Circuitos Combinacionales en FPGA

Introducción a la Microfabricación y las FPGA

Instituto Balseiro

26 de Agosto 2013

La clase pasada

- Introducción a FPGA: qué son, para qué se usan, arquitecturas, cómo se programan.
- Flujo de diseño basado en esquemático y en HDL.
- Introducción a VHDL.
  - VHDL para síntesis: nuestros primeros programas a nivel de gates.
  - VHDL para simulación: nuestros primeros testbenchs. Assert.
- Herramientas:
  - *ISE*: Crear proyecto, fuente, ucf (user constraint file) y todo el proceso de síntesis hasta generar el archivo de programa (bitstream). Reportes.
  - *ISim*: Usando el ISE, crear un testbench y simularlo con el ISE Simulator (ISim).
  - *Impact*: Configurar la FPGA.

Diseño y codificación hasta ahora

- Diseño a nivel de gate.
  - Hacemos tabla de verdad y utilizamos suma de productos.
  - Si podemos dividir en bloques mas pequeños (o ya tenemos bloques mas pequeños): instanciación.
- Library y package. Entidad. Arquitectura.
- Todas las sentencias que vimos hasta ahora son **concurrentes**

Hoy veremos: Register Transfer Level inicial

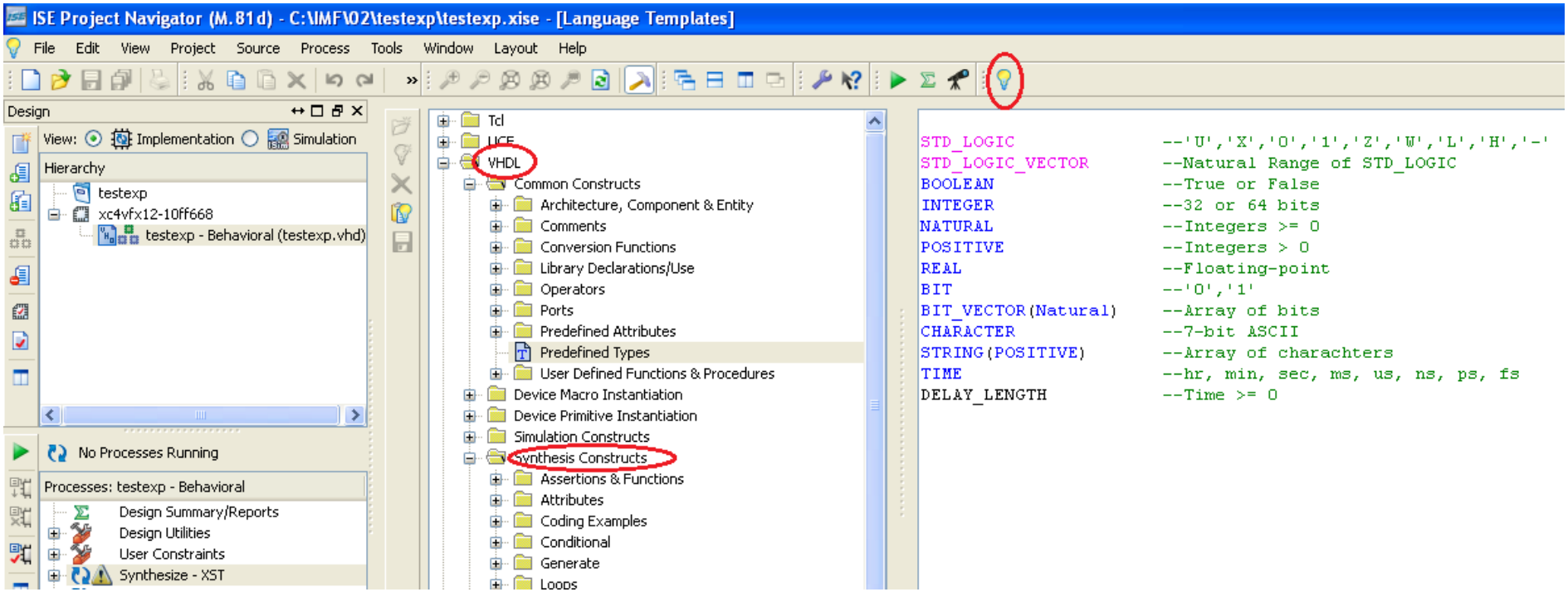
- RTL: Metodologia de diseño pensando la manipulación de datos como transferencia entre registros.
- Hoy vamos a ver las componentes de la Logica Combinacional a este nivel de abstracción (mayor que el de gate).
- Nuevos tipos de datos. Nuevos operadores. Ruteo de señales.

Tipos nativos

```
STD_LOGIC               --'U','X','0','1','Z','W','L','H','-'
STD_LOGIC_VECTOR        --Natural Range of STD_LOGIC
BOOLEAN                 --True or False
INTEGER                 --32 or 64 bits
NATURAL                 --Integers >= 0
POSITIVE                --Integers > 0
REAL                    --Floating-point
BIT                     --'0','1'
BIT_VECTOR(Natural)     --Array of bits
CHARACTER               --7-bit ASCII
STRING(POSITIVE)        --Array of charachters
TIME                    --hr, min, sec, ms, us, ns, ps, fs
DELAY_LENGTH            --Time >= 0
```

- `real`, `time`, `delay_length` NO sintetizables.
- No todas las operaciones sobre estos tipos son sintetizables.

Referencia del lenguaje en ISE



Capitulo 21 del Ashenden.

Operadores nativos

Operator	Description	Data type of operands	Data type of result
`a ** b`	exponentiation	`integer`	`integer`
`a * b`	multiplication		
`a / b`	division	*integer type for constants and array boundaries, not synthesis*	
`a + b`	addition		
`a - b`	subtraction		
`a & b`	concatenation	1-D array, element	1-D array
`a = b`	equal to	any	`boolean`
`a /= b`	not equal to		
`a < b`	less than	scalar or 1-D array	`boolean`
`a <= b`	less than or equal to		
`a > b`	greater than		
`a >= b`	greater than or equal to		
not `a`	negation	`boolean`, `std_logic`, `std_logic_vector`	same as operand
`a` **and** `b`	and		
`a` **or** `b`	or		
`a` **xor** `b`	xor		

- Tipos nativos `integer`, `natural`: 32 bits.
- En general queremos mayor manejo de el numero de bits en cada señal: `IEEE.numeric_std`

Operadores overloaded signed y unsigned

Overloaded operator	Description	Data type of operands	Data type of result
a * b a + b a - b	arithmetic operation	unsigned, natural signed, integer	unsigned signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned, natural signed, integer	boolean boolean

```
library ieee;
use ieee.numeric.std.all;
signal a : unsigned(3 downto 0);
```

Casting

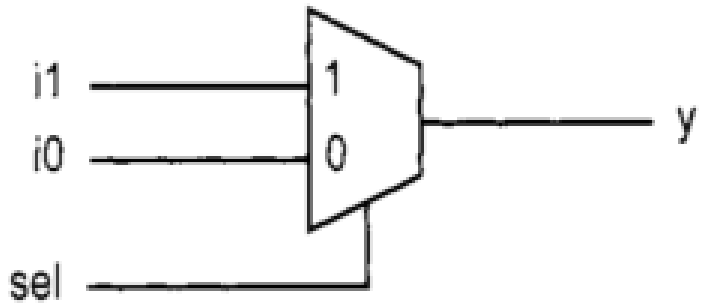
Data type of a	To data type	Conversion function/type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Ruteo

- Hasta ahora, sabemos trabajar con algunos tipos sintetizables y realizar operaciones lógicas, artiméticas y relacionales.
- Nos falta ver cómo seleccionar de diferentes posibles resultados, similar a las sentencias **if** y **case**.
- En hardware, esto se hace *evaluando* todos los posibles resultados en paralelo y *ruteando* el seleccionado a la salida. Esta selección se implementa con *multiplexores*.
- Vamos a ver dos maneras de escribir en VHDL sentencias concurrentes que generan dos modos distintos de ruteo: priority network y

Conditional Signal Assignment - Priority Network

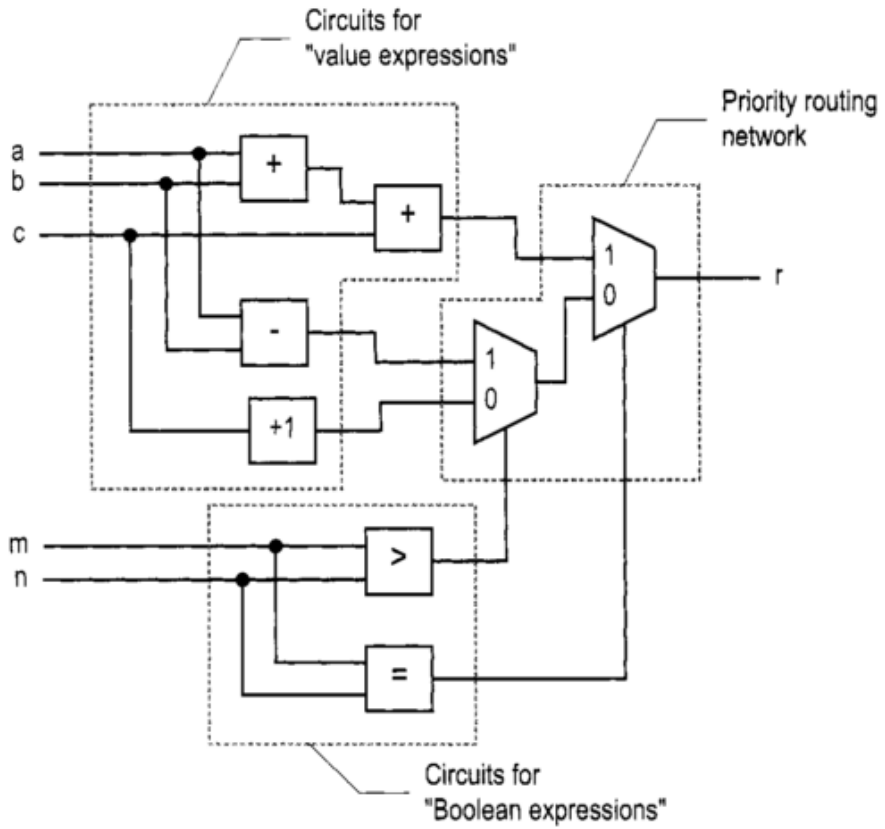
```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
                  . . .
               value_expr_n;
```



sel	y
0 (false)	i0
1 (true)	i1

Conditional Signal Assignment - Priority Network

```
r <= a + b + c when m = n else
     a - b     when m > n else
     c + 1;
```



- Expresiones de valor Y booleanas se evalúan *concurrentemente*.
- Las expresiones booleanas setean las señales de selección de los multiplexores.
- Cada clausula `when-else` introduce un Mux2-1: cuántas mas cláusulas, más larga la cadena y por lo tanto mayores delays.

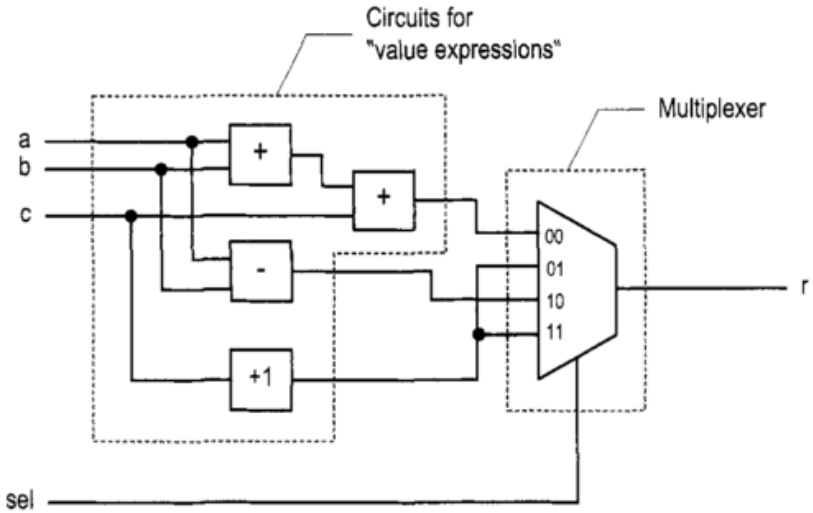
Selected signal assignment

```
with sel select
   sig <= value_expr_1 when choice_1,
          value_expr_2 when choice_2,
          value_expr_3 when choice_3,
          . . .
          value_expr_n when others;
```

- Un `choice` tiene que ser un valor possible de `sel`.
- Tienen que estar todos los valores y ser mutuamente excluyentes. **others**.
- En general son `std_logic_vector` o tipos enumerados.

Selected signal assignment

```
signal sel: std_logic_vector(1 downto 0);
. . .
with sel select
   r <= a + b + c    when "00",
        a - b        when "10",
        c + 1        when others;
```



- Infiere un multiplexor de 2^2 en 1, con `sel` como señal de selección.
- El tamaño del multiplexor crece geométricamente con la cant de bits de `sel`.
- Nuevamente, las expresiones de valor se evalúan concurrentemente.

Constantes

Para limites de arreglos y expresiones. Sintaxis:

```
constant const_name : data_type := value_expression;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity add_w_carry is
   port(
      a, b: in std_logic_vector(3 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(3 downto 0)
   );
end add_w_carry;

architecture hard_arch of add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(4 downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(3 downto 0));
   cout <= sum_ext(4);
end hard_arch;
```

Sumador con constante

```
architecture const_arch of add_w_carry is
    constant N: integer := 4;
    signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
    a_ext <= unsigned('0' & a);
    b_ext <= unsigned('0' & b);
    sum_ext <= a_ext + b_ext;
    sum <= std_logic_vector(sum_ext(N-1 downto 0));
    cout <= sum_ext(N);
end const_arch;
```

Genéricos

Con las constantes, es mas fácil de leer, pero si quiero instanciar un sumador de 5 bits, el de 4 bits no me sirve.
Los `generic` son parámetros de los módulos: me permiten instanciar módulos con la misma estructura, pero distintos tamaños.

```
entity entity_name is
    generic(
        generic_name: data_type := default_values;
        generic_name: data_type := default_values;
        . . .
        generic_name: data_type := default_values
    )
    port(
        port_name: mode data_type;
        . . .
    );
end entity_name;
```

Sumador con Generic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
   generic(N: integer:=4);
   port(
      a, b: in std_logic_vector(N-1 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(N-1 downto 0)
   );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(N-1 downto 0));
   cout <= sum_ext(N);
end arch;
```

Sumador con Generic instanciado

```
signal a4, b4, sum4: unsigned(3 downto 0);
signal a8, b8, sum8: unsigned(7 downto 0);
signal a16, b16, sum16: unsigned(15 downto 0);
signal c4, c8, c16: std_logic;
. . .
-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(arch)
   generic map(N=>8)
   port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));
-- instantiate 16-bit adder
adder_16_unit: work.gen_add_w_carry(arch)
   generic map(N=>16)
   port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));
-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
   port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
```

Procesos

- Hasta ahora, todas las sentencias son concurrentes.
- Procesos: es una sentencia concurrente, pero dentro tiene una cantidad de sentencias secuenciales que describen su comportamiento.
- Sirven para simplificar la tarea de diseño. Pero hay que tener cuidado: nunca hay que perder de vista que aún estas "sentencias secuenciales" están sintetizando hardware concurrente.
- Siempre pensar en qué se está sintetizando.

```
process(sensitivity_list)
begin
   sequential statement;
   sequential statement;
   . . .
end process;
```

Ruteo con *if*

```
if boolean_expr_1 then
   sequential_statements;
elsif boolean_expr_2 then
   sequential_statements;
elsif boolean_expr_3 then
   sequential_statements;
. . .
else
   sequential_statements;
end if;
```

- Genera cascada de Mux2-1 como el conditional signal assignment.
- Son equivalentes si cada branch del if tiene una sola asignación a la misma señal.
- El *if* es mas potente: permite *cualquier número* y *cualquier tipo* de sentencias secuenciales en cada branch.
- Por ejemplo, permite nested ifs.

Ruteo con *case*

```
case sel is
   when choice_1 =>
      sequential statements;
   when choice_2 =>
      sequential statements;
   . . .
   when others =>
      sequential statements;
end case;
```

- Genera multiplexor similar al selected signal assignment.
- Son equivalentes si cada branch del case tiene una sola asignación a la misma señal.
- El *case* es mas potente: permite *cualquier número* y *cualquier tipo* de sentencias secuenciales en cada branch.
- Por ejemplo, permite nested cases.

Error: inferencia de memoria

- El standard VHDL especifica que toda señal *mantiene su valor anterior* si no es asignada.
- En el proceso de síntesis esto infiere un latch.
- Reglas para prevenir la inferencia de memoria no intencionada:
  - Todas las señales de input tienen que estar en la lista de sensitividad.
  - Incluir todos los else-branch en los ifs.
  - Asignar un valor a cada señal en cada branch.

Error: inferencia de memoria

```
process(a)
begin
    if (a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    end if;
end process;
```

Tiene todos los errores

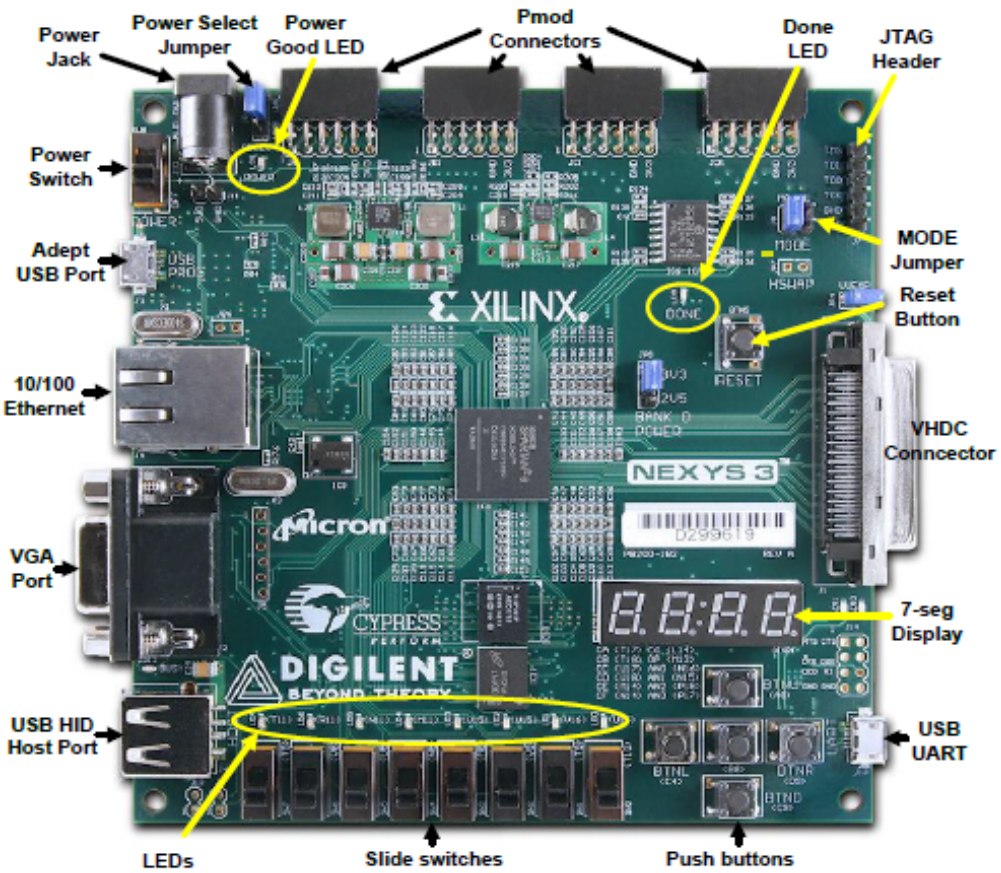
- Señal `b` no está en la lista de sensibilidad.
- `eq` no asignada en primer branch.
- `gt` no asignada en segundo branch.
- El branch `else` no está.

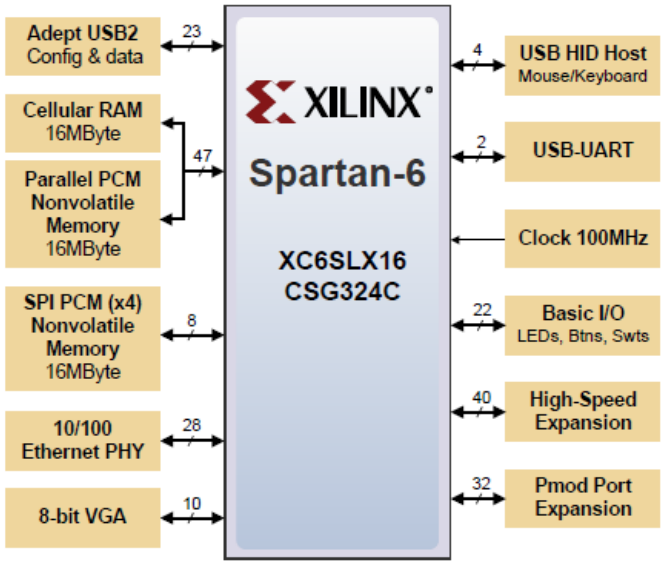
Error: inferencia de memoria corregido

```
process(a,b)
begin
    if (a > b) then
        gt <= '1';
        eq <= '0';
    elsif (a = b) then
        gt <= '0';
        eq <= '1';
    else
        gt <= '0';
        eq <= '0';
    end if;
end process;
```

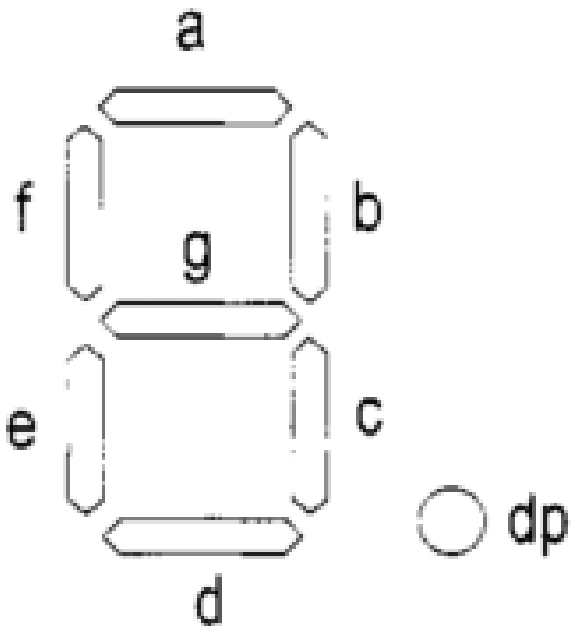
```
process(a,b)
begin
    gt <= '0';
    eq <= '0';
    if (a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    end if;
end process;
```

Nexys3

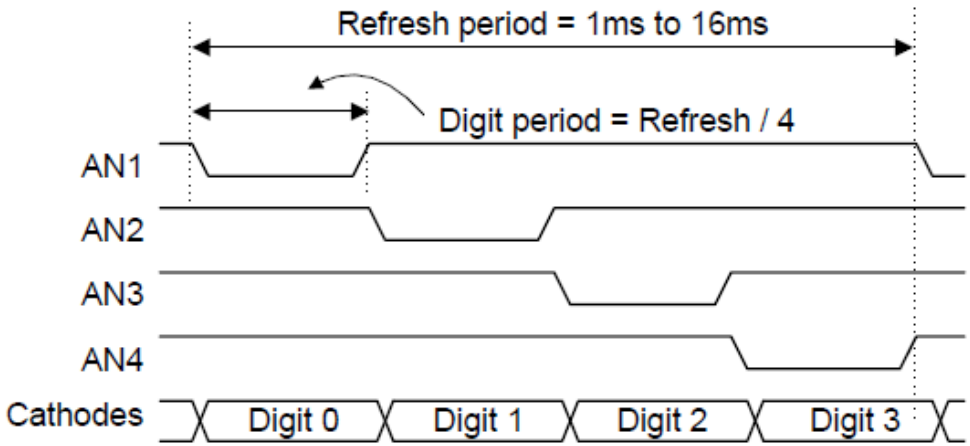

Power Jack
Power Select Jumper
Power Good LED
Pmod Connectors
Done LED
JTAG Header
Power Switch
Adept USB Port
MODE Jumper
Reset Button
10/100 Ethernet
VHDC Conncector
VGA Port
7-seg Display
USB HID Host Port
USB UART
LEDs
Slide switches
Push buttons
XILINX
NEXYS 3
DIGILENT


Adept USB2 Config & data
23
Cellular RAM 16MByte
Parallel PCM Nonvolatile Memory 16MByte
47
SPI PCM (x4) Nonvolatile Memory 16MByte
8
10/100 Ethernet PHY
28
8-bit VGA
10
XILINX
Spartan-6
XC6SLX16 CSG324C
4
USB HID Host Mouse/Keyboard
2
USB-UART
Clock 100MHz
22
Basic I/O LEDs, Btns, Swts
40
High-Speed Expansion
32
Pmod Port Expansion

7 segmentos



- 15 números (0 a F).
- Cada barrita está prendida si tiene un 0.
- Los 4 7seg están multiplexados: selección an0, an1, an2, an3.



A codificar

A codificar...

Ejercicios

- Sumador de signo y magnitud de 4 bits (1 bit signo, 3 bits magnitud). Mostrar signo en led, magnitud en 7seg an0.
- ...

En resumen

Diseño a nivel de Register Transfer inicial, sólo combinacional.

- Continuando con sentencias concurrentes:
  - Tipos de datos. Operaciones aritméticas y relacionales. Overloading y casting.
  - Ruteo concurrente: conditional signal assig (*when* - ruteo con prioridad) y selected signal assig (*select* - gran multiplexor).
  - Constantes y Genéricos
- Agregamos procesos: sentencias secuenciales!
  - Ruteo dentro de procesos: *if* y *case*
  - Reglas para que el proceso infiera un circuito combinacional (i.e., sin memoria).

Ejemplos usando la Nexys3.