

Laboratórios de Sistemas Digitais

Documentação do material de apoio aos projetos finais

Tomás Oliveira e Silva

Índice

1	Notas prévias	3
2	Recomendações	4
3	Geração de números pseudo-aleatórios (pseudo_random_generator.vhd)	6
3.1	A entidade pseudo_random_generator	6
3.2	Princípio de funcionamento da arquitetura heavy	7
3.3	Princípio de funcionamento da arquitetura light	8
4	Comunicações série (protocolo RS232, rs232_controller.vhd)	13
5	LCD (lcd_controller.vhd)	15
6	Descodificação de sinais de infravermelhos (protocolo NEC, ir_decoder.vhd)	17
7	Teclados e ratos PS/2 (ps2_controller.vhd)	19
7.1	Teclados PS/2	20
7.2	Ratos PS/2	21
7.3	A entidade ps2_controller	21
7.4	Fontes de informação	23
8	VGA (vga_config.vhd e vga.vhd)	24
8.1	O ficheiro vga_config.vhd	26
8.2	O ficheiro vga.vhd	27
8.3	Diagrama de blocos da interligação das três entidades VGA	28
8.4	O conceito de pipeline	30
9	Áudio (audio.vhd)	32
10	Outras entidades fornecidas	36
10.1	A entidade seven_segment_decoder	36
10.2	A entidade debouncer	36
10.3	A entidade pulse_generator	37
10.4	A entidade clock_generator	38
10.5	As entidades font_8x8_bold e font_16x16_bold	38
10.6	A entidade blob_sound_rom	39
10.7	A entidade sin_function	40
10.8	A entidade sram_controller	40
10.9	A entidade screen_capture	41
11	Como ordenar alguns números inteiros de uma forma eficiente	42
11.1	Como ordenar dois números inteiros	42
11.2	Como ordenar três ou quatro números inteiros	43
11.3	Como ordenar cinco ou seis números inteiros	43

12 Exemplos e demonstrações fornecidos	44
12.1 Exemplo <code>debouncer_example</code>	44
12.2 Exemplo <code>clock_generator_example</code>	44
12.3 Exemplo <code>pseudo_random_generator_example</code>	45
12.4 Exemplo <code>rs232_controller_example</code>	45
12.5 Exemplo <code>lcd_controller_example</code>	45
12.6 Exemplo <code>ir_decoder_example</code>	46
12.7 Exemplo <code>ps2_controller_example</code>	46
12.8 Exemplo <code>audio_example</code>	46
12.9 Exemplo <code>audio_volume_example</code>	46
12.10 Exemplo <code>sram_controller_example</code>	47
12.11 Exemplo <code>vga_example</code>	47
12.12 Exemplo avançado <code>font_shapes</code>	48
12.13 Demonstração <code>logic_analyzer</code>	48
12.14 Demonstração <code>histogram</code>	50
12.15 Demonstração <code>text_buffer</code>	51
13 Outros ficheiros de apoio (apenas para GNU/Linux)	52
13.1 <i>Scripts</i> de apoio	52
13.2 <i>Makefiles</i>	52
13.3 Programas de apoio	53

1 Notas prévias

Todo o material de apoio aos projetos finais segue as seguintes linhas orientadoras:

- A documentação de todo o material é disponibilizada num único documento. Cabe a cada um ler as partes do documento que lhe interessam.
- O estilo de escrita do código VHDL 2008 fornecido reflete os gostos estéticos do autor deste documento e não segue à risca o que é usado nos slides das aulas teórico-práticas e nos guiões das aulas práticas.
- O código fornecido foi formatado de modo a ter no máximo 160 colunas.
- Todos os comentários estão escritos em Inglês.
- O nome dado às arquiteturas das entidades fornecidas tenta dar alguma informação sobre a sua implementação. Por exemplo, no caso do *debouncer* são fornecidas duas arquiteturas: *basic* e *fancy*. Obviamente, a arquitetura *fancy* é mais complexa (mas tem um tempo de resposta menor).
- É **sempre** usado um único sinal de relógio, chamado `clock`. Um sinal de *reset*, caso exista, tem o nome de `reset`.
- Nos exemplos fornecidos o sinal de relógio que é usado não é necessariamente o de 50 MHz. Caso não o seja, no *top level* do projeto é instanciada a entidade que gera o sinal de relógio com a frequência pretendida a partir do sinal `clock_50`.
- Sempre que é preciso configurar uma das entidades fornecidas através de um ou mais parâmetros genéricos tenha em atenção que as frequências são sempre especificadas em Hertz usando números reais (por exemplo, $50.0e6$ para 50 MHz), e que tempos são sempre especificados em segundos usando também números reais (por exemplo, $10.0e-6$ para 10 micro-segundos).
- Números reais são usados **apenas** para definir constantes (em tempo de análise e de elaboração do código VHDL).
- Os nomes dados a entidades, portas, sinais, constantes, tipos e valores de tipos enumerados estão sempre em `snake_case` (palavras com letras minúsculas, com um *underscore* entre palavras).
- Os nomes dados a constantes e parâmetros genéricos usam apenas letras maiúsculas, com um *underscore* entre palavras.
- Quando não é possível fazer tudo que se pretende num único ciclo de relógio divide-se o que se pretende fazer em tarefas mais pequenas e usa-se um *pipeline* para as organizar temporalmente (isto acontece sempre quando se usa VGA).
- Como explicado na secção 8.4, os nomes de sinais síncronos usados em andares de um *pipeline* terminam em `_0` para os que entram no primeiro andar do *pipeline*, em `_1` para os que entram no segundo, e assim por diante; aos sinais assíncronos gerados num andar do *pipeline* acrescenta-se um `x` no fim.

O material de apoio está distribuído pelos diretórios descritos a seguir.

vhdl_code Código VHDL 2008 das entidades fornecidas (inclui exemplos simples do seu uso).

demonstrations Código VHDL 2008 de algumas demonstrações (exemplos mais complicados do uso das entidades fornecidas).

doc Documentação.

c_code Código (na linguagem de programação C) de alguns programas auxiliares.

bin Alguns *scripts* úteis (GNU/Linux).

useful_files Alguns ficheiros úteis (`master.qsf`, `master.sdc`, ...).

sof Ficheiros de programação de todos os exemplos e demonstrações.

2 Recomendações

Ao escrever o seu código VHDL recomenda-se que:

- **seja estudado o código das entidades fornecidas que vai usar no seu projeto**
Razão: não se devem usar entidades sem se saber como funcionam. Além disso, os comentários existentes no código podem ajudar a esclarecer algum detalhe do funcionamento de uma entidade que esteja omissa na documentação.
- **sejam estudados os exemplos fornecidos que possam ser relevantes para o seu projeto**
Razão: é muito provável que lá encontre qualquer coisa que lhe possa dar jeito.
- **organize o seu código de uma maneira visualmente bem estruturada e comente as partes menos óbvias do seu código**
Razão: o código deve ser fácil de entender por terceiros (e pelo próprio alguns anos depois).
- **sejam eliminadas todas as *latches***
Razão: uma *latch* geralmente cria problemas temporais ou comportamentos inesperados.
Como evitar: num processo combinacional, garantir que em todos os casos possíveis o novo nível lógico de um sinal é especificado.

<pre>-- AVOID -- process(x) is if x = '1' then NS <= 3; end if; ... end process;</pre>	<pre>-- USE -- process(x,PS) is NS <= PS; -- default value if x = '1' then NS <= 3; -- override default end if; ... end process;</pre>
---	--

É possível forçar o Quartus a dar erro quando uma *latch* é sintetizada. Para isso deve-se

1. ativar o Design Assistant durante a compilação (Menu → Assignments → Settings → Design Assistant → Run Design Assistant during compilation),
2. transformar avisos críticos em erros (Menu → Tools → Options → Messages → Promote critical warning messages to error messages), e
3. acrescentar o ficheiro `master.sdc` ao projeto. (É preciso acrescentar este ficheiro ao projeto para evitar que o TimeQuest emita avisos críticos.)

- **se passem todos os sinais de entrada da FPGA por registos**
Razão: se isto não for feito e se um sinal de entrada (assíncrono) mudar de nível lógico muito perto de uma transição de relógio então o estado do sistema pode ficar inconsistente, já que as saídas das partes combinacionais rápidas do circuito vêem o novo valor lógico enquanto que as saídas das partes lentas vêem o valor antigo. Com o uso de registos problemas deste tipo (e fenómenos de meta-estabilidade) desaparecem. Por exemplo, considere utilizar um *debouncer* para todas as botões e interruptores ou, em alternativa, use código como o que se segue.

```
process(clock_50) is
  if rising_edge(clock) then
    my_key <= not key(0); -- convert an asynchronous negative logic signal into
                        -- a synchronous positive logic signal
    ...
    if my_key = '1' then -- use the synchronous signal
      ...
    end if;
  end if;
end process;
```

Pelos mesmo motivo, sinais de *reset* também devem ou ser síncronos ou ser gerados por um circuito síncrono.

- **se use apenas um sinal de relógio**

Razão: quando existem dois ou mais domínios de relógio e quando os relógios não estão sincronizados por PLLs, ou seja, quando as transições de um relógio mais lento não ocorrem praticamente nos mesmos instantes de transições de um relógio mais rápido, podem existir problemas temporais relacionados com tempos de *hold* e de *setup* quando um sinal de um domínio é usado no outro domínio. (O domínio de um relógio é o subconjunto do circuito que é ativado por esse sinal de relógio.) Para evitar este potencial problema use apenas um domínio de relógio e use pulsos de ativação (*enable* ou *strobe*) para despoletar ações mais lentas. Por exemplo:

<pre>-- AVOID -- process(my_key) is if rising_edge(my_key) then ... end if; end process;</pre>	<pre>-- USE -- process(clock) is if rising_edge(clock) then last_my_key <= my_key; -- delay of one clock cycle if my_key = '1' and last_my_key = '0' then ... end if; end if; end process;</pre>
--	---

Em particular, em vez de usar um divisor de frequência use um gerador de pulsos.

- **sejam detetadas e corrigidas todas as violações temporais**

Razão: Uma violação temporal ocorre quando a lógica combinacional não consegue fazer o seu trabalho no tempo que tem para o fazer (normalmente um ciclo de relógio). Logo, uma violação temporal é um erro grave que **tem** de ser corrigido.

Como detetar: incluir no seu projeto o ficheiro `master.sdc` (convenientemente adaptado: comentando os sinais do *top-level* que não são usados — e dando-lhe o nome do projeto).

Como corrigir: se não for possível reduzir o número de camadas da lógica combinacional então o trabalho terá de passar a ser feito em mais do que um ciclo de relógio, quer inserindo um registo no sítio apropriado (*pipeline*), quer usando o resultado da lógica combinacional após um tempo de espera apropriado (controlado através de um sinal de *enable*). No segundo caso é preciso alterar o ficheiro `master.sdc` de modo a descrever corretamente o que se está a passar.

- **sejam analisados todos os avisos emitidos pelo Quartus**

Razão: alguns dos avisos (mensagens a azul ou violeta) assinalam problemas que devem ser corrigidos. Quando fazer: deve-se dar uma vista de olhos pelas mensagens de aviso de vez em quando, e deve-se **necessariamente** fazer o mesmo antes de dar um projeto como concluído.

3 Geração de números pseudo-aleatórios (`pseudo_random_generator.vhd`)

Diz-se que uma sequência de números $\dots, x(-1), x(0), x(1), \dots$, é pseudo-aleatória quando esta é gerada através de uma fórmula matemática, e quando a sequência assim gerada apresenta propriedades estatísticas parecidas com uma sequência de números aleatórios (daí o termo pseudo-aleatório).

3.1 A entidade `pseudo_random_generator`

A entidade `pseudo_random_generator` é capaz de fornecer um novo número pseudo-aleatório com, no máximo, 24 *bits* em cada ciclo de relógio. O seu *interface* é o seguinte:

```
entity pseudo_random_generator is
  generic
  (
    N_BITS : integer range 1 to 24;
    SEED   : std_logic_vector(47 downto 0)
  );
  port
  (
    clock : in  std_logic;
    enable : in  std_logic := '1';
    rnd    : out std_logic_vector(N_BITS-1 downto 0)
  );
end pseudo_random_generator;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

N_BITS Número de *bits* do número pseudo-aleatório a ser gerado.

SEED Semente do gerador. A semente é um número de 48 *bits*, por exemplo `X"0123456789AB"`, que determina o estado inicial do gerador. Se for instanciado mais do que um gerador de números pseudo-aleatórios, é fortemente recomendado que sejam usadas ou sementes diferentes ou arquiteturas diferentes (existem duas).

clock Sinal de relógio.

enable Sinal de *enable*. O gerador só produz um novo número pseudo-aleatório se este sinal estiver a '1' (que é o valor por omissão se durante a instanciação da entidade não se disser nada acerca deste porto). Recomenda-se que se deixe o gerador sempre a trabalhar (*enable* sempre a '1'). Nesse caso, se os instantes nos quais o sinal *rnd* é usado pelo resto da lógica forem despoletados por ações externas à FPGA suficientemente espaçadas no tempo (por exemplo, ao se carregar num botão), então esse sinal será, para todos os efeitos práticos, verdadeiramente aleatório.

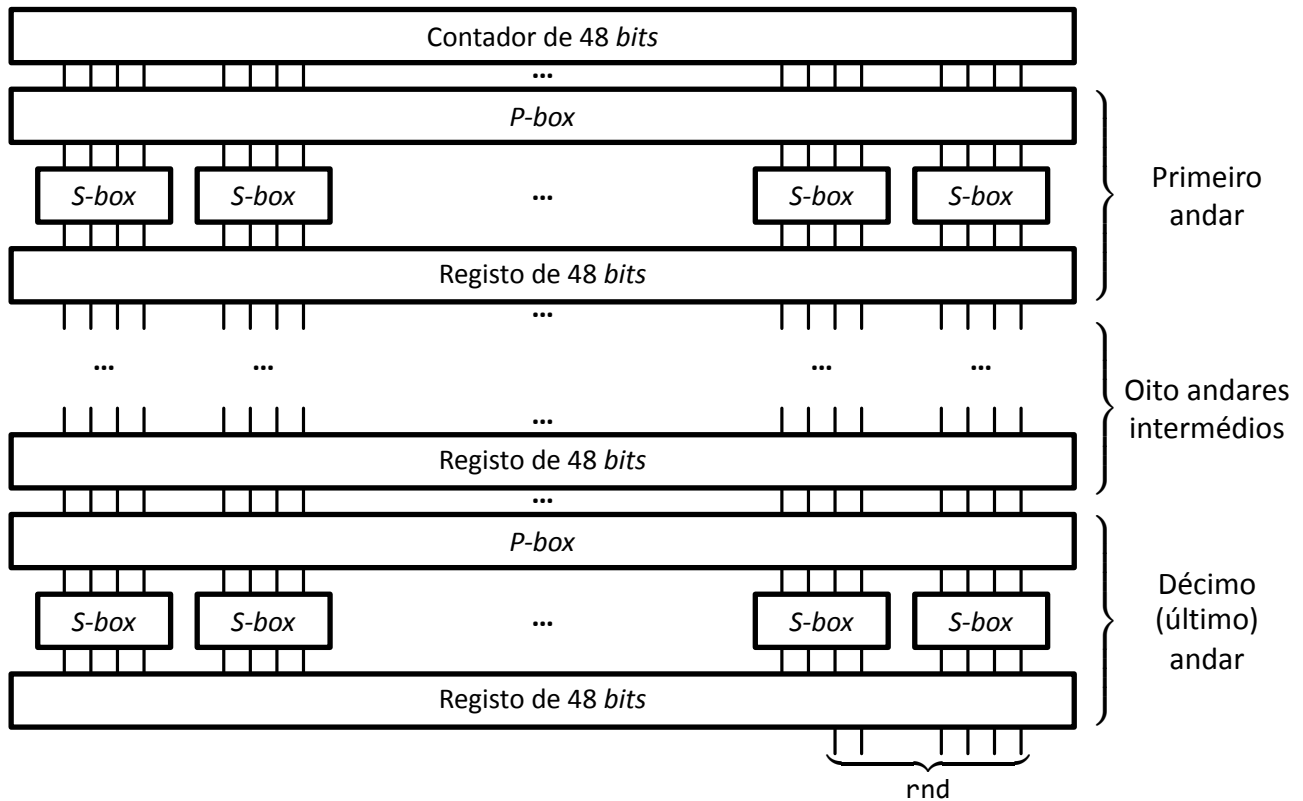
rnd Número pseudo-aleatório uniformemente distribuído gerado (calculado) pela entidade.

Para esta entidade existem duas arquiteturas, com nomes `heavy` e `light`, tendo o seu código VHDL sido gerado automaticamente por um programa escrito em C (`pseudo_random_generator.c`). A primeira usa uma quantidade de recursos da FPGA relativamente grande e foi desenhada tendo em conta algumas das boas práticas correntes relacionadas com segurança informática. A segunda é mais poupada em termos de recursos, mas produz números pseudo-aleatórios que são potencialmente de pior qualidade. Para reduzir ligeiramente os tempos de compilação, recomenda-se que seja utilizada a arquitetura `light` durante o desenvolvimento do projeto, e que se mude para a arquitetura `heavy` na sua versão final.

Esta entidade pode ser vista em ação no exemplo `pseudo_random_generator_example` e na demonstração `histogram`.

3.2 Princípio de funcionamento da arquitetura heavy

A figura seguinte mostra um diagrama de blocos do que está a acontecer dentro da arquitetura heavy (ilustrado para N_BITS igual a 6). Para não complicar a figura não se mostram as ligações dos sinais `clock` e `enable` ao contador e a todos os registos.



A ideia base por detrás do gerador consiste em baralhar os *bits* de um contador. O tipo de contador não é muito relevante (pode contar em binário, em código de Gray, ou outro); apenas é preciso que um contador de n *bits* tenha 2^n estados diferentes. A baralhagem dos *bits* do contador é feita em vários andares (quantos mais melhor) tal como ilustrado na figura, usando em cada um deles dois tipos de caixas:

caixas de permutação (em Inglês *permutation boxes*, abreviadas para *P-box*), que baralham a ordem dos *bits* que por elas passam (por exemplo, o *bit* 0 da saída é o *bit* 12 da entrada, o *bit* 1 da saída é o *bit* 35 da entrada, etc., sem nunca reutilizar um *bit* da entrada), e

caixas de substituição (em Inglês *substitution boxes*, abreviadas para *S-box*), que no nosso caso transformam 4 *bits* em 4 *bits* (os números de 0 a 15 na entrada são transformados em números de 0 a 15 na saída, sem repetições).

As caixas de permutação devem ser diferentes umas das outras, o mesmo acontecendo com as caixas de substituição. Cada uma destas últimas deve ser escolhida de modo a que cada *bit* de saída seja influenciado por todos os *bits* de entrada. Deste modo cria-se um fenómeno de avalanche, que faz com que ao fim de vários andares (no nosso caso bastam 5) cada *bit* do registo na saída do andar seja influenciado por cada um dos *bits* do contador.

Cada andar do nosso gerador usa uma única *P-box* de 48 *bits* (uma *P-box* usa apenas recursos de encaminhamento na FPGA), e usa 12 *S-boxes* (cada uma delas usa 4 blocos de lógica configurável da FPGA). Excluindo o contador, para 10 andares são utilizadas no total 480 blocos de lógica configurável, sendo também usado em cada uma delas o *flip-flop* lá presente.

Deve-se usar como *bits* de saída do gerador no máximo metade dos *bits* presentes no seu andar final.

3.3 Princípio de funcionamento da arquitetura light

Nota para o leitor(a): talvez seja mais prudente avançar já para a secção seguinte. O material apresentado a seguir tem um teor matemático elevado cuja compreensão não é necessária para se fazer uma correta utilização da entidade `pseudo_random_generator`. Um bom aluno(a) deve ser capaz de compreender o que aqui é exposto sem grande dificuldade.

Note que apenas se descreve aqui uma das maneiras possíveis de gerar uma sequência de números inteiros pseudo-aleatória, maneira essa que é particularmente fácil de sintetizar numa FPGA.

3.3.1 Aritmética modular

O nosso ponto de partida é a aritmética modular. Neste tipo de aritmética, depois de termos definido um módulo, que será um número inteiro positivo, apenas estamos interessados nos restos das divisões dos números inteiros por esse módulo. Por exemplo, quando o módulo é 12 e k é um número inteiro, os números $\dots, -10, 2, 14, 26, \dots, 2 + 12k, \dots$, são todos equivalentes, porque quando divididos por 12 dão todos resto 2. Para um módulo de m , teremos então m classes de equivalência, cada uma delas correspondendo a um resto r , com $0 \leq r < m$. É usual representar uma classe de equivalência apenas pelo seu resto r .

As operações aritméticas de soma, subtração e multiplicação são feitas da maneira habitual (tendo em atenção que estamos apenas interessados no resto). Apresentamos de seguida as tabelas das operações soma e multiplicação para os módulos 2 e 3.

$+$	0	1
0	0	1
1	1	0

\times	0	1
0	0	0
1	0	1

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

\times	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Note que a operação de adição módulo 2 corresponde à operação lógica ou-exclusivo (xor) e que a de multiplicação módulo 2 corresponde à operação lógica e (and), pelo que trabalhar módulo 2 é particularmente fácil.

Para um módulo genérico m , o conjunto das classes de equivalência e as operações $+$ e \times definem um anel. Em geral, nem todas as classes de equivalência possuem inverso; isso apenas acontece quando o máximo divisor comum entre r e m é 1. Como a operação de divisão é importante em algumas aplicações, é usual restringir o módulo a ser um número primo p , porque neste caso apenas o elemento neutro da adição, $r = 0$, não tem inverso. Neste caso o anel é também um corpo (é um caso particular de um corpo finito), usualmente designado por \mathbb{F}_p . Grosso modo, afirmar que algo pertence a \mathbb{F}_p diz-nos que esse algo é um número inteiro maior ou igual a 0 e menor do que p , e que as operações aritméticas efetuados com esse inteiro são feitas com aritmética modular (com módulo p).

Daqui para diante o módulo será sempre um número primo p . Na prática, será o número 2, mas a explicação da teoria pode, e deve, ser feita para o caso mais geral.

3.3.2 Fórmula matemática que vamos utilizar

A fórmula matemática que vamos usar para gerar números pseudo-aleatórios é extremamente simples: dado o estado atual do gerador, o estado seguinte é calculado através de uma simples multiplicação. Como estamos a trabalhar com aritmética modular, o número de estados possíveis do gerador é finito, pelo que a sequência que vai ser gerada entrará, mais tarde ou mais cedo, num ciclo. Para permitir sequências com ciclos com um período longo quando se usa um módulo p pequeno, o estado do nosso gerador no instante t

será um conjunto de n números pertencentes a \mathbb{F}_p , que vamos designar por $x_{n-1}(t), \dots, x_0(t)$ e que vamos agrupar no vetor coluna $u(t)$:

$$u(t) = \begin{bmatrix} x_{n-1}(t) \\ x_{n-2}(t) \\ \vdots \\ x_1(t) \\ x_0(t) \end{bmatrix}.$$

A nossa fórmula para atualizar o estado do gerador será

$$u(t+1) = Au(t),$$

onde A é uma matriz quadrada invertível (não singular) de dimensões $n \times n$. Como estamos a trabalhar com aritmética modular, cada entrada da matriz e o seu determinante também pertencem a \mathbb{F}_p (o determinante não pode ser 0 porque não queremos matrizes singulares). Como consequência desta fórmula é possível calcular rapidamente o estado do gerador no instante de tempo t_2 a partir do seu estado no instante t_1 :

$$u(t_2) = A^{t_2-t_1}u(t_1).$$

Como a matriz A é invertível, não é preciso que t_2 seja maior do que t_1 ; é perfeitamente possível andar para trás no tempo. (Por este, e por outros motivos que não vamos expor aqui, a utilização de um gerador de números pseudo-aleatórios deste tipo é altamente desaconselhada em aplicações que exigem elevado sigilo.)

Cada um dos estados possíveis (são p^n ao todo) faz parte de um ciclo; estados diferentes podem, é claro, fazer parte de ciclos diferentes. Mais uma vez, isso é uma consequência do facto de A ser uma matriz invertível. Em particular, o estado zero (tudo a zero) dá sempre origem a um ciclo de período 1 (ponto fixo). Este caso deve ser evitado a todo o custo, já que a sequência resultante, sempre zero, não é aleatória. Felizmente este caso pode ser evitado inicializando o gerador de números pseudo-aleatórios com um estado inicial diferente de tudo a zero.

Restam portanto $p^n - 1$ estados. Será que é possível escolher a matriz A de modo a que todos esses estados formem um único ciclo, de período $p^n - 1$? A resposta é sim. Mais ainda, existem muitas matrizes com essa propriedade pelo que encontrar uma é uma tarefa relativamente simples. Atendendo a que $u(t) = A^t u(0)$, para que o período seja $p^n - 1$ é necessário que A^{p^n-1} seja a matriz identidade. Para que o período não seja mais pequeno é necessário que $A^{(p^n-1)/f}$ não seja a matriz identidade, sendo f um factor de $p^n - 1$. Para testar esta última condição é suficiente restringir f aos números primos que são factores de $p^n - 1$.

Na arquitetura `light` da entidade `pseudo_random_generator` foi usada uma matriz de dimensões 48×48 , escolhida aleatoriamente de modo a satisfazer as condições descritas no parágrafo anterior e sujeitas às restrições de que i) o número de uns em cada linha (o *fan-in*) da matriz não é nem inferior a 2 nem superior a 4, e ii) o número de uns de cada coluna (o *fan-out*) não é nem inferior a 2 nem superior a 5. O período do gerador é $2^{48} - 1 \approx 2.815 \times 10^{14}$. Como são disponibilizados no máximo apenas 24 *bits* para o exterior, parte do estado do gerador não é diretamente exposto. O parâmetro genérico SEED define o estado inicial $u(0)$ do gerador. Note que foram tomadas precauções para garantir que o gerador não fica preso no estado tudo a zero.

3.3.3 Um caso particular muito usado na prática

Apesar de ser possível gerar uma sequência pseudo-aleatória de período máximo usando uma matriz A sem nenhuma estrutura especial, isto é, com a maioria dos seus n^2 elementos diferentes de zero, do ponto de vista prático é muitas vezes preferível usar uma matriz A em que a grande maioria dos seus elementos

é zero. Em particular, é possível escolher n elementos de \mathbb{F}_p , que vamos designar por a_0, \dots, a_{n-1} , de modo a que a matriz

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_2 & a_1 & a_0 \end{bmatrix}$$

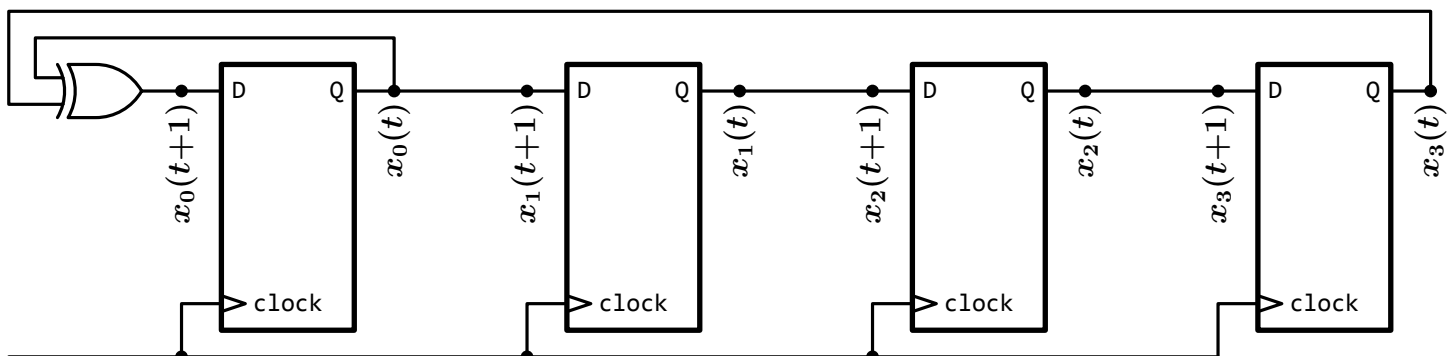
satisfaça as condições de período máximo. A uma matriz com esta forma dá-se o nome de matriz companheira. No nosso caso, como a matriz não pode ser singular, teremos necessariamente de ter $a_{n-1} \neq 0$. Com a matriz A nesta forma a atualização do estado do gerador é feita usando a fórmula

$$\begin{bmatrix} x_{n-1}(t+1) \\ x_{n-2}(t+1) \\ x_{n-3}(t+1) \\ \vdots \\ x_2(t+1) \\ x_1(t+1) \\ x_0(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} x_{n-1}(t) \\ x_{n-2}(t) \\ x_{n-3}(t) \\ \vdots \\ x_2(t) \\ x_1(t) \\ x_0(t) \end{bmatrix},$$

ou seja, temos

$$\begin{aligned} x_{n-1}(t+1) &= x_{n-2}(t) \\ x_{n-2}(t+1) &= x_{n-3}(t) \\ &\vdots \\ x_2(t+1) &= x_1(t) \\ x_1(t+1) &= x_0(t) \\ x_0(t+1) &= a_{n-1}x_{n-1}(t) + a_{n-2}x_{n-2}(t) + \dots + a_0x_0(t) = \sum_{k=0}^{n-1} a_k x_k(t). \end{aligned}$$

No caso $p = 2$ cada um dos $x_k(t)$ é um único *bit* pelo que as equações apresentadas acima descrevem o funcionamento de um registo de deslocamento (*shift register* em Inglês), no qual o *bit* que entra no registo de deslocamento é o ou-exclusivo de alguns dos *bits* internos do estado anterior do mesmo registo (trata-se de um *Linear Feedback Shift Register*). Em particular, na fórmula $x_0(t+1) = \sum_{k=0}^{n-1} a_k x_k(t)$ podemos eliminar as parcelas em que $a_k = 0$ e podemos eliminar as multiplicações por um nas em que $a_k = 1$. Tudo isto faz com que a geração de *bits* pseudo-aleatórios por este método seja muito eficiente e use muito poucos recursos (quer em lógica combinacional quer em registos), como se ilustra na figura seguinte para $n = 4$, $a_0 = 1$, $a_1 = 0$, $a_2 = 0$ e $a_3 = 1$ (pelo que $x_0(t+1) = x_0(t) + x_3(t)$; lembre-se que uma soma módulo 2 é um ou-exclusivo).



Neste caso concreto o gerador tem dois ciclos, um de período 1 no qual o estado é sempre 0000, caso que queremos evitar, e o outro de período 15 no qual o estado do gerador percorre sequencialmente todos os estádios do ciclo 0001, 0011, 0111, 1111, 1110, 1101, 1010, 0101, 1011, 0110, 1100, 1001, 0010, 0100, 1000. Neste último caso, e olhando para apenas um dos *bits* do gerador (não importa qual) verifica-se que num ciclo este toma o valor 0 sete vezes e o valor 1 oito vezes. Logo, os *bits* pseudo-aleatórios gerados não estão distribuídos uniformemente. Para valores elevados de n a diferença entre os dois casos ($2^{n-1} - 1$ contra 2^{n-1}) é irrisória.

Esta maneira de gerar *bits* pseudo-aleatórios, com a matriz A escolhida como descrito no slide anterior, tem, no que diz respeito à sua implementação, a desvantagem de que a lógica combinacional (os ou-exclusivos) está toda concentrada no cálculo de $x_0(t + 1)$, o que pode levar a que a frequência máxima de funcionamento do gerador não seja a mais alta possível (veja adiante como esse problema pode ser resolvido), o que pode acontecer se o número de coeficientes a_k diferentes de zero for maior do que dois. Felizmente isto não é um grande problema, porque para bastantes valores de n é possível ter apenas dois dos a_k iguais a um, e para todos os valores de n maiores do que 4 é possível ter apenas quatro dos a_k iguais a um.

Note que nos livros de engenharia sobre este assunto é usual apresentar este caso particular noutros moldes, usando o conceito de polinómio primitivo. Na opinião do autor deste documento é muito mais elegante e simples apresentar a teoria da maneira como foi aqui feita, que tem como pano de fundo o grupo linear geral das matrizes não singulares (*general linear group* em Inglês).

3.3.4 Outro caso particular muito usado na prática

Além da forma especial da matriz A descrita anteriormente, a forma seguinte, que é uma matriz companheira orientada de outra maneira e é igualmente útil, é por vezes utilizada (para distinguir as duas formas, neste segundo caso vamos passar a usar a letra B em vez da letra A):

$$B = \begin{bmatrix} -b_{n-1} & 1 & 0 & \cdots & 0 & 0 & 0 \\ -b_{n-2} & 0 & 1 & \cdots & 0 & 0 & 0 \\ -b_{n-3} & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ -b_2 & 0 & 0 & \cdots & 0 & 1 & 0 \\ -b_1 & 0 & 0 & \cdots & 0 & 0 & 1 \\ -b_0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}.$$

Neste caso temos de ter $b_0 \neq 0$, e as fórmulas de atualização do estado do gerador são

$$\begin{bmatrix} x_{n-1}(t+1) \\ x_{n-2}(t+1) \\ x_{n-3}(t+1) \\ \vdots \\ x_2(t+1) \\ x_1(t+1) \\ x_0(t+1) \end{bmatrix} = \begin{bmatrix} -b_{n-1} & 1 & 0 & \cdots & 0 & 0 & 0 \\ -b_{n-2} & 0 & 1 & \cdots & 0 & 0 & 0 \\ -b_{n-3} & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ -b_2 & 0 & 0 & \cdots & 0 & 1 & 0 \\ -b_1 & 0 & 0 & \cdots & 0 & 0 & 1 \\ -b_0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1}(t) \\ x_{n-2}(t) \\ x_{n-3}(t) \\ \vdots \\ x_2(t) \\ x_1(t) \\ x_0(t) \end{bmatrix},$$

ou seja,

$$x_k(t+1) = \begin{cases} x_{k-1}(t) - b_k x_{n-1}(t), & \text{para } k = 1, 2, \dots, n-1, \\ -b_0 x_{n-1}(t), & \text{para } k = 0. \end{cases}$$

Neste caso a lógica combinacional está espalhada ao longo do registo de deslocamento, em vez de estar toda concentrada no cálculo de $x_0(t + 1)$. É pois preferível usar esta forma quando se pretender atingir uma frequência de funcionamento do circuito o mais elevada possível.

É interessante constatar que se o estado do gerador for representado pelo polinómio (na variável X)

$$P(t; X) = \sum_{k=0}^{n-1} x_k(t) X^k,$$

definindo

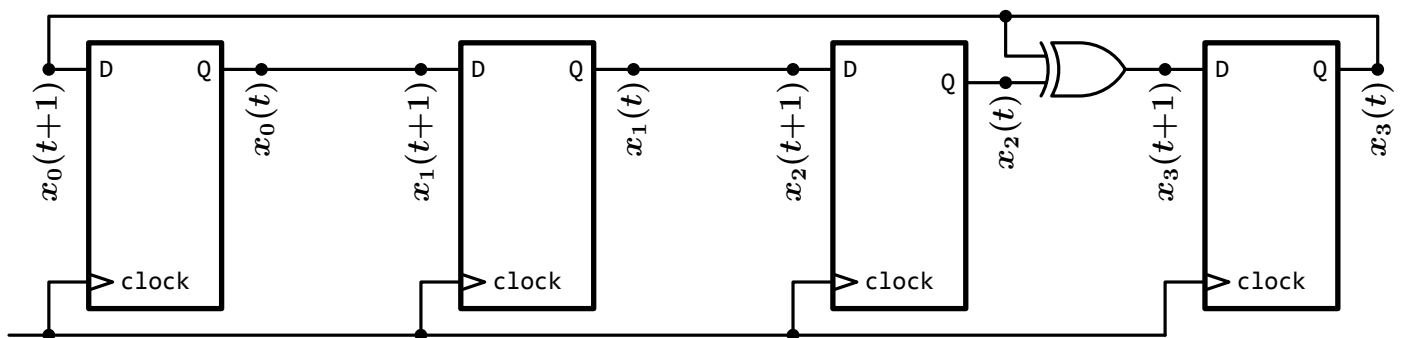
$$B(X) = X^n + \sum_{k=0}^{n-1} b_k X^k$$

(trata-se do tal polinómio primitivo a que se fez alusão atrás), e atendendo a que

$$XP(t, X) = \sum_{k=0}^{n-1} x_k(t) X^{k+1} = x_{n-1} X^n + \dots,$$

verifica-se que o resto da divisão de $XP(t, X)$ por $B(x)$ é dado por $XP(t; X) - x_{n-1}(t)B(X)$. Ora é isso mesmo que a fórmula apresentada no fim do parágrafo anterior faz, pelo que $P(t+1; X)$ é o resto da divisão de $XP(t; X)$ por $B(X)$, sendo obviamente as operações aritméticas feitas em \mathbb{F}_p . É pois possível fazer divisões de polinómios com registos de deslocamento (em base 2 isto é particularmente fácil).

Para $p = 2$, $n = 4$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$ e $b_3 = 1$, o diagrama de blocos do gerador toma a forma da figura seguinte (neste caso $-0 = 0$ e $-1 = 1$, pelo que podemos substituir todas as subtrações por somas).



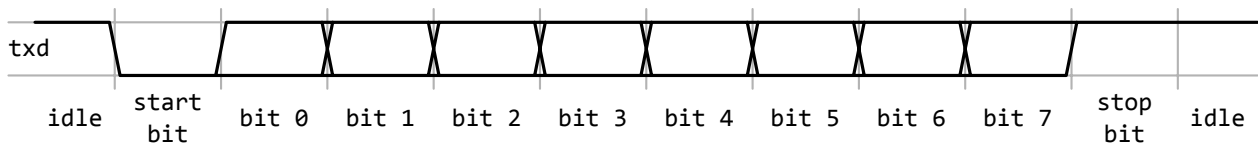
Note que substituindo $x_0(t+1) = x_3(t)$ por $x_0(t+1) = x_3(t) + y(t)$ o circuito da figura anterior pode ser usado para calcular um *cyclic redundancy checksum* (CRC) de 4 bits da sequência $y(t)$.

4 Comunicações série (protocolo RS232, rs232_controller.vhd)

O protocolo RS232 é um protocolo de comunicação ponto-a-ponto que requer apenas três fios condutores para se estabelecer uma comunicação bidirecional simultânea entre dois equipamentos: um fio para estabelecer uma tensão de referência comum aos dois lados da comunicação, chamado vulgarmente *ground*, um para transmitir informação (txd) e um para receber informação (rxd). É evidente que o fio que de um lado é usada para transmitir informação é o que do outro lado é usado para receber informação. Como os dois fios são independentes, pode-se estar a transmitir a a receber informação ao mesmo tempo (transmissão *full duplex*).

No protocolo RS232 não é transmitido o sinal de relógio. Os dois lados da comunicação têm então de combinar previamente a que ritmo vão transmitir informação, ritmo esse a que por razões históricas se dá o nome de *baud rate*. Cada símbolo transmitido, que no caso do RS232 é um *bit*, tem uma duração que é o inverso do *baud rate*. É também preciso combinar previamente quantos *bits* vão ser enviados de cada vez. Na entidade rs232_controller esse número de *bits* é configurado através de um parâmetro genérico. Finalmente, como não se transmite o sinal de relógio, é preciso utilizar *bits* extra para assinalar o início e o fim da transmissão de informação (um no início e pelo menos um no fim). Estamos aqui de propósito a omitir alguma informação; os interessados em usar este protocolo no seu projeto devem procurar outras fontes de informação para colmatar as lacunas na informação aqui disponibilizada.

A figura seguinte ilustra o que acontece ao longo do tempo quando se enviam 8 *bits* de informação usando o protocolo RS232. É mostrado na figura o nível lógico do sinal e não a tensão no fio condutor. Em repouso (*idle* em Inglês) o fio encontra-se no nível lógico '1'.



Note que na implementação do recetor é preciso ter em atenção que existe de certeza uma pequena diferença entre a frequência do relógio do transmissor e a frequência do relógio do recetor.

O *interface* da entidade rs232_controller é o seguinte:

```
entity rs232_controller is
  generic
  (
    CLOCK_FREQUENCY : real range 10.0e6 to 250.0e6;
    BAUD_RATE        : real range 300.0 to 1.0e6 := 115200.0;
    DATA_BITS       : integer range 5 to 10 := 8
  );
  port
  (
    clock : in std_logic;
    reset : in std_logic;

    uart_rxd : in std_logic;
    uart_txd : out std_logic := '1';

    rxd_data  : out std_logic_vector(data_bits-1 downto 0) := (others => '0');
    rxd_valid : out std_logic := '0';

    txd_data   : in std_logic_vector(data_bits-1 downto 0);
    txd_request : in std_logic;
    txd_accepted : out std_logic := '0'
  );
end rs232_controller;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

BAUD_RATE Parâmetro que especifica o número de símbolos (*bits* no caso do protocolo RS232) que podem ser enviados por segundo. No caso do protocolo RS232, alguns dos *bits* enviados não contêm informação útil; são *bits* de sincronização. A duração de um *bit* é o inverso deste parâmetro. Comunicações lentas usam geralmente um *baud rate* de 9600.

DATA_BITS Parâmetro que especifica o número de *bits* de dados a enviar, ou a receber, em cada trama. Para simplificar esta entidade, um eventual *bit* de paridade e um segundo *stop bit* terão de ser tratados pelo utilizador da entidade como se fossem *bits* de dados.

clock Sinal de relógio.

reset Sinal de *reset*.

uart_rxd e **uart_txd** Sinais a ligar aos pinos da FPGA com os mesmos nomes.

rxdata Última informação recebida com sucesso na linha **uart_rxd**.

rxdata_valid Pulso, com a duração de apenas um ciclo de relógio, que assinala que foi colocada nova informação no porto **rxdata**.

txdata informação a enviar para a linha **uart_txd**.

txdata_request quando ativo (a '1'), é pedido à entidade que envie a informação presente no porto **txdata** para a linha **uart_txd**. A entidade não é obrigada a aceitar o pedido de imediato, pelo que quem faz o pedido deve-o manter, bem como a informação no porto **txdata**, até que este seja aceite.

txdata_accepted quando a '1', indica que o pedido de envio de informação foi aceite. Este sinal só fica ativo durante um único ciclo de relógio.

Além desta entidade, também é fornecido no diretório `c_code` o código fonte do programa (para GNU/Linux) `lsd_term`, que pode ser usado, do lado do computador, para se efetuar uma comunicação com o *kit* usando o protocolo RS232. Como atualmente já praticamente não se fabricam computadores com portos RS232, para efetuar essa comunicação tem de se utilizar um cabo especial, que de um lado tem uma ficha RS232 e que do outro tem uma ficha USB (com um circuito integrado lá dentro que faz a conversão RS232 para USB). Caso precise de utilizar um cabo desses no seu projeto por favor contacte o seu professor das aulas práticas.

Esta entidade pode ser vista em ação no exemplo `rs232_controller_example`.

5 LCD (lcd_controller.vhd)

O *kit* DE2-115 tem um visor LCD (cristais líquidos) capaz de mostrar duas linhas de texto, cada uma delas com dezasseis caracteres. Depois da sua inicialização, que é feita automaticamente pela entidade fornecida quando se liga o *kit* ou quando se faz um *reset*, o visor fica à espera de comandos e de dados. Toda a informação a enviar para o visor é composta por 9 *bits*. Um dos *bits*, designado por RS, indica quando a '0' que os 8 *bits* restantes correspondem ao código de uma instrução, e indica quando a '1' que os 8 *bits* restantes correspondem a dados. Na página 14 (secção 12) do [data sheet do visor LCD](#) é apresentada uma tabela com os comandos que podem ser usados para o controlar. No nosso caso, a entidade `lcd_controller` fornecida coloca o sinal R/W sempre a '0', pelo que apenas podem ser usados comandos de escrita.

De propósito, não é dada aqui mais informação sobre o modo de funcionamento do visor, por forma a obrigar eventuais interessados em o usar a pesquisar e interpretar a informação do *data sheet*. Recomenda-se vivamente o estudo do exemplo `lcd_controller_example` fornecido. Tal como feito parcialmente no exemplo, é talvez mais fácil guardar o que se pretende mostrar numa memória e estar sempre a atualizar o visor com o conteúdo dessa memória.

A entidade `lcd_controller` fornecida encarrega-se de fazer a gestão baixo-nível da inicialização e da comunicação com o visor LCD. O *interface* desta entidade é o seguinte:

```
entity lcd_controller is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6
  );
  port
  (
    clock : in std_logic;
    reset : in std_logic := '0';

    lcd_on   : out std_logic;
    lcd_blon : out std_logic;
    lcd_rw   : out std_logic;
    lcd_en   : out std_logic;
    lcd_rs   : out std_logic;
    lcd_data : inout std_logic_vector(7 downto 0);

    txd_rs_and_data : in  std_logic_vector(8 downto 0);
    txd_request      : in  std_logic;
    txd_accepted     : out std_logic
  );
end lcd_controller;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

reset Sinal de *reset*.

lcd_on a **lcd_data** Sinais a ligar aos pinos da FPGA com os mesmos nomes.

txd_rs_and_data comando ou dados a enviar para visor; o *bit* mais significativo corresponde ao sinal RS.

txd_request quando ativo (a '1'), é pedido à entidade que envie a informação presente no porto txd_rs_and_data para o visor. A entidade não é obrigada a aceitar o pedido de imediato, pelo que quem faz o pedido deve-o manter, bem como a informação no porto txd_rs_and_data, até que este seja aceite.

txd_accepted quando a '1', indica que o pedido de envio de informação foi aceite. Este sinal só fica ativo durante um único ciclo de relógio.

Esta entidade pode ser vista em ação no exemplo `lcd_controller_example`.

6 Descodificação de sinais de infravermelhos (protocolo NEC, `ir_decoder.vhd`)

Quando se carrega num botão do comando remoto é enviada uma sequência de trens de pulsos de infravermelhos que codificam uma sequência de *bits*, que por sua vez especificam a marca, modelo e botão carregado do controlo remoto. A forma como cada *bit* é enviado e o número de *bits* enviados depende do formato de transmissão usado.

Existem vários formatos (protocolos) para a transmissão de informação. Atendendo a que foram desenvolvidos por empresas diferentes (NEC, Sony, Philips, *etc.*) esses formatos são, infelizmente, incompatíveis entre si.

No caso do *kit* DE2-115 o sinal que chega à FPGA vindo do sensor de infravermelhos, do tipo `std_logic` e com o nome `irda_rxd`, encontra-se em repouso no nível lógico '1' e passa a '0' quando é detetado um trem de pulsos de infravermelhos. Descodificar um sinal de infravermelhos consiste então em extrair a informação presente na duração dos intervalos de tempo em que `irda_rxd` permanece a '1' e a '0'. A maneira como essas durações são interpretadas depende do protocolo usado. A entidade fornecida, `ir_decoder`, apenas tem uma arquitetura, chamada `nec`, que descodifica o protocolo NEC.

Para comandos remotos que usam o formato de transmissão **NEC**, caso do comando fornecido com o *kit* DE2-115 e de comandos remotos da Samsung, a informação é enviada da seguinte maneira (já ajustada para a lógica negativa do sinal `irda_rxd`):

- 1 *bit* inicial de sincronismo (*start bit*)
9.0 mili-segundos no nível lógico '0' seguido de 4.5 mili-segundos no nível lógico '1'
- 32 *bits* de informação (*data bits*, *bit* menos significativo enviado primeiro)
para um *bit* a '0': 0.6 mili-segundos no nível lógico '0' seguido de 0.6 mili-segundos no nível lógico '1'
para um *bit* a '1': 0.6 mili-segundos no nível lógico '0' seguido de 1.6 mili-segundos no nível lógico '1'
- 1 *bit* final de sincronismo (*stop bit*)
0.6 mili-segundos no nível lógico '0'

A figura seguinte mostra o sinal recebido quando se carrega na tecla A do comando remoto do *kit* (é enviada a sequência de *bits* 0110 0001 1101 0110 1111 0000 0000 1111).



Nos comandos modernos os primeiros 16 *bits* (*bits* 0 a 15) identificam a marca e modelo do comando remoto. Os oito *bits* seguintes (*bits* 16 a 23) indicam o código do botão que foi carregado. Os oito *bits* finais (*bits* 24 a 31) destinam-se a detetar erros na transmissão e contêm uma versão negada do código do botão que foi carregado (isto é, têm de ser os *bits* 16 a 23 negados). Existe uma sequência especial, com um formato diferente, para indicar a repetição da última sequência de *bits* enviada (usada quando um botão do comando remoto é carregado durante muito tempo).

A entidade fornecida pode ser usada como ponto de partida para se fazer um descodificador para outros formatos de transmissão. Informação mais detalhada sobre protocolos de transmissão pode ser consultada na página web <http://www.sbprojects.com/knowledge/ir/index.php>; a informação sobre os vários protocolos pode ser acedida a partir de um menu *drop down* existe no canto superior direito dessa página. Na demonstração `logic_analyzer` mostra-se num ecrã VGA a forma de onda do sinal `irda_rxd` que foi recebido. Analisar essa forma de onda pode ser uma ajuda preciosa para descodificar outros protocolos de transmissão de informação.

O *interface* da entidade `ir_decoder` é o seguinte:

```
entity ir_decoder is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6
  );
  port
  (
    clock : in std_logic;
    reset : in std_logic;

    irda_rxd : in std_logic;

    data : out std_logic_vector(31 downto 0);
    valid : out std_logic := '0'
  );
end ir_decoder;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

clock_frequency Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

reset Sinal de *reset*.

irda_rxd Sinal a ligar ao pino da FPGA com o mesmo nome.

data Última informação recebida do comando de infravermelhos. Note que não é feita deteção de erros; em particular, `data(31 downto 24)` pode ser diferente de `not data(23 downto 16)` (se isto acontecer a informação foi recebida com pelo menos um erro).

valid_data Pulso, com a duração de apenas um ciclo de relógio, que assinala que foi colocada nova informação no porto `data`.

O sinal de repetição, enviado quando se carrega num botão por muito tempo, não é detetado.

O comando remoto fornecido com o *kit* DE2-115 envia os seguintes códigos (valores de `data`):

A X"F0_0F_6B_86"	B X"EC_13_6B_86"	C X"EF_10_6B_86"	power X"ED_12_6B_86"
1 X"FE_01_6B_86"	2 X"FD_02_6B_86"	3 X"FC_03_6B_86"	channel up X"E5_1A_6B_86"
4 X"FB_04_6B_86"	5 X"FA_05_6B_86"	6 X"F9_06_6B_86"	channel down X"E1_1E_6B_86"
7 X"F8_07_6B_86"	8 X"F7_08_6B_86"	9 X"F6_09_6B_86"	volume up X"E4_1B_6B_86"
menu X"EE_11_6B_86"	0 X"FF_00_6B_86"	return X"E8_17_6B_86"	volume down X"E0_1F_6B_86"
play X"E9_16_6B_86"	adjust left X"EB_14_6B_86"	adjust right X"E7_18_6B_86"	mute X"F3_0C_6B_86"

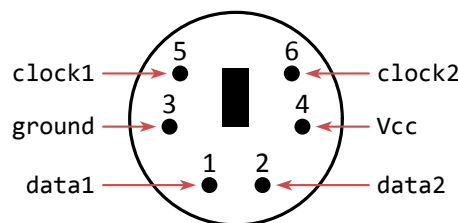
Esta entidade pode ser vista em ação no exemplo `ir_decoder_example`.

7 Teclados e ratos PS/2 (ps2_controller.vhd)

Ao nível físico as ligações entre um teclado ou rato PS/2 e um computador são feitas através de quatro fios:

- um de alimentação (Vcc, +5 V)
- um de massa (ground)
- um com um sinal de relógio (clock)
- um com um sinal de dados (data)

A ficha macho existente na extremidade do cabo que se liga ao computador tem 6 pinos, como se mostra na figura seguinte (mostra-se a frente da ficha, o cabo fica por detrás da figura). A ficha fêmea, que está no computador e no kit DE21-115, é uma imagem de espelho da que é aqui mostrada.

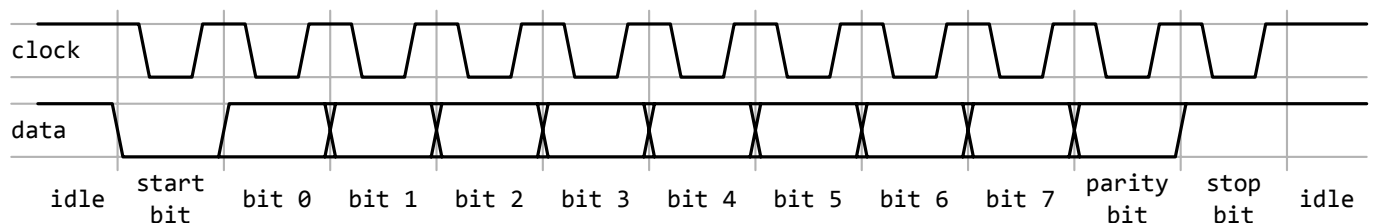


Normalmente apenas clock1 e data1 estão ligados (quer para teclados quer para ratos), mas a ficha foi concebida para se poder ligar dois equipamentos ao mesmo tempo. Nesse caso clock2 e data2 também estão ligados.

Nas linhas de relógio e de dados existe uma resistência de *pull-up*. É suposto apenas forçar uma tensão baixa nestas linhas (nível lógico '0'), o que é feito com um circuito do tipo *open-collector*. Em VHDL, para cada linha usa-se um sinal do tipo *inout* e apenas se coloca a linha nos estados '0' e 'Z'. Deste modo, cada um dos lados de uma ligação deste tipo pode prolongar a seu gosto o tempo em que a linha está com uma tensão baixa (podemos considerar que a linha faz parte de uma porta lógica *wired-and* distribuída). Quando não está a decorrer a transmissão de um *byte* ambas as linhas devem ter uma tensão alta (nível lógico '1').

No kit DE2-115 os sinais clock1, data1, clock2 e data2 estão respetivamente ligados aos pinos da FPGA a que foram atribuídos os nomes ps2_clk, ps2_dat, ps2_clk2 e ps2_dat2.

A figura seguinte mostra o diagrama temporal do que se passa quando o teclado ou o rato envia um *byte*. Temos um *start bit* a '0', oito *bits* de dados (*bit* menos significativo enviado primeiro), um *bit* de paridade ímpar (o ou-exclusivo dos *bits* 0 a 7 e do *bit* de paridade tem de dar '1'), e um *stop bit* a '1'. Ambos os sinais são controlados pelo teclado ou rato (o recetor é completamente passivo).

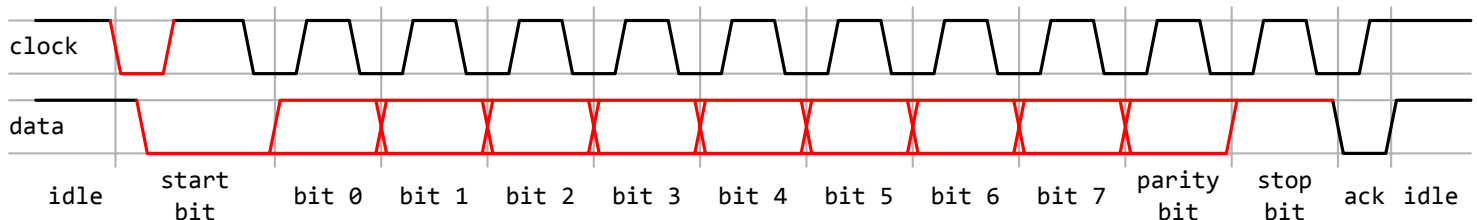


Os vários passos da receção de um *byte* vindo do dispositivo PS/2 são os seguintes:

1. O início da transmissão ocorre quando a linha de dados passa para o nível lógico '0'.
2. São gerados 11 ciclos de relógio. A linha de relógio permanece no mesmo nível lógico entre 30 a 50 micro-segundos.
3. A linha de dados tem de permanecer estável enquanto a linha de relógio estiver no nível lógico '0'.

4. As transições na linha de dados devem ocorrer a uma distância temporal de pelo menos 5 micro-segundos (para ambos os lados) das transições na linha de relógio.

A figura seguinte mostra o diagrama temporal do que se passa quando o teclado ou rato recebe um *byte*. A informação está organizada tal como na transmissão de um *byte*, mas agora é quem envia a informação que controla o pulso inicial a '0' do sinal de relógio e que controla a linha de dados na maior parte do tempo (a vermelho na figura).



Os vários passos do envio de um *byte* para o dispositivo PS/2 são os seguintes:

1. O início da transmissão ocorre quando a linha de relógio passa para o nível lógico '0'. Pouco depois (35 micro-segundos) a linha de dados também passa para o nível lógico '0'. Pouco depois (125 micro-segundos) a linha de relógio é libertada, pelo que volta ao nível lógico '1'.
2. Algum tempo depois (não mais de 1 mili-segundo), o teclado ou rato gera 11 ciclos de relógio.
3. A linha de relógio permanece no mesmo nível lógico entre 30 a 50 micro-segundos.
4. A linha de dados tem de permanecer estável enquanto a linha de relógio estiver no nível lógico '1'.
5. As transições na linha de dados devem ocorrer a uma distância temporal de pelo menos 5 micro-segundos (para ambos os lados) das transições na linha de relógio.
6. No fim o dispositivo PS/2 responde com um sinal de *acknowledge* (chamado *line control bit*).

7.1 Teclados PS/2

Sempre que se carrega numa tecla o teclado envia um ou mais *bytes*, a que se dá o nome (em Inglês) de *make code* da tecla. Uma boa parte das teclas tem um *make code* de apenas um *byte*, a quase totalidade das restantes tem um *make code* de dois *bytes* (o primeiro dos quais é sempre X"E0"), há uma com quatro *bytes* (a tecla *print screen*) e outra com oito *bytes* (a tecla *pause*).

Sempre que se deixa de carregar numa tecla o teclado envia dois ou mais *bytes*, a que se dá o nome (em Inglês) de *break code* da tecla (a tecla *pause* não tem *break code*). Regra geral, o *break code* de uma tecla é o *make code* da tecla com um *byte* extra com o valor X"F0", colocado logo no início quando o *make code* é de um *byte* e colocado a seguir a X"E0" (no meio) quando é de dois *bytes*.

Abstemo-nos de apresentar aqui uma lista de *make* e *break codes*. Procure por "*keyboard scan codes set 2*", ou, melhor ainda, por "*windows platform design notes keyboard scan code specification*", na internet. Como alternativa, compile e descarregue para o kit DE2-115 ou o exemplo `ps2_controller_example` ou a demonstração `text_buffer`, ligue um teclado ao kit e veja o que aparece dos visores de 7 segmentos ou no ecrã VGA quando se carrega numa tecla.

Dos vários comandos que um teclado PS/2 aceita destacamos os seguintes:

- X"FF" (reset): o teclado responde com X"FA", e depois faz um reinicialização. Durante a reinicialização os *leds* piscam uma vez, o teclado verifica se está tudo em ordem (*Basic Assurance Test*), e, caso afirmativo, o teclado fica ativo e é enviado X"AA"; caso haja algum problema é enviado X"FC".
- X"F5" (disable): o teclado responde com X"FA" e depois fica inativo.

- X"F4" (enable): o teclado responde com X"FA" e depois fica ativo.
- X"ED" (set/reset leds): o teclado responde com X"FA" e depois fica à espera de um *byte* com o estado que se pretende para os seus três *leds* (no *bit 0* segue o estado do *scroll lock*, no *bit 1* o do *num lock* e no *bit 2* o do *caps lock*). Depois de o receber responde com um novo X"FA".

7.2 Ratos PS/2

Um rato PS/2, quando ativo e quando se encontra a funcionar no chamado *stream mode*, envia uma mensagem sempre que i) há um movimento do rato, ou ii) se carrega ou deixa de carregar num dos seus botões. A informação enviada inclui o estado dos seus três botões e o deslocamento efetuado pelo rato, *delta_x* e *delta_y*, contado a partir da mensagem anterior. Tanto *delta_x* como *delta_y* são dois números inteiros de 9 *bits*, representados em complemento para dois.

Cada mensagem é composta por três *bytes*, a saber:

- Primeiro *byte* enviado:
 - bit 7*: indica se ocorreu um *overflow* no cálculo de *delta_y*
 - bit 6*: indica se ocorreu um *overflow* no cálculo de *delta_x*
 - bit 5*: *delta_y*(8)
 - bit 4*: *delta_x*(8)
 - bit 3*: sempre a '1'
 - bit 2*: estado do botão do meio
 - bit 1*: estado do botão da direita
 - bit 0*: estado do botão da esquerda
- Segundo *byte* enviado: *delta_x*(7 downto 0)
- Terceiro *byte* enviado: *delta_y*(7 downto 0)

Ratos com mais de três botões podem enviar informação usando outro formato.

Dos vários comandos que um rato PS/2 aceita destacamos os seguintes:

- X"FF" (reset): o rato responde com X"FA", e depois faz um reinicialização. Durante a reinicialização, o rato verifica se está tudo em ordem (*Basic Assurance Test*), e, caso afirmativo, é enviado X"AA" seguido por X"00"; caso haja algum problema é enviado apenas X"FC". Dependendo dos ratos, este pode ficar ou não ativo.
- X"F5" (disable data reporting): o rato responde com X"FA" e depois fica inativo.
- X"F4" (enable data reporting): o rato responde com X"FA" e depois fica ativo.
- X"F3" (set sample rate): o rato responde com X"FA" e depois fica à espera de um *byte* com o valor da nova taxa de amostragem (o rato passará a mandar no máximo esse número de mensagens por segundo, que deve ser 10, 20, 40, 60, 80, 100 ou 200). Depois de o receber responde com um novo X"FA". (Depois de um *reset* a taxa de amostragem é 100.)

7.3 A entidade ps2_controller

A entidade `ps2_controller` fornecida encarrega-se de fazer a gestão baixo-nível de um eventual dispositivo PS/2 que poderá estar ligado ao *kit* DE2-115. Em particular, quando o *kit* é ligado ou quando se faz um *reset* à entidade, esta tenta enviar, de 5 em 5 segundos, um comando de *reset* para o dispositivo PS/2. Caso este responda a entidade determina se se trata de um teclado ou de um rato e programa-o da maneira adequada, passando depois a processar a informação recebida do dispositivo. Após cada 4 segundos de

inatividade é enviado um sinal de ativação do dispositivo PS/2. Caso não haja resposta (o que acontece quando se retira o dispositivo), a entidade passa outra vez à fase de deteção de um dispositivo PS/2.

Erros ao nível dos sinais físicos provocam um *reset* da entidade (em princípio este tipo de erros só acontece quando o teclado/rato está avariado ou quando há muito ruído electromagnético).

O *interface* da entidade `ps2_controller` é o seguinte:

```
entity ps2_controller is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6
  );
  port
  (
    clock : in std_logic;
    reset : in std_logic := '0';

    ps2_clk : inout std_logic;
    ps2_dat : inout std_logic;

    keyboard_detected : out std_logic;
    keyboard_leds      : in  std_logic_vector(2 downto 0);
    key_code           : out std_logic_vector(7 downto 0);
    valid_key_code     : out std_logic;

    mouse_detected    : out std_logic;
    mouse_delta_x     : out std_logic_vector(8 downto 0);
    mouse_delta_y     : out std_logic_vector(8 downto 0);
    mouse_buttons     : out std_logic_vector(2 downto 0);
    valid_mouse_data  : out std_logic
  );
end ps2_controller;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

reset Sinal de *reset*.

ps2_clk Sinal a ligar ao pino da FPGA com o mesmo nome.

ps2_dat Sinal a ligar ao pino da FPGA com o mesmo nome.

keyboard_detected Fica a '1' quando um teclado é detetado.

keyboard_leds Estado pretendido para os *leds* do teclado (*bit 0: scroll lock, bit 1; num lock, bit 2: caps lock*). Quando a entidade deteta uma mudança no estado deste porto envia, logo que possível, um comando para o teclado para este mudar o estado dos seus *leds* de modo a refletir o estado do porto.

key_code Último *byte* recebido do teclado (parte de um *make code* ou de um *break code*).

valid_key_code Pulso, com a duração de apenas um ciclo de relógio, que assinala que foi colocada nova informação no porto `key_code`.

mouse_detected Fica a '1' quando um rato é detetado.

mouse_delta_x Último movimento, na horizontal, recebido do rato.

mouse_delta_y Último movimento, na vertical, recebido do rato.

mouse_buttons Último estado conhecido dos botões do rato (botão esquerdo no *bit* 2, botão do centro um *bit* 1, e botão da direita no *bit* 0).

valid_mouse_data Pulso, com a duração de apenas um ciclo de relógio, que assinala que foi colocada nova informação nos portos `mouse_delta_x`, `mouse_delta_y` e `mouse_buttons`.

Esta entidade pode ser vista em ação no exemplo `ps2_controller_example`.

Os *make* e *break codes* podem ser mais facilmente visualizados na demonstração `text_buffer`.

7.4 Fontes de informação

As seguintes fontes de informação (por vezes algo vagas e não 100% compatíveis), foram usadas para implementar a entidade `ps2_controller`:

- Personal System/2 Hardware Interface Technical Reference, Maio 1988
- [Freescale Semiconductor Application Note 1723](#)
- [USB and PS/2 Multimedia Keyboard Interface](#)
- [The PS/2 Mouse/Keyboard Protocol](#)
- [The PS/2 Keyboard Interface](#)
- [Interfacing the AT keyboard](#)

É frustrante verificar que aparentemente não existe um documento oficial que especifique preto no branco todos os tempos mínimos e máximos das fases inicial e final do envio de informação para um dispositivo PS/2.

8 VGA (vga_config.vhd e vga.vhd)

Um sinal de vídeo é composto por uma sequência de imagens (em Inglês usa-se o termo *video frame*), todas com a mesma dimensão. Cada imagem é composta por uma sequência de linhas horizontais, sendo cada linha composta por uma sequência de *pixels* (abreviatura do Inglês *picture elements*). Cada *pixel* tem uma cor, que no caso do sinal VGA é definida por um conjunto de três números que especificam a intensidade luminosa de três cores primárias: vermelho (r de *red*, em Inglês), verde (g de *green*, em Inglês) e azul (b de *blue*, em Inglês).

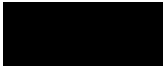
























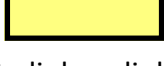
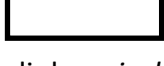
Devido às limitações do equipamento que gera (ou captura) uma imagem a intensidade luminosa de cada componente de cor tem um valor máximo, a que atribuímos o valor de 1.0 (100%). Se cada uma das três componentes for representada por um número de 8 *bits* sem sinal, como acontece no kit DE2-115 e na maioria das placas gráficas, a intensidade máxima, 1.0, corresponde ao número 255 e a intensidade mínima, 0.0, corresponde ao número 0.

Por curiosidade, na conversão de uma imagem a cores para uma a preto e branco, é usual usar a fórmula

$$\text{gray} = 0.30 \times \text{red} + 0.59 \times \text{green} + 0.11 \times \text{blue},$$

que reflete o facto de o nosso sistema visual ser bastante sensível aos tons verdes e muito pouco sensível aos tons azuis.

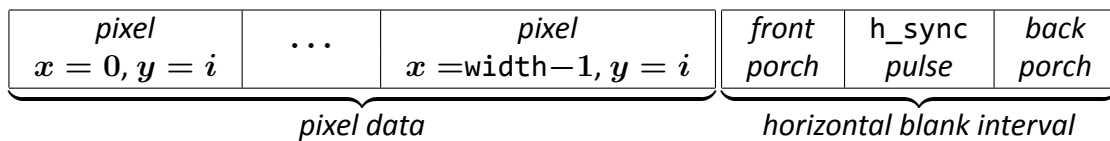
Existem várias maneiras de especificar uma cor. Os sinais VGA usam o espaço de cores RGB. Na tabela seguinte apresentamos algumas cores que podem ser obtidas combinando as três cores primárias (vermelho, verde e azul).

r,g,b	côr	r,g,b	côr	r,g,b	côr
0.0,0.0,0.0		0.0,0.0,0.5		0.0,0.0,1.0	
0.0,0.5,0.0		0.0,0.5,0.5		0.0,0.5,1.0	
0.0,1.0,0.0		0.0,1.0,0.5		0.0,1.0,1.0	
0.5,0.0,0.0		0.5,0.0,0.5		0.5,0.0,1.0	
0.5,0.5,0.0		0.5,0.5,0.5		0.5,0.5,1.0	
0.5,1.0,0.0		0.5,1.0,0.5		0.5,1.0,1.0	
1.0,0.0,0.0		1.0,0.0,0.5		1.0,0.0,1.0	
1.0,0.5,0.0		1.0,0.5,0.5		1.0,0.5,1.0	
1.0,1.0,0.0		1.0,1.0,0.5		1.0,1.0,1.0	

Cada imagem (*frame*) é transmitida sequencialmente linha a linha, e, dentro de cada linha, *pixel a pixel*, começando no canto superior esquerdo e acabando no canto inferior direito. Entre linhas da mesma imagem é introduzido um compasso de espera relativamente pequeno (chamado em Inglês *horizontal blank interval*) e entre imagens é introduzido um compasso de espera bastante maior, correspondente ao de várias linhas (chamado em Inglês *vertical blank interval*).

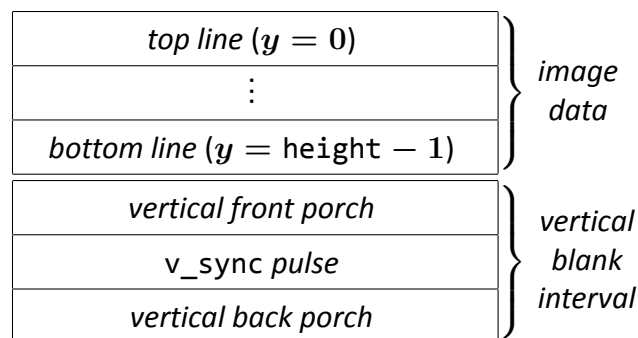
Além dos sinais r, g e b, que especificam a cor de cada *pixel*, num sinal VGA também é preciso definir um sinal de sincronismo horizontal, *h_sync*, activo numa parte do *horizontal blank interval*, e um sinal de sincronismo vertical, *v_sync*, activo numa parte do *vertical blank interval*.

Para um sinal de vídeo VGA de resolução $\text{width} \times \text{height}$ (width é o número de *pixels* na horizontal e height o na vertical) a informação a enviar na linha i da imagem está estruturada como se mostra a seguir.



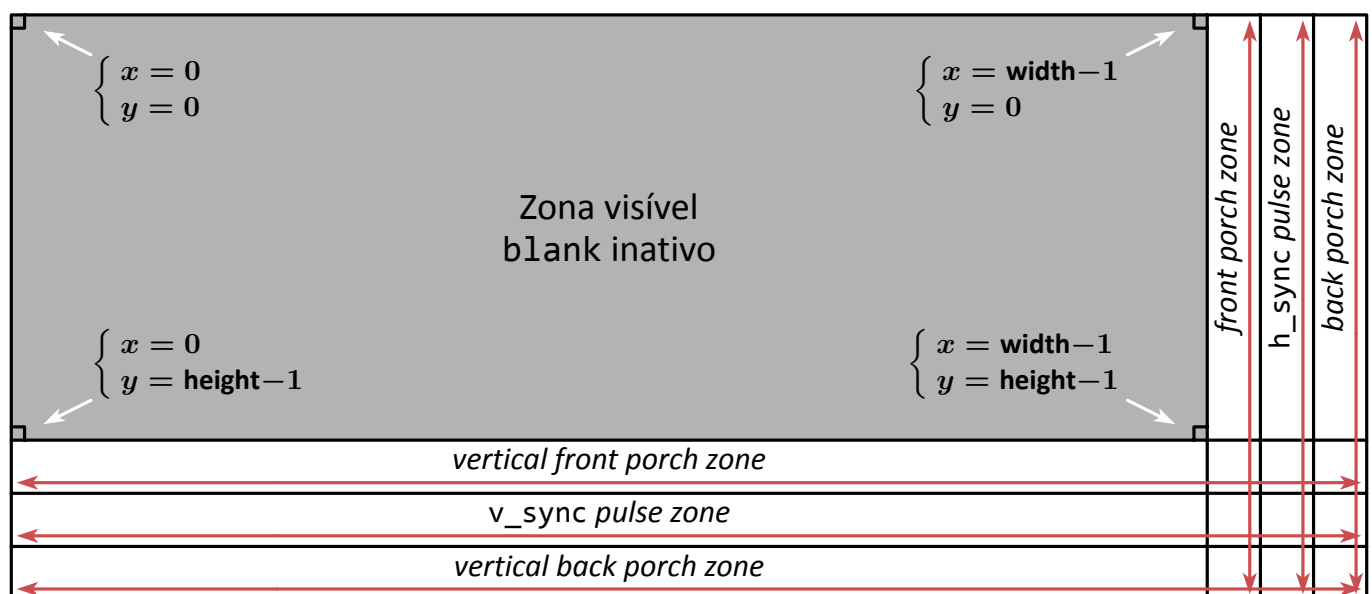
Esta informação é enviada sequencialmente, começando no lado esquerdo e acabando no lado direito. No mesmo *pixel* os sinais r, g e b são enviados em simultâneo em canais físicos separados.

A informação a enviar numa imagem completa está organizada como se ilustra a seguir (a informação também é enviada sequencialmente, agora começando em cima e acabando em baixo, sendo cada linha enviada como descrito no parágrafo anterior).



Para efeito de geração dos sinais físicos que são colocados no cabo que liga ao monitor, também é útil definir um sinal que não é transmitido, chamado *blank*, que indica, quando activo, que se está fora da zona visível da imagem. Esse sinal é habitualmente usado pelo circuito integrado que gera os sinais físicos para forçar os sinais r, g e b a zero fora da zona da imagem.

Toda a informação é enviada ao ritmo do *pixel clock*. Em particular, cada uma das três partes do *horizontal blank interval* tem uma duração que é um múltiplo da duração de um *pixel*. Algo de semelhante se passa com cada uma das três partes do *vertical blank interval*: cada uma delas tem uma duração que é um múltiplo da duração de uma linha. Todos os sinais podem ser gerados usando como referência a figura seguinte. O tempo avança da esquerda para a direita e de cima para baixo.



Um modo de vídeo não é especificado apenas pelo número de *pixels* na horizontal e na vertical da imagem. Também tem de se dizer quantas imagens são geradas por segundo (em Inglês, o termo usado é *refresh rate*). A partir destes três parâmetros existem regras para calcular a duração das zonas *front* e *back porch*

(quer horizontal quer vertical), e para calcular a duração e polaridade dos sinais de sincronismo horizontal e vertical. As que estão currentemente em uso, desenvolvidas pela *Video Electronics Standards Association* (VESA), são conhecidas pelos nomes *Generalized Timing Formula Standard* (GTF), de 1999, e *Coordinated Video Timings Standard* (CVT), de 2003. No sistema operativo GNU/Linux existem dois comandos, com os nomes `gtf` e `cvt`, que podem ser usados para calcular todos os parâmetros de um modo gráfico. Por exemplo, para o modo gráfico 1280x1024@60Hz (60 imagens por segundo, cada uma com a resolução 1280x1024) obtem-se

```
$ gtf 1280 1024 60
```

```
# 1280x1024 @ 60.00 Hz (GTF) hsync: 63.60 kHz; pclk: 108.88 MHz
```

```
Modeline "1280x1024_60.00" 108.88 1280 1360 1496 1712 1024 1025 1028 1060 -HSync +Vsync
```

pelo que

- o *pixel clock* é de 108.88 MHz (cada *pixel* tem a duração de aproximadamente 9.184 nano-segundos),
- cada linha tem um total de 1712 *pixels*, dos quais 1280 são visíveis, sendo que o sinal `h_sync`, de polaridade negativa, está ativo entre os *pixels* 1360 e 1496,
- existem no total 1060 linhas, das quais 1024 são visíveis, sendo que o sinal `v_sync`, de polaridade positiva, está ativo entre as linhas 1025 e 1028.

O exemplo `vga_example` e as demonstrações `logic_analyzer`, `text_buffer` e `histogram` mostram algumas coisas interessantes que se podem fazer com sinais VGA.

8.1 O ficheiro `vga_config.vhd`

Como várias entidades de um projeto podem precisar de conhecer todos os detalhes acerca do modo gráfico que vai ser usado nesse projeto, e como passar esses parâmetros todos através de genéricos é muito pouco prático, decidiu-se colocar todos esses parâmetros, na forma de constantes, num pacote (*package*) VHDL, a que se deu o nome `vga_config`, e cujo código está guardado no ficheiro `vga_config.vhd`. Para dar a conhecer a uma entidade o conteúdo desse pacote, basta incluir o ficheiro `vga_config.vhd` no nosso projeto e colocar a linha

```
use work.vga_config.all;
```

antes da declaração da entidade que precisa de conhecer essa informação (pode ser logo a seguir à linha `use ieee.std_logic_1164.all;`). Neste pacote estão guardados todos os detalhes acerca de alguns modos gráficos, sendo que o código seguinte lá presente

```
-----
---- The video mode that will be used ----
-----
constant VGA_MODE : vga_mode_t := VGA_MODE_800_600_72;
```

seleciona o modo gráfico que vai ser usado. Para mudar o modo gráfico basta pois editar o ficheiro `vga_config.vhd`, igualando `VGA_MODE` a um dos modos predefinidos (pode definir outros, se assim o desejar). Note que as constantes `VGA_FREQUENCY`, `VGA_WIDTH`, `VGA_HEIGHT` e `VGA_REFRESH_RATE` estão sempre de acordo com o modo gráfico selecionado. No pacote `vga_config` são também declarados alguns tipos de dados que facilitam muito trabalhar com sinais VGA:

```
--
-- VGA coordinates subtypes (we use a range larger than strictly necessary)
--
```

```

subtype vga_x_t is integer range -2*VGA_MODE.h_period to 2*VGA_MODE.h_period;
subtype vga_y_t is integer range -2*VGA_MODE.v_period to 2*VGA_MODE.v_period;
--
-- signals generated by the vga_controller entity
--
type vga_data_t is record
  --
  -- Signals required by the video DAC (Digital to Analog Converter)
  --
  h_sync      : std_logic; -- horizontal sync pulse
  v_sync      : std_logic; -- vertical sync pulse
  blank_n     : std_logic; -- blank signal ('1' when inside display area)
  --
  -- Signals useful to the generation of the color of each pixel
  --
  x           : vga_x_t;   -- x coordinate of the video signal
  y           : vga_y_t;   -- y coordinate of the video signal
  end_of_line  : std_logic; -- end of line pulse
  end_of_frame : std_logic; -- end of video frame pulse
end record vga_data_t;
--
-- Pixel color data type (used by the vga_output entity)
--
type vga_rgb_t is record
  r : std_logic_vector(7 downto 0); -- red color component
  g : std_logic_vector(7 downto 0); -- green color component
  b : std_logic_vector(7 downto 0); -- blue color component
end record vga_rgb_t;

```

O tipo de dados `vga_data_t` agrupa todos os sinais que são gerados pela entidade `vga_controller`, que é uma das três entidades que temos de instanciar num projeto que usa VGA. A partir das coordenadas x e y , no nosso projeto temos de gerar uma *côr* para o *pixel* com essas coordenadas, *côr* essa que pode ser convenientemente guardada num sinal do tipo `vga_rgb_t`. A entidade `vga_output`, que é a segunda das três entidades que temos de instanciar num projeto que usa VGA, recebe um `vga_data_t` e um `vga_rgb_t` e gera todos os sinais que é preciso enviar para o exterior da FPGA. A terceira entidade que é preciso instanciar, com o nome `vga_clock_generator`, é a que gera o sinal de relógio para os *pixels*. **Recomenda-se** que esse sinal de relógio seja o sinal de relógio de todo o sistema.

Para conveniência de quem pretende usar sinais VGA no seu projeto, as três entidades mencionadas no parágrafo anterior estão todas declaradas no ficheiro `vga.h`. (Também lá podíamos ter colocado o conteúdo do ficheiro `vga_config.vhd` mas decidimos não exagerar!)

8.2 O ficheiro `vga.vhd`

Das três entidades declaradas em `vga.h`, a mais simples é `vga_clock_generator`:

```

entity vga_clock_generator is
  port
  (
    clock_50 : in  std_logic; -- 50MHz clock
    vga_clock : out std_logic  -- VGA pixel clock
  );
end vga_clock_generator;

```

Note que não é preciso especificar a frequência pretendida porque esta é conhecida (está especificada no pacote `vga_config`). Se essa frequência for muito próxima de 50 MHz, o sinal `vga_clock` será o sinal

clock_50; se isso não acontecer, o sinal vga_clock será gerado através de uma instanciação da entidade clock_generator (que terá de ser adicionada ao projeto), descrita sumariamente na secção 10.4.

A entidade vga_controller é igualmente simples:

```
entity vga_controller is
  port
  (
    clock      : in  std_logic;      -- pixel clock (main clock)
    reset      : in  std_logic := '0'; -- if active, reset the VGA controller
    vga_data_0 : out vga_data_t      -- control signals
  );
end vga_controller;
```

Note que vga_data_0 contém vários sinais. Por exemplo, vga_data_0.x e vga_data_0.y são as coordenadas do *pixel* que está no momento a ser considerado. Note ainda que se terminou o nome deste porto com _0. Como veremos em breve, esta terminação tem como objetivo indicar qual o alinhamento temporal deste sinal em relação a outros sinais.

A declaração da entidade vga_output só é mais complicada que as anteriores porque é preciso especificar todos os sinais a ligar a pinos da FPGA:

```
entity vga_output is
  port
  (
    clock      : in  std_logic; -- pixel clock
    vga_data   : in  vga_data_t; -- the control signals
    vga_rgb    : in  vga_rgb_t;  -- the corresponding pixel color

    vga_clk    : out std_logic;   -- vga pixel clock
    vga_hs     : out std_logic;   -- vga horizontal sync
    vga_vs     : out std_logic;   -- vga vertical sync
    vga_sync_n : out std_logic;   -- vga sync signal (active low)
    vga_blank_n : out std_logic;  -- vga blank signal (active low)
    vga_r      : out std_logic_vector(7 downto 0); -- vga red component
    vga_g      : out std_logic_vector(7 downto 0); -- vga green component
    vga_b      : out std_logic_vector(7 downto 0); -- vga blue component
  );
end vga_output;
```

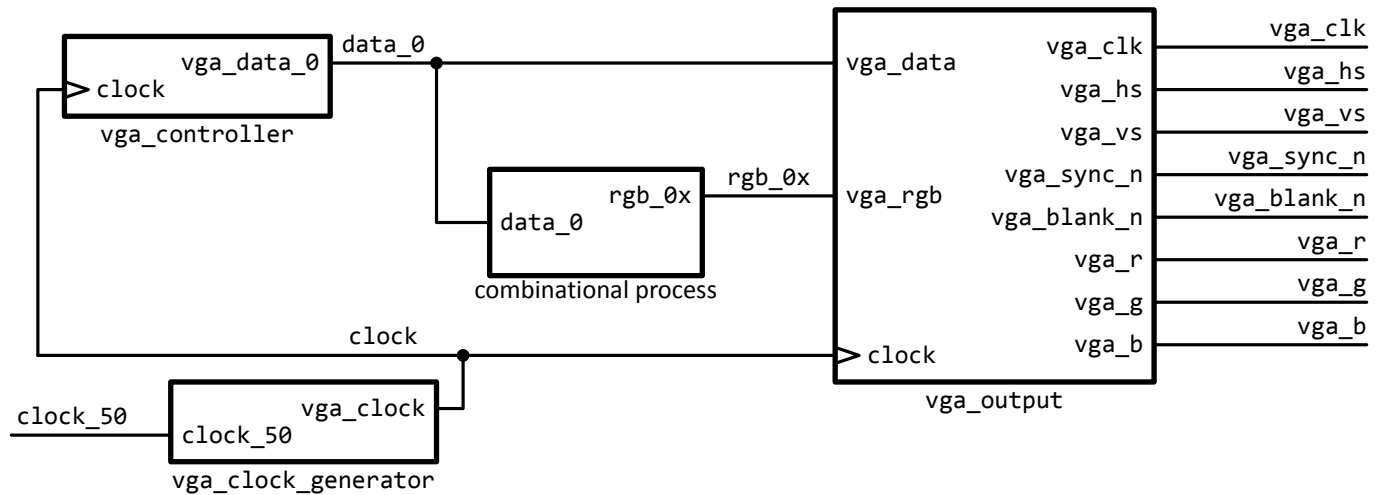
Em cada transição positiva do relógio clock, esta entidade recebe os sinais de sincronismo e a cor a utilizar, respetivamente em vga_data e vga_rgb, e coloca-os da maneira apropriada nos pinos da FPGA (portos vga_clk, ..., vga_b). Note que é importante que vga_data e vga_rgb estejam alinhados temporalmente, ou seja, que a cor que é fornecida em vga_rgb corresponda de facto às coordenadas do *pixel* que estão em vga_data.

8.3 Diagrama de blocos da interligação das três entidades VGA

Como exemplo, considere o caso muito simples em que se pretende criar uma imagem preta com um bordo branco de 4 *pixels* de largura. O processo combinacional seguinte dá conta do recado:

```
combinational : process(data_0) is
begin
  rgb_0x.r <= X"00"; rgb_0x.g <= X"00"; rgb_0x.b <= X"00"; -- black by default
  if data_0.x < 4 or data_0.x >= VGA_WIDTH-4 or data_0.y < 4 or data_0.y >= VGA_HEIGHT-4 then
    rgb_0x.r <= X"FF"; rgb_0x.g <= X"FF"; rgb_0x.b <= X"FF"; -- the border is white
  end if;
end process;
```

Como o processo é combinacional a cor está disponível **antes** da próxima transição positiva do sinal de relógio (assinalamos isso dando um nome que termina em `_0x` ao sinal que contém a cor). Podemos então fornecer à entidade `vga_output` os sinais `data_0` e `rgb_0x`, pois na próxima transição do relógio eles estarão alinhados temporalmente. O diagrama de blocos seguinte mostra como as várias entidades estão ligadas entre si.

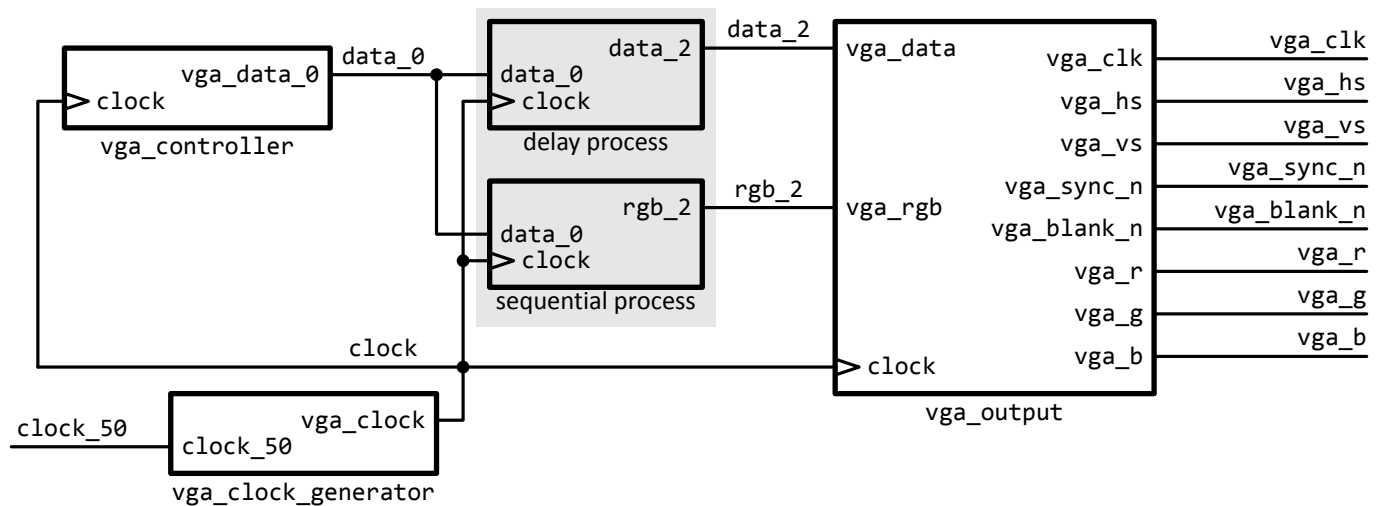


Se a lógica combinacional for demasiado lenta teremos de a subdividir em parcelas mais pequenas, como se ilustra a seguir (faz-se o `if` num ciclo de relógio e escolhe-se a cor no ciclo de relógio seguinte).

```

delay : process(clock) is
begin
    if rising_edge(clock) then
        data_1 <= data_0;
        data_2 <= data_1;
    end if;
end process;
sequential : process(clock) is
begin
    if rising_edge(clock) then
        border_1 <= '0'; -- not in border by default
        if data_0.x < 4 or data_0.x >= VGA_WIDTH-4 or data_0.y < 4 or data_0.y >= VGA_HEIGHT-4 then
            border_1 <= '1'; -- in border
        end if;
        rgb_2.r <= X"00"; rgb_2.g <= X"00"; rgb_2.b <= X"00"; -- black by default
        if border_1 = '1' then
            rgb_2.r <= X"FF"; rgb_2.g <= X"FF"; rgb_2.b <= X"FF"; -- the border is white
        end if;
    end if;
end process;

```



Note que as partes que "calculam coisas" foram colocados no mesmo processo sequencial. No primeiro ciclo de relógio determina-se o valor de `border_1` a partir das coordenadas presentes em `data_0`; em simultâneo, atrasa-se o sinal `data_0` obtendo-se `data_1`. No segundo ciclo de relógio, usa-se o valor de `border_1` para escolher a cor a guardar em `rgb_2`; ao mesmo tempo que se faz isto atrasa-se `data_1`, obtendo-se `data_2`. Deste modo `data_2` e `rgb_2` ficam alinhados temporalmente, pelo que a imagem irá ser gerada corretamente.

8.4 O conceito de *pipeline*

A maneira de fazer as coisas descrita acima é um exemplo de um *pipeline*. Uma outra maneira de escrever o código anterior, na qual os andares do *pipeline* estão claramente identificados e completamente separados é a seguinte (**recomenda-se** que escreva o seu código desta maneira):

```

stage_0_to_1 : process(clock) is
begin
  if rising_edge(clock) then
    data_1 <= data_0;
    if data_0.x < 4 or data_0.x >= VGA_WIDTH-4 or data_0.y < 4 or data_0.y >= VGA_HEIGHT-4 then
      border_1 <= '1'; -- in border
    else
      border_1 <= '0'; -- not in border
    end if;
  end if;
end process;

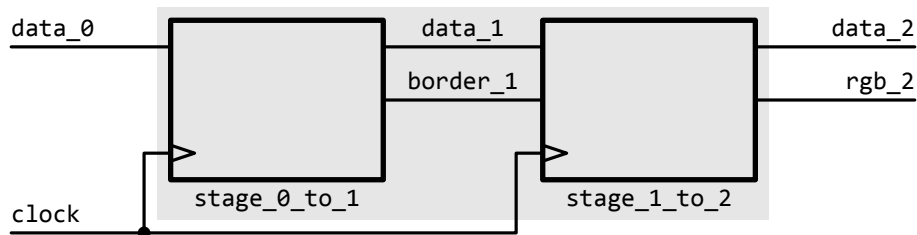
stage_1_to_2 : process(clock) is
begin
  if rising_edge(clock) then
    data_2 <= data_1;
    if border_1 = '1' then
      -- the border is white
      rgb_2.r <= X"FF";
      rgb_2.g <= X"FF";
      rgb_2.b <= X"FF";
    else
      -- the rest of the image is black
      rgb_2.r <= X"00";
      rgb_2.g <= X"00";
      rgb_2.b <= X"00";
    end if;
  end if;
end process;
  
```

```

end if;
end process;

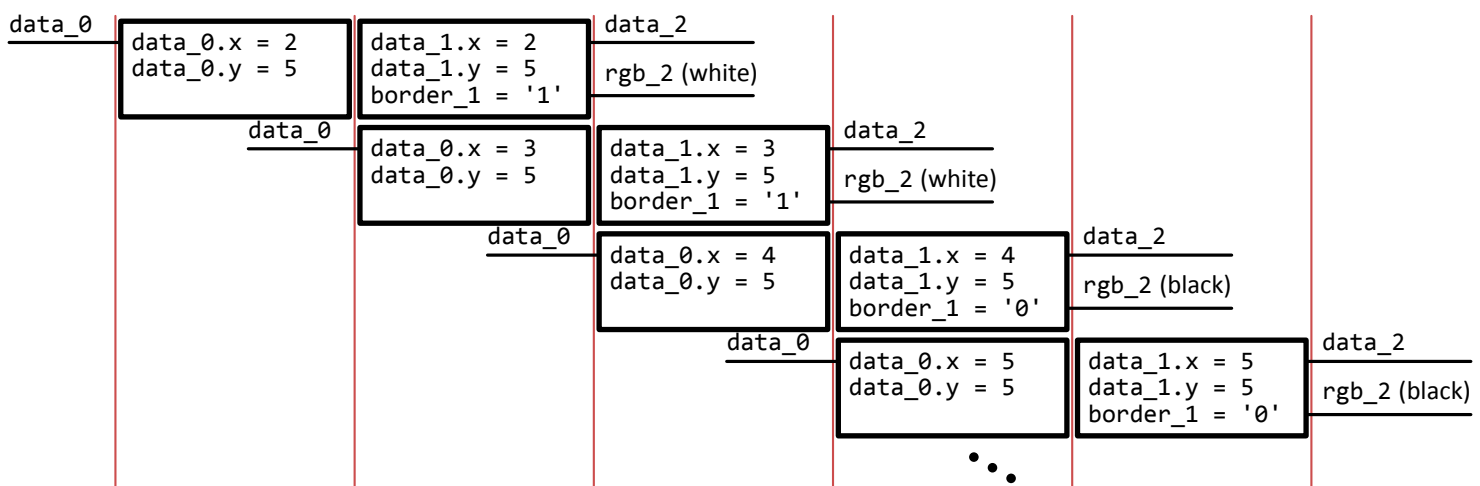
```

Com o código organizado desta maneira a zona com fundo cinzento claro da figura anterior passa a ter o seguinte aspeto.



Neste exemplo simples não foi preciso utilizar sinais assíncronos (processos combinacionais). Note que o último andar do *pipeline* podia ter sido feito de uma maneira puramente combinacional. No entanto, para evitar erros de distração e para facilitar uma eventual alteração do número de andares do *pipeline*, é preferível usar sempre sinais síncronos.

Na figura seguinte mostra-se um exemplo de como é que a informação flui no *pipeline* ao longo do tempo. Cada faixa vertical corresponde a um ciclo de relógio.



Note que neste caso cada *pixel* é processado em dois ciclos de relógio (latência de 2). Note ainda que em cada ciclo de relógio se começa a processar um novo *pixel*.

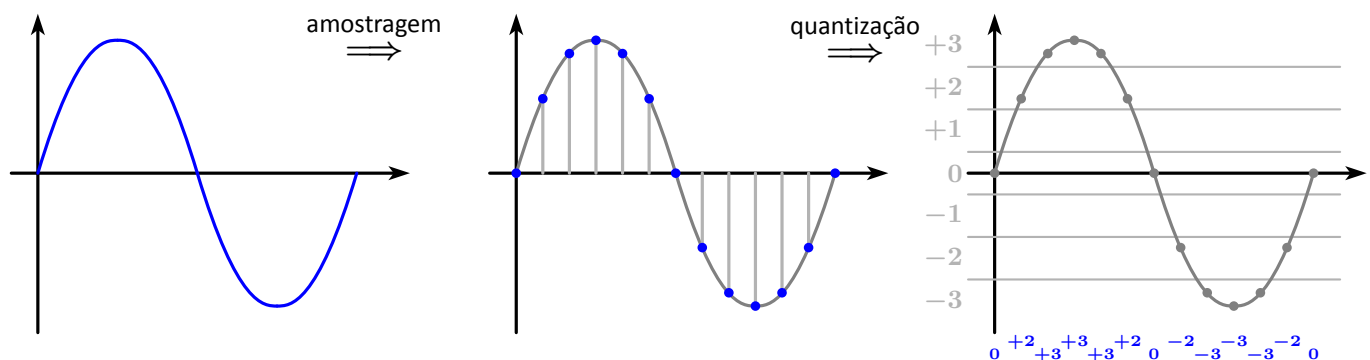
Usando um *pipeline* é relativamente simples gerar duas ou mais imagens independentes (nos primeiros andares do *pipeline*) e depois usar os andares finais para combinar as imagens numa única. Esta técnica é particularmente útil para quem pretende fazer jogos ou *screen savers*.

Para terminar, note que em geral cada andar de um *pipeline* pode ter uma parte sequencial (controlada pelo sinal de relógio) e uma parte combinacional. Recomenda-se que sejam dados nomes aos sinais que reflitam o andar do *pipeline* onde são gerados. Por exemplo, os sinais de entrada do andar número 2 devem acabar em `_2`, os sinais síncronos gerados por esse andar devem acabar em `_3`, e os sinais assíncronos gerados por esse andar devem acabar em `_2x` (para indicar que o nível lógico de cada sinal deste tipo pode mudar de valor antes da próxima transição positiva do sinal de relógio; os outros só podem mudar de nível lógico nas transições de relógio).

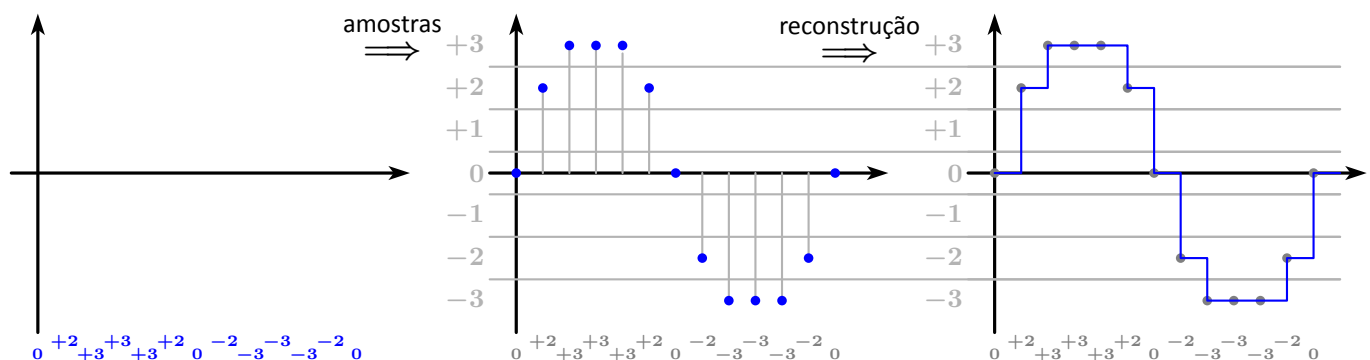
9 Áudio (audio.vhd)

Um codec (*COder-DECoder*) de áudio permite converter um sinal de áudio, composto por um ou mais canais, do domínio analógico para o digital, e, mais importante, permite fazer a conversão inversa, do domínio digital para o analógico. A primeira funcionalidade é utilizada, por exemplo, para gravar sons, e a segunda para os reproduzir.

A conversão do domínio analógico para o digital é feita por um conversor analógico-digital (*Analog-to-Digital Converter*, abreviado por ADC), e envolve tipicamente três passos distintos. No primeiro, filtra-se o sinal analógico de entrada, removendo as suas frequências mais altas (se isso não fosse feito, o resultado da conversão teria ruído indesejado). No segundo, o sinal é amostrado a uma determinada frequência de amostragem. Matematicamente, nesta operação passa-se do sinal $s(t)$ onde a variável tempo é contínua (número real), para o sinal $s(kT)$, onde k é um número inteiro e T é o período de amostragem. Sabe-se que a frequência máxima do sinal amostrado não pode ser superior a metade da frequência de amostragem. Finalmente, no terceiro passo o sinal amostrado é discretizado (quantizado), obtendo-se desta maneira uma aproximação de $s(kT)$ por um número com n bits. A figura seguinte ilustra estas duas últimas operações (usando uma quantização de 3 bits centrada em zero).



A conversão do domínio digital para o analógico é feita por um conversor digital-analógico (*Digital-to-Analog Converter*, abreviado por DAC), e também envolve tipicamente três passos distintos. No primeiro, a representação binária da amostra do sinal é convertida num sinal analógico. No segundo, o valor desse sinal analógico é mantido até ao instante da próxima amostra. No terceiro, é feita uma filtragem do sinal resultante, mais uma vez para lhe retirar as suas frequências mais altas. A figura seguinte ilustra as duas primeiras operações.



Um codec de áudio também pode ser configurado/programado de modo a se controlar alguns aspetos da cadeia de gravação e reprodução de som, tais como por exemplo, escolher a frequência de amostragem (na entidade fornecida esta foi fixada em 48 kHz), escolher o número de *bits* de cada amostra (foi fixado em 16), e escolher ganhos a aplicar aos sinais analógicos de entrada e de saída (na entidade fornecida podem ser ajustados). O diagrama funcional da figura 9 (página 21) do [data sheet](#) do codec de áudio mostra o que este é capaz de fazer. Recomenda-se o estudo deste diagrama pelos elementos dos grupos que querem utilizar áudio nos seus projetos finais.

O ficheiro `audio.vhd` declara duas entidades, `audio_controller` e `audio_io`. Ambas são necessárias para gravar ou reproduzir sinais de áudio. O *interface* da entidade `audio_controller` é o seguinte:

```
entity audio_controller is
  generic
  (
    CLOCK_FREQUENCY : real range 40.0e6 to 250.0e6
  );
  port
  (
    clock : in std_logic;
    reset : in std_logic;

    i2c_sclk : inout std_logic;
    i2c_sdat : inout std_logic;

    use_line_in : in std_logic;
    line_in_gain : in std_logic_vector(4 downto 0);

    use_mic : in std_logic;
    mic_boost : in std_logic;

    line_bypass : in std_logic;
    mic_bypass : in std_logic;
    volume : in std_logic_vector(6 downto 0)
  );
end audio_controller;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

reset Sinal de *reset*.

i2c_sclk e **i2c_sdat** Sinais a ligar aos pinos da FPGA com os mesmos nomes.

use_line_in Quando ativo (a '1'), a entrada do conversor analógico-digital é o sinal *line-in*.

line_in_gain Ganho a aplicar à entrada *line-in*; "00000" corresponde a -34.5 dB e "11111" corresponde a +12.0 dB (saltos de 1.5 dB entre valores consecutivos do ganho). Relembra-se que a escala em dB é uma escala logarítmica; +20 dB corresponde a um ganho (multiplicativo) de 10.

use_mic Quando ativo (a '1') e quando `use_line_in` está a '0', a entrada do conversor analógico-digital é o sinal *mic*.

mic_boost Ganho a aplicar à entrada *mic*; '0' corresponde a 0 dB e '1' corresponde a +20 dB.

line_bypass Quando a '1', parte do sinal *line-in* é misturado, com um ganho de -6 dB, com o sinal *line-out*.

mic_bypass Quando a '1', parte do sinal *mic* é misturado, com um ganho de -6 dB, com o sinal *line-out*.

volume Ganho a aplicar à saída *line-out*; abaixo de "0110000" produz silêncio (*mute*), "0110000" corresponde a um ganho de -73 dB e "1111111" corresponde a um ganho de +6 dB (saltos de 1 dB entre valores consecutivos do ganho).

Recomenda-se que inicialmente se coloque `line_in_gain` a "10000" (ganho do *line-in* de -10.5 dB) e se coloque `volume` a "1100000" (ganho do *line-out* de -25 dB).

O *interface* da entidade `audio_io` é o seguinte:

```
entity audio_io is
  generic
  (
    CLOCK_FREQUENCY : real range 40.0e6 to 250.0e6
  );
  port
  (
    clock      : in std_logic;
    clock_50   : in std_logic; -- the 50MHz clock (used to generate aud_xck)

    aud_xck     : out std_logic;
    aud_bclk    : in  std_logic;
    aud_adclrck : in  std_logic;
    aud_adcdat  : in  std_logic;
    aud_dacdat  : out std_logic;

    from_left   : out std_logic_vector(15 downto 0);
    from_right  : out std_logic_vector(15 downto 0);
    valid       : out std_logic;

    to_left     : in std_logic_vector(15 downto 0);
    to_right    : in std_logic_vector(15 downto 0)
  );
end audio_io;
```

Nota: esta entidade instancia a entidade `clock_generator`, que por isso também precisa de ser adicionada ao projeto. Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

clock_50 Sinal de relógio de 50 MHz, usado para gerar o sinal a enviar para o porto `aud_xck`.

aud_xck a **aud_dacdat** Sinais a ligar aos pinos da FPGA com os mesmos nomes.

from_left Última amostra recebida do canal esquerdo, interpretada como sendo um inteiro de 16 *bits* representado em complemento para 2.

from_right Última amostra recebida do canal direito, interpretada como sendo um inteiro de 16 *bits* representado em complemento para 2.

valid Pulso, com a duração de apenas um ciclo de relógio, que assinala que foi colocada nova informação nos portos `from_left` e `from_right`. São gerados cerca de 48000 pulsos por segundo. Atendendo à maneira como esta entidade foi implementada, este sinal não é ativado se `use_line_in` e `use_mic` estiverem ambos a '0'; por isso, em projetos onde se pretenda apenas reproduzir um som é necessário colocar um desses dois portos a '1', tal como feito no exemplo `audio_example`.

to_left Próxima amostra a ser enviada para o canal esquerdo, interpretada como sendo um inteiro de 16 *bits* representado em complemento para 2. Esta amostra deve ser colocada neste porto o mais tardar 10 micro-segundos após a última ativação do porto `valid` e o seu valor deve ser mantido até ao próximo pulso de `valid`.

to_right Próxima amostra a ser enviada para o canal direito, interpretada como sendo um inteiro de 16 *bits* representado em complemento para 2. Esta amostra deve ser colocada neste porto o mais tardar 10 micro-segundos após a última ativação do porto `valid` e o seu valor deve ser mantido até ao próximo pulso.

Por exemplo, para enviar o que se recebe do *line-in* para o *line-out*, põe-se o porto use_line_in da entidade audio_controller a '1' e faz-se (estamos aqui a usar sinais com nomes iguais aos dos portos da entidade audio_io):

```
process(clock) is
  if rising_edge(clock) then
    if valid = '1' then
      to_left  <= from_left;
      to_right <= from_right;
    end if;
  end if;
```

O instante de tempo no qual to_left e to_right vão de facto ser utilizados ocorre cerca de 10 microsegundos depois de valid ter estado a '1', mas como from_left e from_right não são alterados até ao próximo pulso de valid esta maneira de fazer as coisas é muito prática. Para reproduzir um som pode-se usar a mesma estratégia (nota: só são gerados pulsos no sinal valid se use_line_in@ estiver a '1' ou se use_mic estiver a '1'):

```
process(clock) is
  if rising_edge(clock) then
    if valid = '1' then
      to_left  <= left_rom(addr); -- or left_ram(addr) when a RAM is used
      to_right <= right_rom(addr); -- or right_ram(addr) when a RAM is used
      addr <= addr+1;
    end if;
  end if;
```

Estas entidades podem ser vista em ação nos exemplos `audio_example` e `audio_volume_example`.

10 Outras entidades fornecidas

Apresentamos aqui uma descrição sucinta de outras entidades fornecidas que são potencialmente úteis.

10.1 A entidade `seven_segment_decoder`

A entidade `seven_segment_decoder` (ficheiro `seven_segment_decoder.vhd`) é uma versão ligeiramente melhorada da que foi fornecida para ser usada em alguns guiões das aulas práticas. Além de permitir mostrar algarismos hexadecimais, permite também mostrar algumas outras letras e alguns símbolos. Para saber quais dê uma olhadela no código da entidade. O *interface* desta entidade é o seguinte:

```
entity seven_segment_decoder is
  port
  (
    code   : in  std_logic_vector(4 downto 0); -- code of the character
    enable : in  std_logic := '1';             -- if inactive, blank the display
    seg    : out std_logic_vector(6 downto 0) -- active low outputs
  );
end seven_segment_decoder;
```

Note que para mostrar apenas algarismos hexadecimais podemos escrever, no *port map* da instanciação da entidade, algo do género `code => '0' & digit`. Se isto for feito o quartus elimina tudo que é redundante (os casos 16 a 31, nos quais o *bit* mais significativo de `code` está a '1'), sendo sintetizada uma entidade muito parecida com a utilizada nos trabalhos prácticos.

10.2 A entidade `debouncer`

A entidade `debouncer` (ficheiro `debouncer.vhd` e exemplo `debouncer_example`) oferece mais funcionalidades que a fornecida para ser usada nos guiões das aulas práticas. Em particular, a partir de um sinal "sujo", isto é, com transições rápidas indesejáveis, é gerado um sinal "limpo", isto é, sem essas transições, e também são gerados pulsos, ativos durante apenas um ciclo de relógio, quando é detetada uma transição de '0' para '1' ou é detetada uma transição de '1' para '0' do sinal limpo. O seu *interface* é o seguinte.

```
entity debouncer is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6;
    WINDOW_DURATION  : real range 0.0 to 100.0e-3 := 20.0e-6;
    DELAY_DURATION   : real range 0.0 to 100.0e-3 := 20.0e-3;
    INITIAL_LEVEL    : std_logic                  := '0'
  );
  port
  (
    clock : in  std_logic;
    reset : in  std_logic;
    dirty  : in  std_logic;
    clean  : out std_logic;
    zero_to_one_pulse : out std_logic := '0';
    one_to_zero_pulse : out std_logic := '0'
  );
end debouncer;
```

Existem duas arquiteturas para esta entidade: `fancy` e `basic`. Diferem ligeiramente no seu grau de sofisticação. A arquitetura `fancy` responde mais rapidamente a uma transição do sinal "sujo".

Descreve-se a seguir a função de cada um dos parâmetros/portos do entidade debouncer.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

WINDOW_DURATION Parâmetro que especifica a duração, em segundos, da janela temporal a usar. O sinal é considerado limpo quando permanece estável durante este intervalo de tempo. Para a arquitetura fancy, deve ser usado um valor de $20.0e-6$. Para a arquitetura basic deve ser usado um valor de $20.0e-3$. Deve ser usado um valor de $20.0e-3$.

DELAY_DURATION Parâmetro apenas usado na arquitetura fancy. Depois da aceitação de uma transição do sinal sujo, este parâmetro especifica a duração de tempo, em segundos, durante a qual alterações do sinal sujo (*dirty*) são ignoradas.

INITIAL_LEVEL Nível lógico inicial do sinal limpo. Para evitar uma deteção de uma transição do sinal limpo quando a FPGA começa a trabalhar, deve ser utilizado o valor inicial '1' para os botões do *kit* key(0) a key(3), e deve ser utilizado o valor inicial '0' para os interruptores sw(0) a sw(17). Em geral, deve ser usado o nível lógico correspondente ao estado em repouso do sinal que se quer limpar.

clock Sinal de relógio.

reset Sinal de *reset*.

dirty Sinal que se quer limpar.

clean Sinal limpo. (Na instanciação da entidade pode não ser ligado: *clean* => *open*).

zero_to_one_pulse Pulso que é ativado, apenas durante um ciclo de relógio, sempre que ocorre uma transição de '0' para '1' no sinal limpo. (Na instanciação da entidade pode não ser ligado: *zero_to_one_pulse* => *open*).

one_to_zero_pulse Pulso que é ativado, apenas durante um ciclo de relógio, sempre que ocorre uma transição de '1' para '0' no sinal limpo. (Na instanciação da entidade pode não ser ligado: *one_to_zero_pulse* => *open*).

10.3 A entidade pulse_generator

A entidade *pulse_generator* (ficheiro *pulse_generator.vhd*) gera impulsos, com a duração de apenas um ciclo de relógio, a um ritmo fixo pré-estabelecido. O seu *interface* é o seguinte:

```
entity pulse_generator is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6; -- (in Hz) frequency of the clock signal
    PULSE_FREQUENCY : real range 0.01 to 250.0e6 -- (in Hz) frequency of the pulse signal
  );
  port
  (
    clock : in std_logic;          -- main clock
    reset : in std_logic := '0'; -- reset
    pulse : out std_logic          -- pulse signal (will be set to '1' at the pulse frequency)
  );
end pulse_generator;
```

Atendendo que o código desta entidade é muito simples a única coisa de é dita aqui acerca do seu funcionamento é que depois de um *reset* o primeiro pulso ocorre no fim do período (utilizações possíveis: temporizador, *auto-repeat*). Estude o código desta entidade e as suas utilizações nos exemplos fornecidos!

10.4 A entidade `clock_generator`

A entidade `clock_generator` (ficheiro `clock_generator.vhd` e exemplo `clock_generator_example`) gera, a partir do sinal `clock_50` de 50 MHz, um outro sinal de relógio com uma frequência próxima da por nós definida. Isto é feito instanciando a entidade `altpll` fornecida pela Altera/Intel, que nos permite usar uma das quatro PLLs (abreviatura de *Phase Locked Loop*), que temos disponíveis na nossa FPGA. Uma PLL permite, dentro de certos limites, gerar um sinal de relógio cuja frequência é um múltiplo racional da frequência do sinal que lhe é fornecida. Esta entidade é instanciada pelas entidades `vga_clock_generator` e `audio_io` e, a não ser que o aluno(a) se queira aventurar a trabalhar com um relógio que não seja de 50 MHz, não precisa que ser instanciada diretamente.

O interface da entidade `clock_generator` é o seguinte.

```
entity clock_generator is
  generic
  (
    FREQUENCY : real range 1.0e6 to 250.0e6
  );
  port
  (
    clock_50 : in  std_logic;
    new_clock : out std_logic
  );
end clock_generator;
```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

FREQUENCY Parâmetro que especifica a frequência desejado, em Hertz, do novo sinal de relógio.

clock_50 Como o nome do porto indica, relógio de 50 MHz.

new_clock Sinal de relógio gerado. A frequência deste relógio é num/den vezes 50 MHz, sendo num e den dois números inteiros pequenos cuja razão é uma aproximação da razão entre a frequência pretendida e 50 MHz. Para descobrir qual é a frequência exata do novo sinal de relógio, vá ao *Compilation report* e selecione, por esta ordem, *Fitter, Resource Section, PLL Summary/Usage*.

10.5 As entidades `font_8x8_bold` e `font_16x16_bold`

A entidade `font_8x8_bold` (ficheiro `font_8x8_bold` e exemplo `font_shapes`) usa internamente uma ROM para guardar 128 imagens monocromáticas, cada uma delas com 8 *pixels* de largura e de altura, e permite acessos de leitura a essa ROM de modo para se determinar se um determinado pixel de uma determinada imagem é '0' ou '1'. A maioria (95) dessas imagens correspondem aos caracteres ASCII e foram extraídas, com ligeiras alterações aqui e acolá, do ficheiro `font8x8_basic.h` do projeto <https://github.com/dhepper/font8x8>, gentilmente colocado no domínio público por Daniel Hepper e Marcel Sondaar. As restantes 33 imagens foram criadas pelo autor deste documento.

Esta entidade pode ser usada para mostrar texto numa imagem VGA. O exemplo `text_buffer` mostra como isso pode ser feito. O seu *interface* é o seguinte.

```
entity font_8x8_bold is
  port
  (
    clock : in std_logic;

    char_0 : in  std_logic_vector(6 downto 0);
    row_0  : in  std_logic_vector(2 downto 0);
```

```

    column_0 : in  std_logic_vector(2 downto 0);
    data_1   : out std_logic
  );
end font_8x8_bold;

```

Descreve-se a seguir a função de cada um dos seus portos.

clock Sinal de relógio.

char_0 Número da imagem (ou carácter) pretendida.

row_0 Número da linha pretendida (0 corresponde à linha de cima e 7 à de baixo).

column_0 Número da coluna pretendida (0 corresponde à coluna da esquerda e 7 à da direita).

data_1 Cór do *pixel* pretendido ('0' corresponde à cor de fundo), disponível no ciclo de relógio seguinte (nome do porto terminado em _1!).

A entidade `font_16x16_bold` (ficheiro `font_16x16_bold` e exemplo `font_shapes`) é semelhante à entidade `font_8x8_bold`, sendo que as imagens têm 16 *pixels* de largura e de altura e foram obtidas (e ligeiramente modificadas) do ficheiro `arial_bold.c`, disponível em http://www.rinkydinkelectronics.com/r_fonts.php, e gentilmente colocado no domínio público por MBWK. O seu *interface* é

```

entity font_16x16_bold is
  port
  (
    clock : in std_logic;

    char_0 : in  std_logic_vector(6 downto 0);
    row_0  : in  std_logic_vector(3 downto 0);
    column_0 : in  std_logic_vector(3 downto 0);
    data_1  : out std_logic
  );
end font_16x16_bold;

```

10.6 A entidade `blop_sound_rom`

A entidade `blop_sound_rom` (ficheiro `blop_sound_rom.vhd` e exemplo `audio_example`) é uma ROM, com dois portos de leitura síncrona, que armazena 4096 amostras de um som que soa como “blop”, som esse que foi gentilmente colocado no domínio público por Mark DiAngelo. O seu *interface* é o seguinte.

```

entity blop_sound_rom is
  port
  (
    clock : in std_logic;

    addr0_0 : in  std_logic_vector(11 downto 0);
    data0_1 : out std_logic_vector(15 downto 0);

    addr1_0 : in  std_logic_vector(11 downto 0);
    data1_1 : out std_logic_vector(15 downto 0)
  );
end blop_sound_rom;

```

Descreve-se a seguir a função de cada um dos seus portos.

clock Sinal de relógio.

addr0_0 Número da amostra pretendida para o primeiro porto de leitura.

data0_1 Valor lido do primeiro porto de leitura, disponível no ciclo de relógio seguinte.

addr1_0 Número da amostra pretendida para o segundo porto de leitura.

data1_1 Valor lido do segundo porto de leitura, disponível no ciclo de relógio seguinte.

10.7 A entidade `sin_function`

A entidade `sin_function` (ficheiro `sin_function.vhd` e exemplo `audio_example`), como o seu nome indica, calcula o seno do ângulo colocado no seu porto de entrada. Isso é feito usando um *pipeline* com 5 andares, pelo que o resultado só fica disponível 5 ciclos de relógio depois. Apesar de ter sido projetado para permitir a geração de sinais de rádio FM (isso, infelizmente, sai fora do âmbito desta unidade curricular), também pode ser usado para produzir tons puros (sons com apenas uma frequência) para áudio (tal como feito no exemplo `audio_example`).

O *interface* desta entidade é o seguinte.

```
entity sin_function is
  port
  (
    clock : in  std_logic;
    arg_0 : in  std_logic_vector(17 downto 0);
    sin_5 : out std_logic_vector(15 downto 0)
  );
end sin_function;
```

Descreve-se a seguir a função de cada um dos seus portos.

clock Sinal de relógio.

arg_0 Ângulo cujo seno é pretendido, codificado desde 0 (correspondendo a um ângulo de 0 radianos) até $2^{18} - 1$ (correspondendo a um ângulo de $(1 - 2^{-18})2\pi$ radianos); o valor n (sem sinal) corresponde ao ângulo $n/2^{18} 2\pi$ radianos.

sin_5 Seno do ângulo correspondente a `arg_0`, multiplicado por 32767.5 e truncado para um número inteiro representado em complemento para 2. Devido ao *pipeline* de 5 andares, o valor presente neste porto corresponde ao ângulo colocado no porto `arg_0` 5 ciclos de relógio antes.

10.8 A entidade `sram_controller`

O kit DE2-115 tem uma memória RAM rápida (conhecida em Inglês por *Static RAM*) ligada à FPGA, memória essa que tem 2^{20} palavras de 16 *bits*. A entidade `sram_controller` (ficheiro `sram_controller.vhd` e exemplo `sram_controller_example`) permite utilizar essa memória quer para leitura quer para escrita. O seu *interface* é o seguinte.

```
entity sram_controller is
  generic
  (
    CLOCK_FREQUENCY : real range 1.0e6 to 250.0e6
  );
  port
  (
    clock : in std_logic;

    sram_addr : out  std_logic_vector(19 downto 0);
    sram_dq   : inout std_logic_vector(15 downto 0);
```

```

sram_ce_n : out    std_logic;
sram_oe_n : out    std_logic;
sram_we_n : out    std_logic;
sram_ub_n : out    std_logic;
sram_lb_n : out    std_logic;

write_addr    : in  std_logic_vector(19 downto 0);
write_data    : in  std_logic_vector(15 downto 0);
write_request : in  std_logic;
write_accepted : out std_logic;

read_addr     : in  std_logic_vector(19 downto 0);
read_data     : out std_logic_vector(15 downto 0);
read_request  : in  std_logic;
read_valid    : out std_logic
);
end sram_controller;

```

Descreve-se a seguir a função de cada um dos seus parâmetros/portos.

CLOCK_FREQUENCY Parâmetro que especifica a frequência, em Hertz, do sinal de relógio. (Usado para calcular o número de ciclos de relógio correspondentes a determinados intervalos de tempo.)

clock Sinal de relógio.

sram_addr a **sram_lb_n** Sinais a ligar aos pinos da FPGA com os mesmos nomes.

write_addr Endereço usado para escritas na memória.

write_data Informação que se pretende escrever.

write_request quando ativo (a '1'), é pedido à entidade que escreva a informação presente no porto **write_data** na posição de memória dada pelo porto **write_addr**. A entidade não é obrigada a aceitar o pedido de imediato, pelo que quem faz o pedido deve-o manter, bem como a informação nos portos **write_addr** e **write_data**, até que este seja aceite.

write_accepted quando a '1', indica que o pedido de escrita de informação foi aceite. Este sinal só fica ativo durante um único ciclo de relógio.

read_addr Endereço usado para leituras da memória.

read_data Última informação lida.

read_request quando ativo (a '1'), é pedido à entidade que leia a informação guardada na posição de memória dada pelo porto **read_addr** e que a coloque no porto **read_data**. A entidade não é obrigada a aceitar o pedido de imediato, pelo que quem faz o pedido deve-o manter, bem como a informação no porto **read_addr**, até que este pedido seja dado como feito.

read_valid quando a '1', indica que o pedido de leitura de informação foi concluído. Este sinal só fica ativo durante um único ciclo de relógio.

10.9 A entidade screen_capture

A entidade **screen_capture** (ficheiro **screen_capture.vhd**) permite capturar (transferir para um PC) uma imagem VGA. Foi utilizada para obter alguma das imagens apresentadas na parte final deste documento.

Apenas para conhecedores!

11 Como ordenar alguns números inteiros de uma forma eficiente

Em alguns dos projetos finais de Laboratórios de Sistemas Digitais poderá ser necessário ordenar por ordem crescente vários números inteiros. O método combinacional que descrevemos aqui usa redes de ordenação, conhecidas em Inglês pelo nome de *sorting networks*, e é muito eficaz quando o número de inteiros a ordenar é pequeno. Para mais informações sobre redes de ordenação consulte a secção 5.3.4 (*networks for sorting*, páginas 219 a 247) do livro

Donald E. Knuth,
Sorting and Searching,
The Art of Computer Programming, volume 3.
 Addison-Wesley, Reading, Massachusetts, terceira edição, 1998.

Como alternativa, poderá pesquisar este assunto na *internet* (procure "*minimum-time sorting networks*" e "*zero-one principle*").

11.1 Como ordenar dois números inteiros

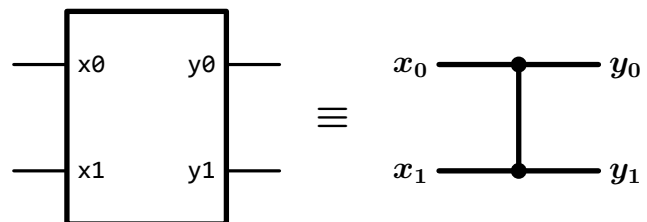
O caso mais simples ocorre quando se pretende ordenar apenas dois números inteiros. A entidade combinacional apresentada a seguir resolve facilmente este problema.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sort_two is
  generic
  (
    N_BITS : natural range 1 to 32
  );
  port
  (
    x0 : in  unsigned(N_BITS-1 downto 0); -- the first number to sort
    x1 : in  unsigned(N_BITS-1 downto 0); -- the second number to sort
    y0 : out unsigned(N_BITS-1 downto 0); -- the smallest of the two, min(x0,x1)
    y1 : out unsigned(N_BITS-1 downto 0)  -- the largest of the two, max(x0,x1)
  );
end sort_two;

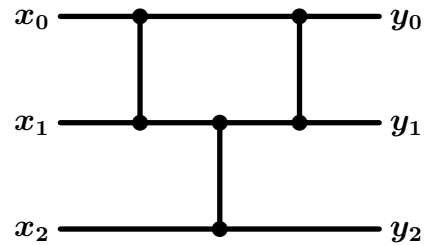
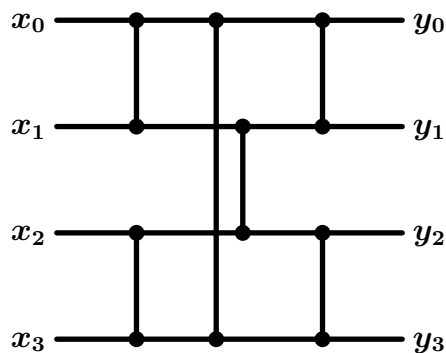
architecture min_and_max of sort_two is
begin
  y0 <= x0 when (x0 < x1) else x1;
  y1 <= x1 when (x0 < x1) else x0;
end min_and_max;
```

A figura ao lado mostra duas maneiras diferentes de representar graficamente uma instanciação desta entidade. Do lado esquerdo apresenta-se uma representação convencional da entidade e do lado direito apresenta-se a representação habitualmente usada em redes de ordenação.



11.2 Como ordenar três ou quatro números inteiros

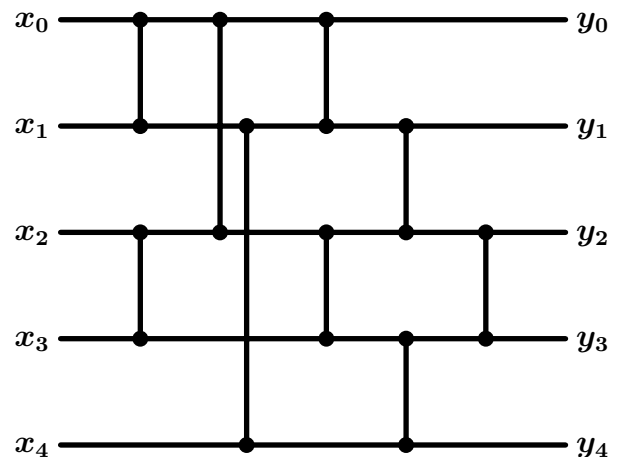
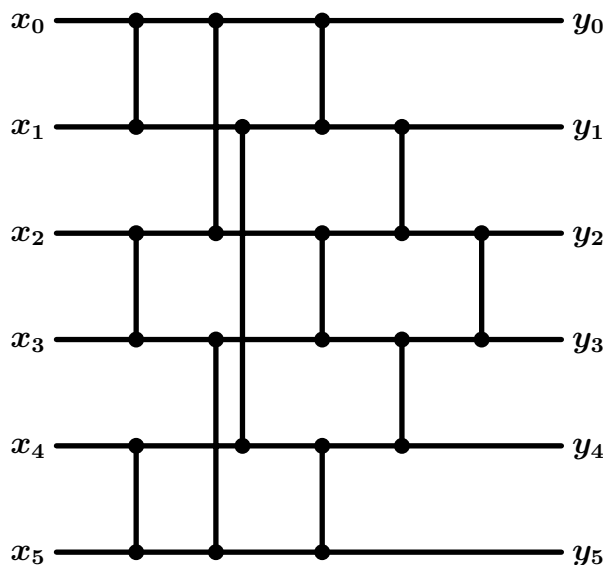
Para ordenar quatro números inteiros podemos usar a rede de ordenação apresentada do lado esquerdo da figura seguinte.



Note que se se eliminar x_3 e y_3 e todas as instanciações da entidade `sort_two` que estão ligadas à linha que os une, se obtém uma rede de ordenação para três números, como pode ser observado do lado direito da figura. Note ainda que o atraso máximo de propagação da lógica combinacional corresponde ao atraso de três entidades `sort_two`.

11.3 Como ordenar cinco ou seis números inteiros

Para ordenar seis números inteiros podemos usar a rede de ordenação apresentada do lado esquerdo da figura seguinte.



Note que se se eliminar x_5 e y_5 e todas as instanciações da entidade `sort_two` que estão ligadas à linha que os une, se obtém uma rede de ordenação para cinco números, como pode ser observado do lado direito da figura. Nestes dois casos o atraso máximo de propagação da lógica combinacional corresponde ao atraso de cinco entidades `sort_two`.

12 Exemplos e demonstrações fornecidos

Para ilustrar o funcionamento das entidades fornecidas foram feitos, em VHDL 2008, os exemplos e as demonstrações que se descrevem a seguir. O código dos exemplos encontra-se no diretório `vhd1_code`, num ficheiro com o nome do exemplo a que se acrescentou `_t1.vhd` (por exemplo, para o exemplo `debouncer_example` o nome do ficheiro é `vhd1_code/debouncer_example_t1.vhd`), e o das demonstrações do diretório `demonstrations`, também num ficheiro a que se acrescentou `_t1.vhd` (por exemplo, para a demonstração `histogram` o nome do ficheiro é `demonstrations/histogram_t1.vhd`). No cabeçalho destes ficheiros encontra-se a lista completa de todos os ficheiros que devem fazer parte do projeto correspondente.

O código dos exemplos é suficientemente simples para ser compreendido na íntegra pelos alunos. O das demonstrações é mais complicado, mas mesmo assim espera-se que possa ser compreendido por uma boa parte dos alunos. Todos os exemplos e demonstrações foram precompilados. Os respetivos ficheiros `.sof` encontram-se armazenados no diretório `sof`.

Apresentamos a seguir uma breve descrição sumária do que cada exemplo ou demonstração faz. Esta descrição também pode ser encontrada no cabeçalho do ficheiro que contém o *top level* do exemplo ou demonstração.

12.1 Exemplo `debouncer_example`

Este exemplo mostra como a entidade `debouncer` pode ser usada para limpar os sinais provenientes dos botões e interruptores do *kit* DE2-115. O seu comportamento é o seguinte.

- Ao carregar em `key(0)` é feito um *reset* a um contador de 8 bits.
- Ao mover `sw(0)` para a posição de ligado é somado 1 ao contador.
- Ao mover `sw(1)` para a posição de ligado é subtraído 1 ao contador.
- Sempre que uma das três ações anteriores é efetuada o *led* verde `ledg(8)` é ligado durante um décimo de segundo.
- O valor do contador é mostrado em binário nos *leds* verdes `ledg(7 downto 0)`.

Neste exemplo existe um sinal de *reset* global que é ativado durante um período de tempo muito curto logo a seguir à FPGA ter sido programada.

12.2 Exemplo `clock_generator_example`

Este exemplo mostra como a entidade `clock_generator` pode ser usada para criar um relógio com uma frequência diferente de 50 MHz. O seu comportamento é o seguinte.

- A entidade `clock_generator` é usada para gerar um relógio de 250 MHz.
- A partir do relógio de 250 MHz é gerado, usando a entidade `pulse_generator`, um pulso com uma frequência de 10 Hz.
- O pulso é usado para incrementar um contador **decimal** de dois algarismos dez vezes por segundo.
- Usando duas instanciações da entidade `seven_segment_decoder`, o valor do contador é mostrado em dois *displays* de sete segmentos.
- O estado dos interruptores `sw(4 downto 0)` é mostrado noutro *display* de sete segmentos (usando mais uma vez uma instanciação da entidade `seven_segment_decoder`, o que permite ver todos os padrões gerados por esta entidade).

12.3 Exemplo `pseudo_random_generator_example`

Este exemplo mostra como a entidade `pseudo_random_generator` pode ser usada. O seu comportamento é o seguinte.

- Duas instanciações da entidade `pseudo_random_generator`, com arquiteturas diferentes, são usadas para gerar dois números pseudo-aleatórios, um com 18 *bits* e o outro com 8 *bits*.
- A entidade `pulse_generator` é utilizada para gerar um pulso com uma frequência de 10 Hz.
- Dez vezes por segundo, os estados dos geradores de números pseudo-aleatórios são transferidos para os *leds* vermelhos e verdes.

12.4 Exemplo `rs232_controller_example`

Este exemplo mostra como a entidade `rs232_controller` pode ser usada. O seu comportamento é o seguinte.

- Quando o interruptor `sw(0)` está desligado, o que é recebido é transmitido (*loop back*).
- Quando o interruptor `sw(0)` está ligado, sempre que o botão `key(0)` é premido é enviado um *byte*. É enviado primeiro `X"20"`, que corresponde no código ASCII a um espaço, da vez seguinte é enviado `X"21"`, e assim sucessivamente.
- O último *byte* recebido é mostrado, em hexadecimal, nos dois *displays* de sete segmentos mais à direita.

Em GNU/Linux pode ser usado o programa `lsd_term`, cujo código fonte se encontra no diretório `c_code`, e um cabo *USB to RS232* para comunicar com o *kit*.

12.5 Exemplo `lcd_controller_example`

Este exemplo mostra como a entidade `lcd_controller` pode ser usada. O seu comportamento é o seguinte.

- É usado um `for ... generate` para instanciar *debouncers* para os quatro botões.
- É usado um `case ... is` com muitos casos possíveis para inicializar e atualizar o LCD.
- Durante a inicialização do LCD a forma dos caracteres com códigos 0 (círculo) e 1 (disco) é redefinida.
- Dois sinais, do tipo `std_logic_vector(127 downto 0)`, são usados para guardar os códigos de todos os caracteres das duas linhas do LCD.
- O conteúdo desses dois sinais está sempre a ser reenviado para o LCD, de modo a que se parte desses sinais for alterada, o LCD pouco depois mostra automaticamente a nova informação.
- Ao premir `key(0)` a linha de cima do LCD é deslocada para a esquerda, sendo inserido um círculo (código 0) na posição mais à direita.
- Ao premir `key(1)` a linha de cima do LCD é deslocada para a esquerda, sendo inserido um disco (código 1) na posição mais à direita.
- Ao premir `key(2)` a linha de cima do LCD é deslocada para a direita, sendo inserido um 'a' (código 97) na posição mais à esquerda.
- Ao premir `key(3)` a linha de cima do LCD é deslocada para a direita, sendo inserido um 'b' (código 98) na posição mais à esquerda.

12.6 Exemplo `ir_decoder_example`

Este exemplo mostra como a entidade `ir_decoder` pode ser usada. O seu comportamento é o seguinte.

- O estado dos três primeiros *leds* verdes é controlado pelas teclas A, B e C do comando remoto do *kit* DE2-115 (caso precise dele para o seu projeto peça um ao seu professor). Cada vez que se carrega numa dessas teclas o estado do *led* correspondente comuta para o outro valor (*toggle*).
- O último comando recebido pela entidade `ir_decoder` é mostrado, em hexadecimal, nos *displays* de sete segmentos. Se o comando recebido estiver claramente errado (se os *bits* 24 a 31 não forem o complemento dos *bits* 16 a 23), os *displays* piscam com uma frequência de 1 Hz.
- É utilizado um gerador de pulsos para produzir o sinal que faz piscar os *displays*.

12.7 Exemplo `ps2_controller_example`

Este exemplo mostra como a parte do teclado da entidade `ps2_controller` pode ser usada. O seu comportamento é o seguinte.

- O *led* verde `ledg(8)` é ligado quando é detetado que um teclado PS/2 foi ligado ao *kit* DE2-115.
- O estado das teclas QWERT do teclado (tecla a ser carregada ou não) é mostrado nos cinco primeiros *leds* vermelhos.

12.8 Exemplo `audio_example`

Este exemplo mostra como as entidades `audio_controller` e `audio_io`, com a ajuda das entidades `sin_function` e `blorp_sound_rom`, podem ser usadas para produzir sons. O seu comportamento é o seguinte.

- Usando a entidade `sin_function`, é produzido um tom puro de 1 kHz, que é enviado para o canal esquerdo de *line-out*.
- Usando a entidade `blorp_sound_rom`, sempre que o botão `key(0)` é carregado, é enviado o som "blorp" para o canal direito de *line-out*.

12.9 Exemplo `audio_volume_example`

Este exemplo, mais complicado que o anterior, mostra como as entidades `audio_controller` e `audio_io` podem ser usadas para trabalhar com sons. O seu comportamento é o seguinte.

- Quando o interruptor `sw(17)` está ligado, um sinal de 1 kHz é enviado para o conversor digital-analógico (isto tem precedência em relação ao que é descrito a seguir).
- Quando o interruptor `sw(16)` está ligado, o sinal `use_line_in` é colocado a '1', pelo que o sinal de áudio *line-in* é usado como entrada do conversor analógico-digital. (Estamos pois a converter o sinal de áudio de entrada para valores digitais.)
- Os interruptores `sw(15 downto 11)` controlam a amplificação que é aplicada ao sinal *line-in*. Valores mais altos significam amplificações maiores.
- Quando o interruptor `sw(10)` está ligado, e o interruptor `sw(16)` está desligado, o sinal `use_mic` é colocado a '1', pelo que o sinal de áudio *mic* é usado como entrada do conversor analógico-digital. (Estamos pois a converter o sinal de áudio do microfone para valores digitais.)

- O interruptor `sw(9)` controla a amplificação a aplicar ao sinal vindo do microfone.
- Quando o interruptor `sw(8)` está ligado, parte do sinal de áudio *line-in* aparece também no sinal de áudio *line-out*.
- Quando o interruptor `sw(7)` está ligado, parte do sinal de áudio *mic* aparece também no sinal de áudio *line-out*.
- Os interruptores `sw(6 down to 0)` controlam a amplificação que é aplicada ao sinal *line-out*. Valores mais altos significam amplificações maiores.
- Os *leds* vermelhos por cima dos interruptores que controlam ganhos estão sempre ligados. Os por cima de interruptores que controlam a fonte do som (*line-in* ou *mic*) piscam a 5 Hz. Os outros estão sempre desligados.
- É usado um relógio de 100 MHz (só para mostrar que isso é possível).

Mais mais detalhes, estude o diagrama funcional da figura 9 (página 21) do [data sheet](#) do codec de áudio.

12.10 Exemplo `sram_controller_example`

Este exemplo mostra como a entidade `sram_controller` pode ser usada para guardar alguns segundos de som. O seu comportamento é o seguinte.

- Utilizando uma posição de escrita na memória SRAM diferente da de leitura (para mais detalhes estude o código fornecido!) o áudio presente na entrada *line-in* é enviado para a saída *line-out* com um atraso de 4 segundos.
- É usado um relógio de 75 MHz (só para confirmar que a entidade `sram_controller` funciona bem a uma frequência diferente de 50 MHz).

Note que como a memória SRAM tem 2^{20} palavras de 16 *bits* é possível armazenar cerca de 10.9 segundos de áudio (dois canais a 48 kHz):

$$10.9 \approx \frac{2^{20}}{2 \times 48000}.$$

12.11 Exemplo `vga_example`

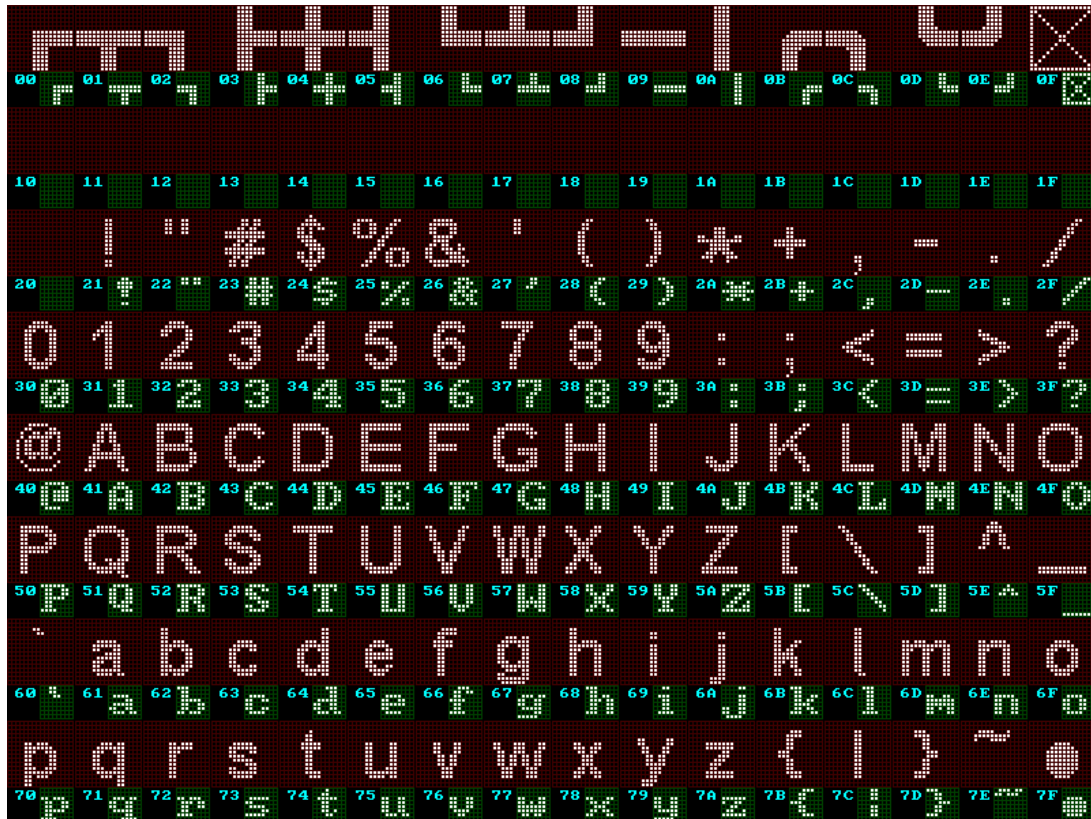
Este exemplo mostra como as entidades `vga_controller`, `vga_output` e `vga_clock_generator` podem ser usadas. O seu comportamento é o seguinte.

- Instancia as três entidades necessárias para gerar uma imagem VGA. Note que o relógio a usar no projeto é o relógio que é usado para gerar a imagem e não `clock_50`.
- É gerada uma imagem com um bordo branco de 16 *pixels*, dentro do qual se colocou um padrão axadrezado a preto e verde escuro.

Os grupos que pretendam usar VGA nos seus projetos podem usar o conteúdo deste projeto como ponto de partida. Recomenda-se vivamente a utilização de um *pipeline* (ver secção 8.4) para gerar uma imagem VGA. Não só é essa a maneira mais elegante de o fazer, como isso em geral permite obter frequências máximas de funcionamento mais elevadas, o que por sua vez permite utilizar modos gráficos de maior resolução.

12.12 Exemplo avançado `font_shapes`

Neste exemplo, mais complicado que os restantes desta secção, gera uma image VGA que mostra a forma de todas as imagens armazenadas nas entidades `font_8x8_bold` e `font_16x16_bold`. Foi utilizada a entidade `screen_capture` para capturar a imagem que é gerada. Como pode ser observado na figura seguinte, existem 16 imagens (índices X"10" a X"1F") que estão vazias (é pois possível lá pôr o que muito bem entender).



12.13 Demonstração `logic_analyzer`

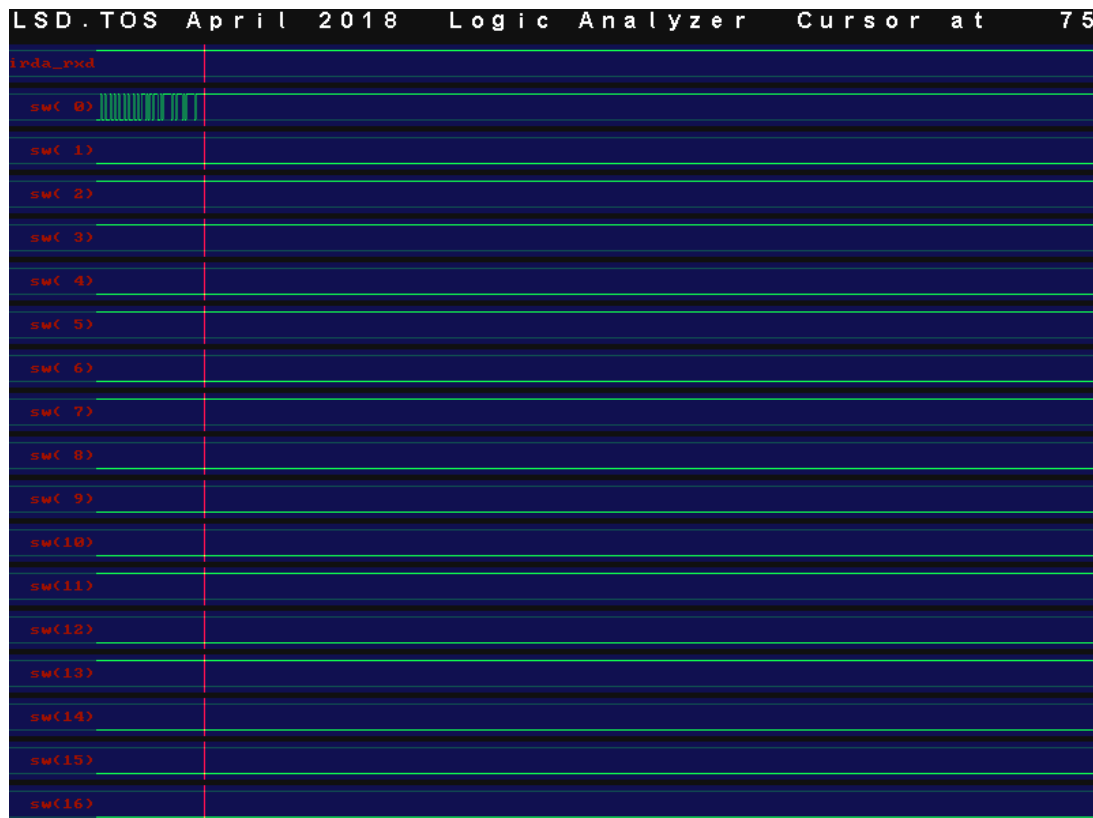
Nesta demonstração a forma de onda dos sinais nos pinos de entrada `irda_rxd` e `sw(17 downto 0)` da FPGA é mostrado numa imagem VGA (como os sinais só podem tomar os valores '0' e '1', chama-se a este tipo de visualizador um analisador lógico). O seu comportamento é o seguinte.

- Os sinais de entrada da FPGA `irda_rxd` e `sw(17 downto 0)`, e um sinal de relógio de 1kHz, são amostrados a 100 kHz.
- A imagem VGA é atualizada sempre que um destes sinais muda ou quando se carrega em `key(2)`.
- É feito um *reset* ao sinal de relógio de 1 kHz quando o analisador lógico começa a gravar. (Deste modo este sinal de relógio dá uma boa ajuda visual sobre a escala temporal do que está a ser mostrado na imagem.)
- O analisador lógico guarda 8192 amostras de cada sinal.
- O cursor vermelho vertical pode ser movido para a direita usando o botão `key(0)`.
- O cursor vermelho vertical pode ser movido para a esquerda usando o botão `key(1)`.
- A imagem é deslocada para a direita ou para a esquerda (*scroll*) de modo a manter o cursor sempre dentro da zona visível da imagem.

- A posição do cursor é mostrada, em base 10, na zona de cima da imagem.
- O botão key(3), quando carregado, impede qualquer modificação da imagem (útil quando se pretende capturar uma imagem).

O código da entidade *top level* desta demonstração está escrito de modo a que seja simples mudar quer a frequência de amostragem dos sinais a visualizar quer a lista dos próprios sinais a visualizar.

O que esta demonstração faz é parecido com o que o *Signal Tap Analyzer* faz, só que funciona de uma maneira interativa (logo em muitos casos muito mais prática). Na figura seguinte mostra-se o aspeto típico do analisador lógico quando se usa a resolução 800x600@72Hz.



Note que como neste modo gráfico apenas se vêem 600 linhas as últimas formas de onda (correspondentes a sw(17) e ao sinal de relógio de 1 kHz) não aparecem. Pode ser observado *bouncing* em sw(0) com uma duração de cerca de 0.7 mili-segundos (com os sinais amostrados a 100 kHz, um *pixel* corresponde a 0.01 mili-segundos). No instante em que se ligou sw(0) alguns dos interruptores estavam ligados e os outros desligados.

Na figura seguinte é apresentado um exemplo extremo de *bouncing* no interruptor sw(0). Neste caso foi usado o modo gráfico 1400x900@60Hz e depois foi usando o programa gimp para remover da imagem partes irrelevantes. Note que, de acordo com o relógio de 1 kHz, o bouncing demora no total 13 mili-segundos e que neste caso existem muitas comutações nos primeiros 2 mili-segundos.

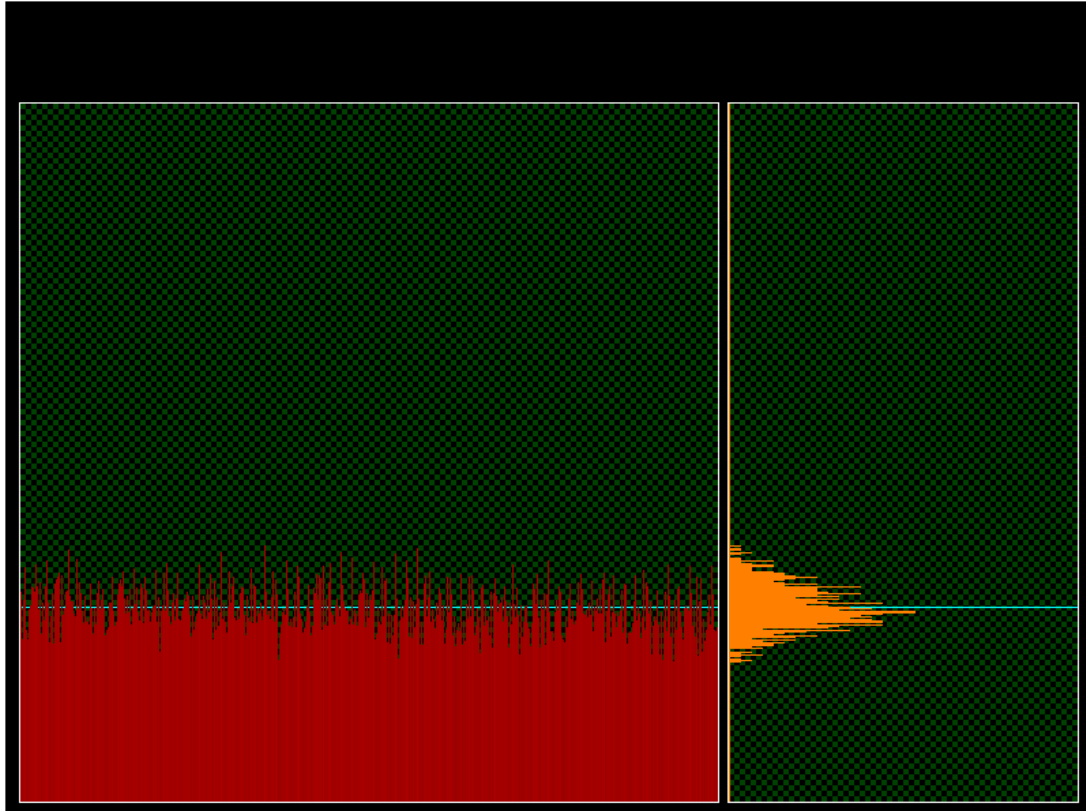


12.14 Demonstração histogram

Esta demonstração serve para verificar experimentalmente que o gerador de números pseudo-aleatórios da entidade `pseudo_random_generator` gera de facto números distribuídos uniformemente. O seu comportamento é o seguinte.

- Quando se carrega em `key(0)` é gerado um novo número pseudo-aleatório de 1 a 6, que é mostrado no *display* de sete segmentos do lado direito, sendo os números pseudo-aleatórios gerados anteriormente são deslocados uma posição para a esquerda.
- No lado esquerdo da imagem, é mostrado um histograma de números aleatórios de 9 *bits*, histograma esse que vai evoluindo ao longo do tempo à medida que vão sendo gerados mais números pseudo-aleatórios.
- Apenas para a imagem ficar menos monótona, a cor de cada barra do histograma depende da sua altura.
- A média dos números pseudo-aleatórios também é mostrada (linha a azul claro).
- No lado direito da imagem é desenhado um histograma do histograma. No início as barras do primeiro histograma tem quase todas a mesma altura, pelo que este segundo histograma está muito concentrado. Ao longo do tempo as alturas das barras do primeiro histograma vão-se dispersando cada vez mais, pelo que o segundo histograma fica também mais disperso.
- Quando o botão `key(1)` está carregado, o histograma não é atualizado.

Na figura seguinte mostra-se o aspeto típico desta demonstração pouco depois de a FPGA ter sido programada.

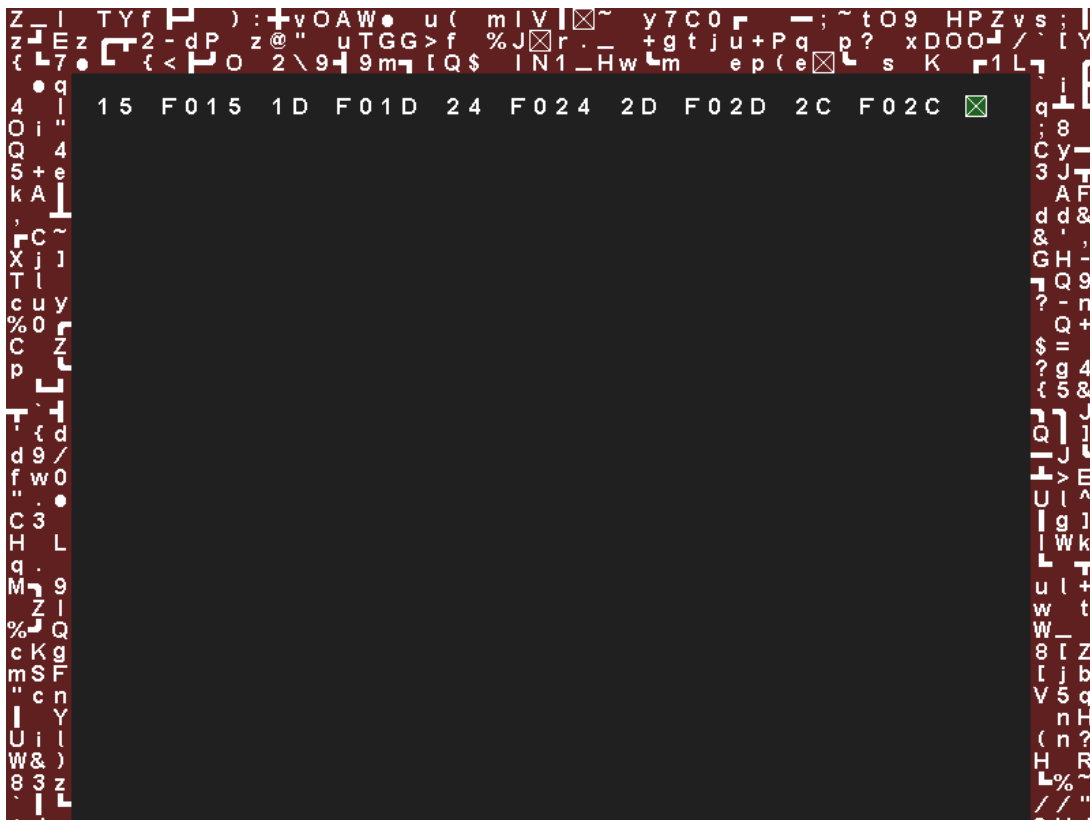


12.15 Demonstração `text_buffer`

Esta demonstração mostra mais uma utilização das entidades `ps2_controller` e `vga_controller` em funcionamento. O seu comportamento é o seguinte.

- Se for detetado um teclado PS/2 então
 - o *led* verde `ledg(0)` é aceso,
 - os interruptores `sw(2 downto 0)` controlam os três *leds* do teclado, e
 - os últimos dois *bytes* recebidos do teclado são mostrados em hexadecimal nos quatro *displays* de sete segmentos do lado direito.
- Se for detetado um rato PS/2 então
 - o *led* verde `ledg(1)` é aceso,
 - os *leds* verdes `ledg(7 downto 5)` mostram o estado dos botões do rato, e
 - os quatro *displays* de sete segmentos do lado esquerdo mostram, em hexadecimal, as coordenadas X e Y acumuladas do movimento do rato.
- Em ambos os caso, o *led* verde `ledg(8)` é ligado durante um décimo de segundo sempre que é recebida informação do dispositivo PS/2.
- Tanto o controlador PS/2 como o VGA são reinicializados quando o interruptor `sw(3)` estiver ligado.
- Parte que um ecrã de texto, com um bordo inicializado com caracteres pseudo-aleatórios, é mostrado no ecrã.
- Todos os bytes recebidos do teclado PS/2 são colocados no ecrã de texto.
- A zona visível do ecrã de texto pode ser deslocada para cima e para baixo quer através dos interruptores `sw(4)` e `sw(5)`, quer através de movimentos do rato.

Na figura seguinte mostra-se o aspeto desta demonstração depois das teclas QWERT de um teclado PS/2 terem sido carregadas.



13 Outros ficheiros de apoio (apenas para GNU/Linux)

É recomendado que os ficheiros executáveis do quartus possam ser usados sem introduzir o seu caminho completo. Uma maneira de fazer isso consiste em acrescentar as linhas de código seguintes

```
if [ -e /opt/quartus/17.1/quartus/bin ]; then
  export PATH=$PATH:/opt/quartus/17.1/quartus/bin
fi
```

no fim do ficheiro `.bashrc` que reside no diretório de entrada do utilizador (substitua `/opt/quartus/17.1` pelo caminho que usou na instalação do quartus).

13.1 Scripts de apoio

No diretório `bin` podem-se encontrar os seguintes *scripts* bash, a serem usado a partir de um terminal:

clean_project Este *script* apaga, no diretório corrente, todos os ficheiros temporários criados pelo quartus.

compile Este *script* compila um projeto. Tem como único argumento o nome do projeto. É pressuposto que esse nome seja também o nome do diretório onde está armazenado o projeto.

create_project Este *script* cria um diretório com o nome do projeto, que é o primeiro argumento do *script*. Os restantes argumentos são nomes dos ficheiros `.vhd` que vão fazer parte do projeto. Se esses ficheiros existirem serão **copiados** para o diretório do projeto e, se não existirem, serão criados no diretório do projeto. É pressuposto que a entidade *top level* do projeto tenha um nome que é o nome do projeto a que se acrescentou `_t1`.

program_sof Este *script* permite programar a FPGA com um ficheiro `.sof`. Tem como único argumento o nome desse ficheiro. (Não se esqueça de colocar o interruptor localizado no canto inferior esquerdo do *kit* na posição RUN.)

program_pof Este *script* permite programar a FPGA com um ficheiro `.pof`. Tem como único argumento o nome desse ficheiro. (Não se esqueça de colocar o interruptor localizado no canto inferior esquerdo do *kit* na posição PROG.)

Por exemplo, partindo do diretório de raiz onde está guardado o material de apoio, para criar e compilar o exemplo `debouncer_example` basta fazer, num terminal:

```
> bin/create_project debouncer_example vhd1_code/debouncer_example_t1.vhd vhd1_code/debouncer.vhd
> bin/compile debouncer_example
```

(Relembra-se que pode encontrar a lista dos ficheiros de devem fazer parte do projeto no cabeçalho do ficheiro que contém a entidade *top level*, que neste caso é `debouncer_example_t1.vhd`.) Depois disso, para programar a FPGA basta fazer:

```
> bin/program_sof debouncer_example/output_files/debouncer_example.sof
```

13.2 Makefiles

No diretório de raiz do material de apoio encontra-se um ficheiro com o nome `makefile`. Este ficheiro contém instruções para o programa `make` que permitem automatizar algumas tarefas. Por exemplo, é possível criar e compilar o exemplo `debouncer_example` através do comando

```
> make debouncer_example
```

13.3 Programas de apoio

No diretório `c_code` pode-se encontrar o código fonte dos seguintes programas:

lsd_term Este programa permite comunicar com o *kit* DE2-115 através de um cabo *USB to RS232*. Pode ser compilado usando o comando (no diretório onde está o código fonte do programa):

```
> make lsd_term
```

O seu texto de ajuda é o seguinte:

Terminal emulator (LSD.TOS, April 2018)
DETI, Universidade de Aveiro

Usage: `lsd_term [-t] [-d device_name] [com_spec]`

`com_spec` is either
 `baud_rate`

or
 `baud_rate,parity,data_bits,stop_bits`

`baud_rate` is one of 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
 115200, 230400, 460800, 921600, 1152000

`parity` is one of N (none), E (even), O (odd)

`data_bits` is one of 5, 6, 7, 8

`stop_bits` is one of 1, 2

the default `com_spec` is 115200,N,8,1
the default `device_name` is `/dev/ttyUSB0`

the `-t` option turns on test mode (one character displayed per line)

pseudo_random_generator Este programa gera automaticamente as duas arquiteturas da entidade `pseudo_random_generator`.

screen_capture Este programa é usado para capturar imagens VGA.