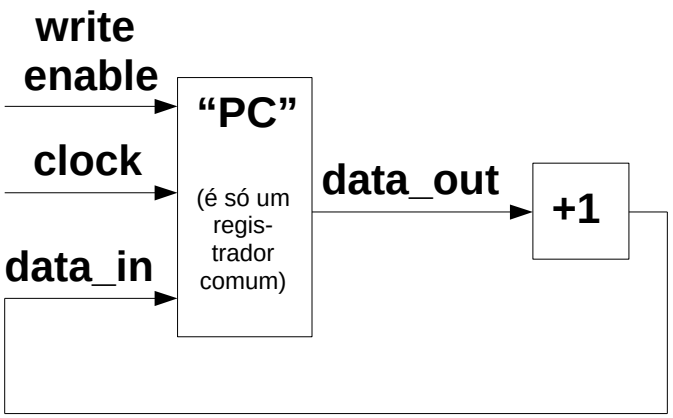
Não esqueça do *reset*. Teste separadamente esse *flip-flop* T pois essa é uma atitude saudável.

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 a ROM ao PC! 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 já partir do estado 0 (*fetch*), e a atualização do PC no estado 1 (*decode/execute*). Se não houver um registrador adicional para armazenar a instrução, o roteamento do argumento via jump enable deve permanecer no estao 1. Ou seja, o destino do salto codificado nos bits menos significativos da instrução deve permanecer na entrada do registrador PC. 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*. Restrição: deve ser usado endereço absoluto (como no *j* do MIPS) e não endereço relativo (como no *branch* do MIPS). Agora faça um *testbench* bem pensado, decente. Faça a codificação das instruções em binário e coloque no "rom.vhd", executando alguns *NOPs* depois saltando alguns endereços para a frente, e depois alguns para trás 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. Como exemplo, veja o trecho que usei:

```
architecture a_un_controle of un_controle is
   signal opcode: unsigned(1 downto 0);
begin
   -- coloquei o opcode nos 3 bits MSB
   opcode <= instr(7 downto 6);
   -- meu jump: opcode 11
   jump_en <=  '1' when opcode="11" 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, para dez instruções, ele deverá executar em vinte *clocks*. Confira isso como teste de sanidade

Com tudo funcionando no simulador conforme o especificado deve ser feito o envio do pacote .zip em GHDL (mais infos no Moodle).

Baseado no projeto exemplo de circuitos síncronos para a placa DE10Lite, inclua o ROM-PC-UC no projeto do Quartus. A demonstração do funcionamento deve ser feita apresentando nos displays de sete segmentos HEX5 e HEX4 o endereço da instrução em decimal, e nos displays HEX1 e HEX0 o código da instrução. Mantenha o clock em 0,5 e 10 Hz conforme o último laboratório.