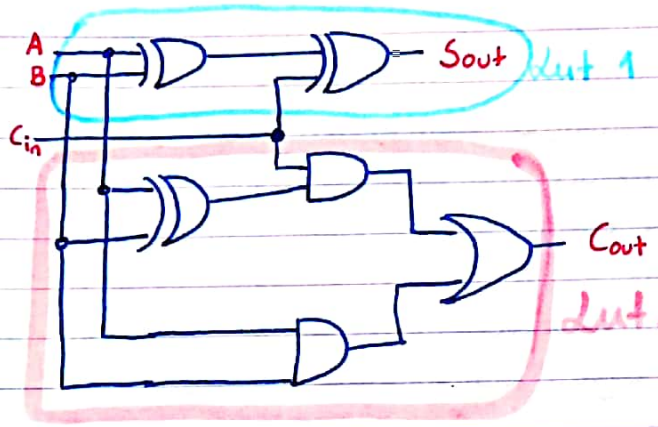
exercício 1

-> Para implementar um full adder seriam necessárias, no mínimo 2 lut's 3:1. A primeira, teria como saída a soma dos bits, a segunda teria como saída o transporte de saída. Ambas as luts teriam 3 entradas, A e B (os bits que queremos somar) e Cin (o transporte de entrada).



opção B

exercício 2

-> Um bloco lógico implementado baseado em LUTs, implementa todas as 2^{2^m} funções lógicas de m entradas ($m \geq 2$). Um bloco lógico, por exemplo, baseado em multiplexers pode implementar funções de várias entradas, sem necessáriamente, implementar todo o conjunto de funções de um certo valor m de entradas.

-> Uma função lógica implementável numa lut 4:1, implementa $2^{2^4} = 2^{16}$ funções lógicas, logo só poderá ser implementada num conjunto de luts que implementem 2^{16} ou mais funções lógicas.

A) $2^4 \times 2 = 2^5$ -> inferior

B) $2^8 \times 2 = 2^9$ -> inferior

C) 2^{32} // ✓

Exercício 3

-> Caso liguemos o botão KEY(0) diretamente à saída LEDR(0) então o LEDR(0) acende quando o botão KEY(0) não está premido pois os Leds são ativos a '1', e quando pressionado, o botão KEY transmite '0'.

-> opção A

Exercício 4

-> A construção entity do VHDL permite descrever a interface de um módulo.

-> opção B

Exercício 5

-> Para converter um inteiro (integer) ~~diretamente~~ para std_logic_vector, devemos usar a função "to_unsigned", do seguinte modo:

```
use IEEE.NUMERIC_STD.all;
...
my_slv <= std_logic_vector(to_unsigned(my_int, my_slv'length));
```

-> não pode ser diretamente, logo opção D

Exercício 6

-> Em VHDL um porto corresponde a um sinal da interface de um módulo. opção B

Exercício 7

-> Nem todos os circuitos são sintetizáveis, existem descrições que podem ser simuladas, no entanto o circuito não pode ser sintetizado por falta de um elemento do tipo descrito no mundo real.

-> "wait for" são úteis em arquiteturas "Comportamentais, e em test benches, logo não serão construções sintetizáveis. opção A

Exercício 8

-> Ao observar o código podemos verificar que as entradas possuem 2 bits e as saídas 4, logo o bloco terá o formato 2:4, pelo que se trata de um descodificador binário 2:4. Opção D

Exercício 9

-> s_a <= "1010"
s_b <= "0110"
sig <= (s_a(3) & s_a) + (s_b(3) & s_b)
↳ "11010" + "00110" -> após a 1 é acrescentado bit, conta mantém-se o tipo, logo

sig é do tipo signed (4 downto 0) opção B

Exercício 10

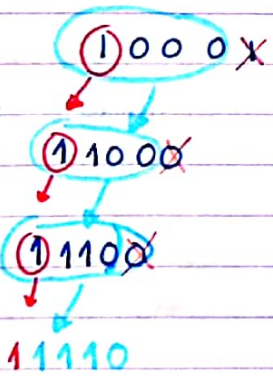
-> s_a = "1010" = ~~-8~~ -6
s_b = "0110" = 6 opção e
-6 x 6 = ~~-48~~ -36

Exercício 11

-> s_reg <= s_reg(N-1) & s_reg(N-1 downto 1);

-> Ao observar a linha de código podemos concluir que a cada deslocação o bit s_reg(0) é perdido, logo, trata-se de um deslocamento ~~lógico~~ aritmético à direita. exemplo:

opção c



1 0 0 0 1
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
direita

Exercício 12

```
process (clk)
  begin
    if (fallin_edge (clk)) then
        dOut <= dIn;
    end if;
end process;
```

traduzindo, sempre que o clock não está ativo a entrada é refletida na saída, quando o clock ativa, o sinal permanece igual ao anterior (não altera)

Reage aos impulsos negativos ← opção A

Exercício 13

```
process (clk)
  begin
    if (clk = '1') then
      dataOut <= dataIn;
    end if;
end process;
```

→ opção A

Exercício 14

→ Packages podem incluir: tipos, subtipos, constantes, funções, atributos, ...

→ Não incluem arquiteturas/implementações opção D

Exercício 15

→ Registo é um circuito formado por n flip-flops, que armazena n bits simultaneamente.

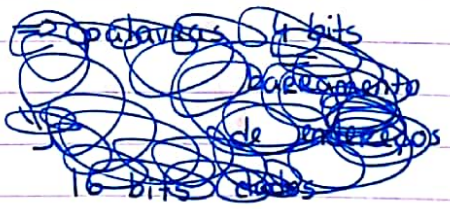
↳ 32 bits = 32 Flip-Flops opção B

Exercício 16

→ memória RAM 2 portos

↳ 4 bits barr. endereços
↳ "length do array"

↳ 16 bits barr dados
↳ "length dos dados armazenados no array"



~~4 x 16 = 64 bits armazenados~~

$2^{4bits} \times 16 = 256$ bits armazenados

PARTE II

Exercício 17

```
process (enable, counter)
 begin
   if (enable = '1') then counter <= counter + 1;
   end if;
end process;
```

Por cada vez que ativemos o enable o resultado será incrementado a 1 Contador edge triggered crescente.

Exercício 18

-> O sinal C muda de '0' para '1' aos 30ns (só responde aos impulsos positivos do clk).

Exercício 19

-> O módulo é um valor x que um contador pode contar.
O reset é ativo quando s_count(3) x s_count(0) = '1'
↳ valor máximo = 1001 = $2^3 + 2^0 = 9$ // = módulo

Exercício 20

-> o nº mais elevado contado é 9
ao contar de 0 a 9 temos 10 contagens.
em cada 10 contagens o bit count(3) é ativo 2 vezes (1000 e 1001), logo:

10 contagens —— 2 vezes
100 —— x

$$x = \frac{100 \times 2}{10} = 20\ \%\ \text{duty cycle}$$

Exercício 21

-> A frequência ser igual a 50MHz significa que o período é 20ns (período do clk). Sendo necessárias 10 contagens do clock para o bit count(3) ser ativo por 40ns, logo, de 200 em 200ns o bit count(3) é ativo.

Exercício 22

```
R1 <= std_logic_vector (unsigned(A) + unsigned(B));
R2 <= std_logic_vector ( signed(A) + signed (B));
```

→ ao somar os valores A e B, obtemos sempre os mesmos vetores, embora quando convertidos para signed ou unsigned passam ter valores diferentes R1 será sempre igual a R2

Exercício 23

signed { a = -1, b = 1 — a < b → gtSigned = '0'

Unsigned { a = 15, b = 1 — a > b → gtUnsigned = '1'

Exercício 24

→ Ao analisar o código, consideremos o timerOut como sendo uma LED, LED essa que acende quando o utilizador dá início ao start e apaga quando a contagem termina (atraso à desoperação)

Exercício 25

→ Trata-se de uma RAM com 2 portos de acesso (porta da escrita e porta da leitura)

Exercício 26

→ address = "length do array" → $2^{address}$ = palavras que cabem no array

→ dataSize = "length das palavras"

→ $2^{addrBusSize} \geq 64\,000$ palavras armazenadas

↳ addrBusSize = 16

addrBusSize = dataBusSize = 16

EXERCÍCIO 27

↳ ao multiplicarmos nºs de 2 bits, sabemo que só poderão aparecer operandos de 0 a 3, logo a ROM irá armazenar os valores das "tabuadas" de 0 a 3, até ao 3, isto é, a ROM conterá:

0×0 = 0	1×0 = 0	2×0 = 0	3×0 = 0
0×1 = 0	1×1 = 1	2×1 = 12	3×1 = 3
0×2 = 0	1×2 = 2	2×2 = 24	3×2 = 6
0×3 = 0	1×3 = 3	2×3 = 6	3×3 = 9

(opção C)

EXERCÍCIO 28

↳ opção A

PARTE III

EXERCÍCIO 29

↳ dataIn = "1100"

dataOut(0) = 0 xor 0 = '0'

dataOut(1) = 1 xor 0 = '1'

dataOut(2) = 1 xor 1 = '0'

dataOut(3) = 1

dataOut <= "1010" -> D

EXERCÍCIO 30

↳ o objetivo é obter o diretório/caminho percorrido pelo dataIn, ao observar a arquitetura constatamos que o dataIn é utilizado pela primeira vez no bloco "in_reg", onde lhe é atribuído o sinal s_op, que corresponde ao bloco "op1", que depende do operand (numBits é necessário), de seguida a informação é passada ao bloco "op2" para que "sin" seja calculada caso dataIn se equadre no bloco "out_reg" (opção d)

Exercício 31

↳ opção 3

~~Máquina de Moore, o estado atual depende do anterior~~

-> O módulo Decide com arquitetura Behavioral é baseado numa máquina de estados finitos do tipo de Moore pois a saída depende apenas do estado atual.