Diseño para Síntesis

Sergio López Buedo, Elías Todorovich
Septiembre 2012
etodorov@exa.unicen.edu.ar

Universidad Nacional del Centro
de la Provincia de Buenos Aires

Indice

- Introducción a la codificación para síntesis.
- Herramientas y opciones.
- Ejemplos de Inferencia en XST.

Diferencias entre simulación y síntesis

- En el simulador los procesos tienen una funcionalidad determinada por la semántica de VHDL.
  - Se activan cuando lo indica la lista de sensibilidad.
  - Se paran en las sentencias wait, o cuando llegan al end process.
- En la síntesis se infiere un circuito a partir del código del proceso.
  - No es posible inferir un circuito de cualquier código VHDL
    - Límites del sintetizador
    - Límites de la tecnología (P.ej. wait for 10 ns)

Ejemplos de estructuras no sintetizables

- Especificaciones de tiempo
  - Limitación tecnológica

```
a <= '1' after 10 ns; wait for 5 ns;
```

- Diseños con varios relojes entremezclados
  - Hay que usar siempre los axiomas del diseño síncrono

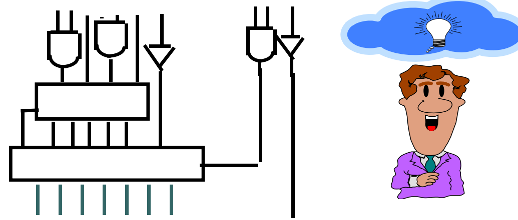
```
process(clk1, clk2)
```

- Flip-flops que capturen en subida y en bajada
  - No existen como tales en la FPGA, se juntan subida y bajada

```
if clk'event then
```

Estilo de código ideal

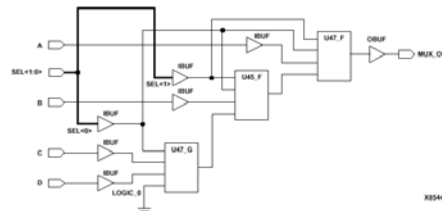
- Al codificar, hay que tener en cuenta el HW que se está modelando



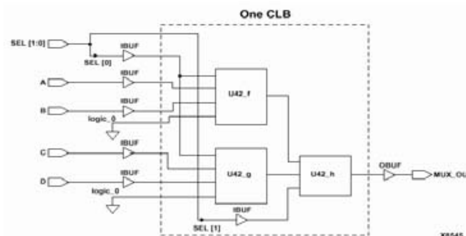
- Regla de oro: si no sé que HW quiero generar, lo más probable es que la herramienta de síntesis tampoco.
- Debe hacerse un análisis previo del flujo de datos, para crear un código RTL.

IF-THEN frente a CASE

- IF-THEN



- CASE produce una lógica balanceada



- En ningún caso hay que olvidarse del valor por defecto

Adaptándose a la arquitectura de la FPGA

- Usar elementos primitivos de la FPGA
  - Flip-flops D …en vez de JK o RS
  - No usar latches si no están disponibles en el dispositivo
  - No usar gated clocks, utilizar el clock enable
- Tener siempre en cuenta la arquitectura de la FPGA
  - Evitar funciones lógicas de muchas entradas
  - Dejar hacer a la herramienta para los circuitos 'estándar'
- Emplear los recursos embebidos
  - Memorias (BlockRAM)
  - Multiplicadores /Bloques DSP
  - ¡Asegurarse de que la herramienta de síntesis los usa!
    - Existen unos umbrales a partir de los cuales se usan, no siempre bien documentados.

Adaptándose a lo que busca el sintetizador

- Muy importante: tener en cuenta como infiere el sintetizador.
- Hay que tener siempre a mano el manual de referencia.

XILINX

Chapter 2

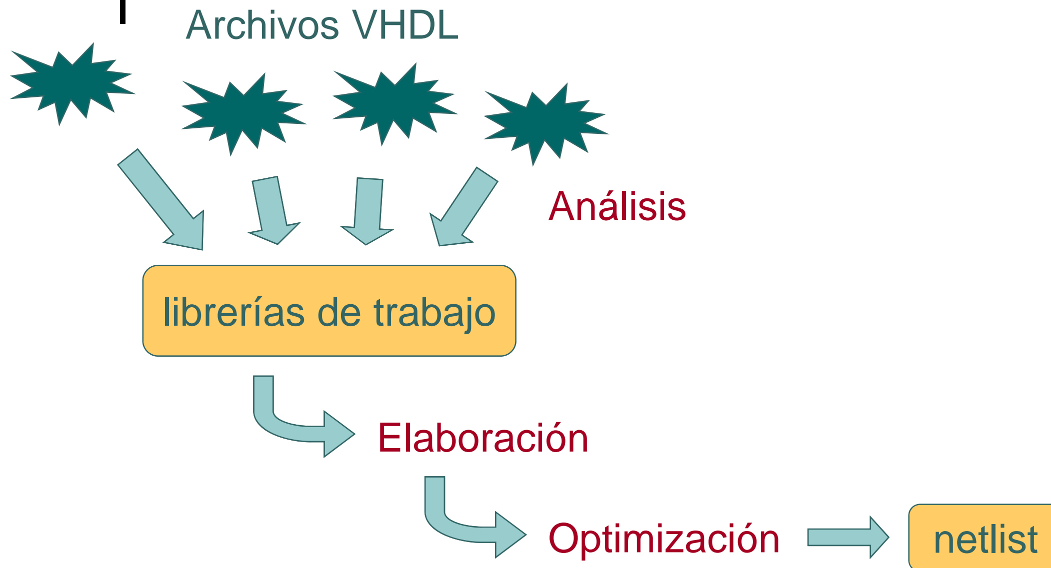
HDL Coding Techniques

This chapter contains the following sections:

- "Introduction"
- "Signed/Unsigned Support"
- "Registers"

- Código en principio dependiente de la herramienta
  - La realidad es que todos los sintetizadores funcionan muy parecido
  - Problema: opciones y restricciones (si que son propietarias)

Pasos de la síntesis



Indice

- Introducción a la codificación para síntesis.
- Herramientas y opciones.
- Ejemplos de Inferencia en XST.

Herramientas comerciales de síntesis

- Synopsys (adquiere a Synplicity)
  - Synplify
  - Synplify Pro
- Mentor Graphics
  - Precision
  - LeonardoSpectrum
- Xilinx Synthesis Technology (XST)

Estrategias de síntesis

- Desde el Project Navigator
  - Diseños sencillos, sin problemas
  - Síntesis transparente
  - Para usar XST
- Adicionalmente (Project Navigator)
  - Diseños más complejos
  - Acceso detallado a los reports/esquemáticos
  - Editores de archivos de restricciones (*constraint files*)
- Scripts TCL
  - El orden de compilación no puede ser establecido manualmente.
  - Un script permanece como documentación, la información de las ventanas de configuración se pierde.

Opciones de síntesis más comunes

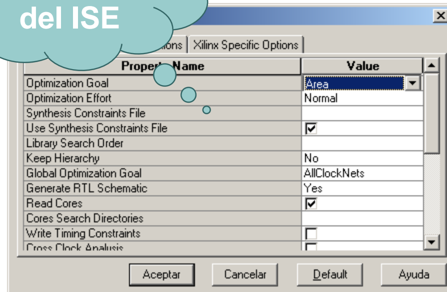
- Optimización
  - Área
  - Velocidad
- Esfuerzo
  - Alto
  - Bajo
- Compartir recursos
- Preservar la jerarquía
- Síntesis de las máquinas de estados
- Frecuencia de reloj objetivo.
- Tipo de dispositivo.
- No añadir pads de entrada/salida, no usar registros de los IOBs.
- Máximo fanout admisible.

Opciones específicas de XST

- Extracción de componentes
  - Si XST infiere alguno de los componentes para los que dispone de macros optimizadas, puede usar o no estas macros dependiendo de esta opción
  - Estos componentes son:
    - Memorias RAM, ROM
    - Circuitos aritméticos
    - Multiplexores, decodificadores, codificadores con prioridad, desplazadores
    - Registros de desplazamiento
    - Clock enable complejos
- Duplicar registros / eliminar registros equivalentes.
- Balanceo de registros en un pipeline
  - Es posible mover también la primera y/o la última etapa

Como especificar las opciones en XST

Dentro del ISE



Embebidas en el código

```
architecture Behavioral of priority2 is

attribute priority_extract: string;
attribute priority_extract of code : signal is "force";

begin

code <= "000" when sel(0) = '1' else
        "001" when sel(1) = '1' else
        "010" when sel(2) = '1' else
        "011" when sel(3) = '1' else
        "100" when sel(4) = '1' else
        "101" when sel(5) = '1' else
        "110" when sel(6) = '1' else
```

En un archivo separado

```
BEGIN MODEL crc32
INST stopwatch opt_mode = area;
INST U2 ram_style = bloc;
NET myclock clock_buffer = true;
NET data_in iob = true;
END;
```

Atributos y pragmas en el código VHDL

- Las opciones de síntesis se pueden especificar como atributos dentro del código VHDL

```
ATTRIBUTE opt_mode: string;
ATTRIBUTE opt_mode OF stopwatch : ENTITY IS "area";
```

- De esta manera se pueden definir distintos atributos para cada módulo, o componente del diseño

```
ATTRIBUTE mux_style: string;
ATTRIBUTE mux_style OF mux_out : SIGNAL IS "muxf";
```

- Los atributos se escriben en la parte declarativa de la aequitectura (antes del *begin*) o en la entidad
- Muy útil: desactivar la síntesis en un trozo de código

```
-- pragma translate_off
-- pragma translate_on
```

Archivos de restricciones XCF

- Las opciones de síntesis también se pueden especificar en un archivo de restricciones:
  - Archivo XCF, heredero del UCF
  - Antiguo CST, formato similar a VHDL
- Las opciones se identifican por la palabra clave MODEL,
  - Para diferenciarlas de las opciones de implementación
- Se pueden definir independientemente para cada módulo

```
BEGIN MODEL crc32
INST stopwatch opt_mode = area;
INST U2 ram_style = bloc;
NET myclock clock_buffer = true;
NET data_in iob = true;
END;
```

- Los atributos definidos para una entidad se aplican a todas las instancias de esa entidad

Indice

- Introducción a la codificación para síntesis.
- Herramientas y opciones.
- Ejemplos de Inferencia en XST.

Inferencia de circuitos aritméticos

- XST es capaz de inferir todos estos circuitos aritméticos
  - Sumadores y restadores, con y sin acarreo
  - Comparadores de igualdad y magnitud, con y sin signo
  - Multiplicadores y divisores (sólo por potencias de dos)
- Modelar sumadores con acarreo de entrada es trivial

```
res <= a + b + cin;
```

  donde a y b son vectores, y cin un escalar
- Para el acarreo de salida se puede hacer un sumador de un bit más, ese bit extra en el resultado será el acarreo

```
res <= ('0' & a) + ('0' & b);
cout <= res(res'left);
```

Aritmética: Sumadores/Restadores

- XST infiere también sumadores/restadores

```
res <= (a + b) when op='0' else (a – b);
```

- Utiliza una macro optimizada que gasta una slice para cada dos bits de la operación

```
Number of Slices:              4  out of    768     0%
Number of 4 input LUTs:        8  out of   1536     0%
```

- Pero si se desactiva la opción '*Resource Sharing*', cada operación se realiza por separado, incrementándose notablemente el área utilizada:

```
Number of Slices:             13  out of    768     1%
Number of 4 input LUTs:       24  out of   1536     1%
```

Aritmética: Sumadores/Restadores

- Ejemplo 1 (addsub.vhd)
  - Implementar utilizando la opción “resource sharing” y sin ella.
  - Ver resultados en el synthesis report y RTL schematic.

Aritmética: Operadores con y sin signo

- Para escoger que tipo de aritmética se empleará, existen varias posibilidades:
  - Utilizar la librería adecuada

```
use IEEE.std_logic_unsigned.all;
...
p <= a * b;
```

  - Utilizar señales del tipo signed o unsigned

```
signal a,b : signed(7 downto 0);
...
p <= a * b;
```

  - Utilizar std_logic_vector, y hacer un *casting* a signed o unsigned. Esto es posible porque son "*closely related types*"

```
use IEEE.std_logic_arith.all;
...
p <= unsigned(a) * unsigned(b);
```

Multiplicadores

- Al inferir multiplicadores, hay que tener en cuenta las características de los multiplicadores embebidos.
  - 17x17 bits sin signo
  - 18x18 bits con signo

```
use IEEE.STD_LOGIC_UNSIGNED.all;

entity multiplier is
   port ( a : in std_logic_vector(17 downto 0);
         b : in std_logic_vector(17 downto 0);
         p : out std_logic_vector(35 downto 0));
end multiplier;

architecture behavioral of multiplier is
begin
              p <= a * b;
end behavioral;
```

Multiplicadores: Resultados

- El multiplicador anterior es sin signo y de 18 bits:

```
Selected Device : 3s50pq208-5
Number of Slices:                 19  out of   768    2%
Number of 4 input LUTs:           36  out of  1536    2%
Number of bonded IOBs:            72  out of   124   58%
Number of MULT18X18s:              3  out of     4   75%
```

- Mejor si se convierte a multiplicador con signo:

```
p <= signed(a) * signed(b);
```

```
Selected Device : 3s50pq208-5
Number of bonded IOBs:            72  out of   124   58%
Number of MULT18X18s:              1  out of     4   25%
```

Multiplicadores: Resultados

- Ejemplo 2 (mult.vhd)
  - Implementar el multiplicador con y sin signo
  - Utilice LUTs en vez de bloques multiplicadores

Aritmética: Acumuladores

- Un acumulador es un circuito ligeramente distinto a un contador
  - El contador es del tipo C <= C ± 1
  - El acumulador es del tipo A <= A ± B, donde B puede ser una constante, o una señal (o variable)
- XST reconoce los acumuladores hacia arriba, abajo o mixtos (incluso cuando los valores hacia arriba y abajo son distintos)

```
process (C, CLR)
begin
        if (CLR='1') then
                tmp <= "0000";
        elsif (C'event and C='1') then
                tmp <= tmp + D;
        end if;
end process;
```

Máquinas de estados

- XST puede inferir una FSM a partir de su descripción
  - Para ello es imprescindible que tenga reset, bien sea síncrono o asíncrono
  - En el informe de la síntesis se puede ver si la FSM se ha inferido correctamente

Found finite state machine <FSM_0> for signal <state>.

States	4
Transitions	5
Inputs	1
Outputs	1
Reset type	asynchronous
Encoding	automatic
State register	d flip-flops

- La extracción de máquinas de estados se puede desactivar: hay que poner la opcíon *"FSM Encoding Algorithm*" a "*None*"

Extracción de multiplexores

- XST dispone de macros optimizadas para multiplexores
  - Usando los multiplexores de la cadena de acarreo, MUXCY
  - Empleando los F5MUX de las slices. Esta es la que proporciona los mejores resultados, puede mapear un MUX8:1 en un 2 slices con muy buen retardo
- Por ejemplo con este multiplexor de 8 a 1

```
O <= I(CONV_INTEGER(SEL));
```

- Los resultados son

```
# Multiplexers                 : 1
#      1-bit 8-to-1 multiplexer   : 1
...
Device utilization summary:
-------------------------
Selected Device : 3s50pq208-5
Number of Slices:                2  out of    768     0%
Number of 4 input LUTs:          4  out of   1536     0%
```

Multiplexores: Resultados

- Ejemplo 3 (mux8_1.vhd)
  - Implementar el multiplexor
  - Ver reporte de síntesis
  - FPGA Editor

Codificadores con prioridad: Ejemplo

- El siguiente trozo de código modela un típico codificador con prioridad:

```
code <= "000" when sel(0) = '1' else
        "001" when sel(1) = '1' else
        "010" when sel(2) = '1' else
        "011" when sel(3) = '1' else
        "100" when sel(4) = '1' else
        "101" when sel(5) = '1' else
        "110" when sel(6) = '1' else
        "111" when sel(7) = '1' else
        "000";
```

- La opción "*Priority Encoder Extraction"* fija si extraerán codificadores con prioridad

```
Macro Statistics :
# Priority Encoders                  : 1
#     3-bit 1-of-9 priority encoder: 1
```

Registros de desplazamiento: SRL16

- XST es capaz de inferir registros de desplazamiento, y los implementa usando las primitivas SRL16 de Virtex

```
process (CLK)
begin
        if rising_edge(CLK) then
                tmp <= tmp(6 downto 0) & SI;
        end if;
end process;
SO <= tmp(7);
```

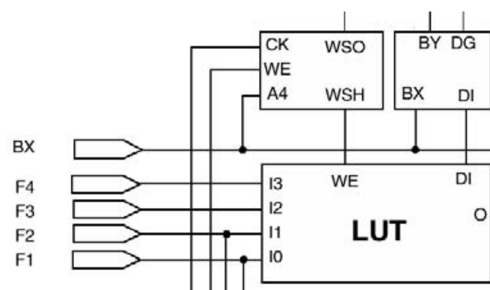
- El resultado es muy compacto

```
Number of Slices:               1  out of   768    0%
Number of Slice Flip Flops:     1  out of  1536    0%
Number of 4 input LUTs:         1  out of  1536    0%
```

- XST implementa el último bit del desplazador en el flip-flop del slice, para mejorar el clock-to-output (x3 típicamente)

Limitaciones de SRL16

- El componente SRL16 se implementa en la LUT, y tiene una funcionalidad muy limitada



- No puede implementar ni resets, ni carga paralela… sólo permite como mucho un clock enable

Registros de desplazamiento: Sin SRL16

- Si se añade un reset asíncrono

```
process (clk,rst)
begin
        if rst='1' then
                tmp <= (others=>'0');
        elsif rising_edge(clk) then
                tmp <= tmp(6 downto 0) & SI;
        end if;
end process;
SO <= tmp(7);
```

- El área se incrementa muy apreciablemente

```
Number of Slices:              5  out of    768     0%
Number of Slice Flip Flops:      8  out of   1536     0%
```

- Conclusión: para optimizar los resultados, hay que escribir el código VHDL teniendo en cuenta la arquitectura de la FPGA

Registros de desplazamiento: Solución

- Una solución (parcial) es que el bitstream puede configurarse para que el registro se inicie con unos ciertos contenidos
- Se hace mediante el atributo INIT, sólo vale para la instanciación explícita del SRL16

```
attribute INIT : string;
attribute INIT of shifter : label is "FFFF";

begin

shifter: SRL16 port map(
        CLK => CLK,
        D => SI,
        Q => SO,
        A0 => '1',
        A1 => '1',
        A2 => '1',
        A3 => '0');
```

Decodificadores

- XST interpreta los decodificadores como los circuitos que pasan de binario a one-hot (o one-cold)

```
res <= "00000001" when sel = "000" else
       "00000010" when sel = "001" else
       "00000100" when sel = "010" else
       "00001000" when sel = "011" else
       "00010000" when sel = "100" else
       "00100000" when sel = "101" else
       "01000000" when sel = "110" else
       "10000000";
```

- En el log se comprueba que se ha extraído el decodificador:

```
Macro Statistics
# Decoders : 1
1-of-8 decoder : 1
```

- XST no infiere un decodificador si no se utiliza alguna de las posibles entradas, salvo que sean consecutivas y al del final del espacio de códigos

Desplazadores lógicos

- XST dispone también de macros optimizadas para las operaciones de desplazamiento (lógico o aritmético) y rotación
- Las infiere tanto si se utilizan los operadores de VHDL (sll, srl, rol, ror) como si se usa la concatenación
- Funcionan descripciones de este tipo:

```
use IEEE.numeric_std.all
...
with SEL select
          SO <= DI when "00",
                DI sll 1 when "01",
                DI sll 2 when "10",
                DI sll 3 when others;
```

- Desplazamientos del mismo tipos consecutivos. También funciona esta descripción más compacta:

```
SO <= DI sll TO_INTEGER(SEL);
```

Eliminación de registros equivalentes

- En este contador – muy mal resuelto – los registros q y count son similares:

```
process(clk)
        variable temp : std_logic_vector(7 downto 0);
begin
        if rising_edge(clk) then
                temp := count + 1;
                q <= temp;
                count <= temp;
        end if;
end process;
```

- Y por tanto uno de ellos es eliminado

```
Register <count> equivalent to <q> has been removed
Found 8-bit register for signal <q>.
```

- Hay veces que el problema no es tan evidente

```
attribute enum_encoding : string;
attribute enum_encoding of state_type : type is "0000 1111 1010 0101";
```

Duplicación de registros: Ejemplo

- En este circuito la señal reg_input tiene un fanout enorme, y es por tanto una buena candidata para duplicación de registros

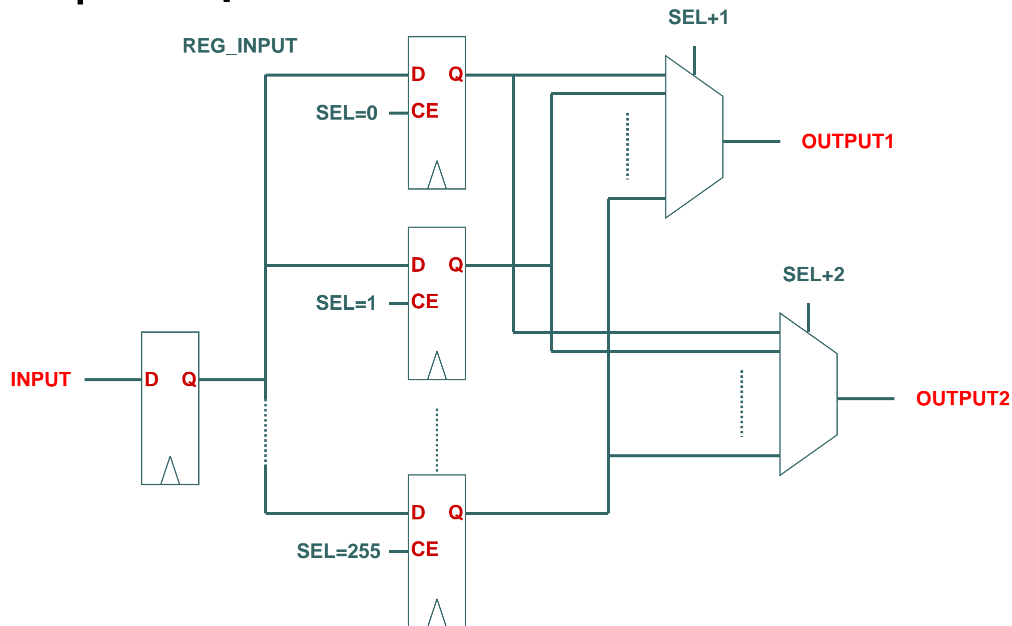
```
type data_array_t is array (255 downto 0) of std_logic_vector(7 downto 0);
signal data_array : data_array_t;

begin

        process(clk)
        begin
                if rising_edge(clk) then
                        reg_input <= input;
                        data_array(CONV_INTEGER(sel)) <= reg_input;
                end if;
        end process;

        output1 <= data_array(CONV_INTEGER(sel + 1));
        output2 <= data_array(CONV_INTEGER(sel + 2));
```

Duplicación de registros: Esquema



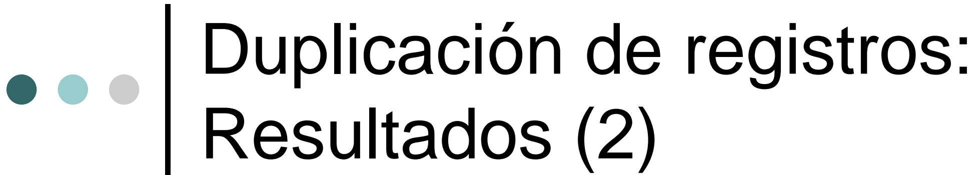
Duplicación de registros: Resultados (1)

- En efecto, el registro se ha duplicado

  FlipFlop reg_input_7 has been replicated 1 time(s)
  FlipFlop reg_input_0 has been replicated 1 time(s)
  ...
  FlipFlop reg_input_5 has been replicated 1 time(s)
  FlipFlop reg_input_6 has been replicated 1 time(s)

- Aunque el fanout ha resultado ser mucho menor de lo esperado, porque el circuito se ha implementado de una manera muy eficiente con dos memorias *distributed RAM*

```
Macro Statistics :
# RAM                              : 2
#      256x8-bit dual-port distributed RAM: 2
# Registers                        : 1
#      8-bit register              : 1
# Adders/Subtractors               : 2
#      7-bit adder                 : 1
#      8-bit adder                 : 1
```



Duplicación de registros: Resultados (2)

- El camino crítico con duplicación del registro queda así

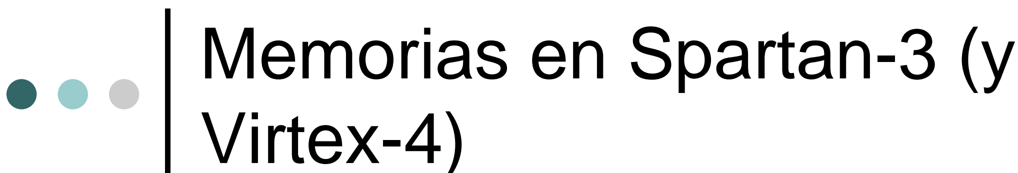
```
FD:C->Q          16   0.626   0.995  reg_input_4_1 (reg_input_4_1)
RAM16X1D:D             0.401         Mram_data_array_ren_inst_ramx_207
    ----------------------------------------
    Total                   2.022ns (1.027ns logic, 0.995ns route)
```

- Y al desactivar esta opción (o pasar a optimización en área)

```
FD:C->Q          32   0.626   1.321  reg_input_4 (reg_input_4)    RAM16X1D:D            0.401
Mram_data_array_inst_ramx_65
    ----------------------------------------
    Total                   2.348ns (1.027ns logic, 1.321ns route)
```

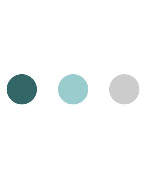
- Y si además se desactiva la extracción de RAM

```
FD:C->Q         256   0.626   1.409  reg_input_7 (reg_input_7)    FDE:D                 0.176
data_array_254_7
    ----------------------------------------
    Total                   2.211ns (0.802ns logic, 1.409ns route)
```



Memorias en Spartan-3 (y Virtex-4)

- Distributed RAM (y ROM)
  - Construidas con las LUTs de los CLBs
  - 16x1 de doble puerto, uno de lectura/escritura y otro de lectura
  - 64x1, 32x1 y 16x1 de simple puerto
  - Lectura asíncrona, escritura síncrona
- BlockRAM
  - Bloques separados de la matriz de CLBs
  - 18Kbits, configurables desde 16384x1, hasta 512x32. También son posibles configuraciones desde 2048x9 hasta 512x36 (paridad)
  - Doble puerto (ambos de lectura/escritura)
  - Lectura y escritura síncronas
  - La temporización entre DI y DO puede ser lectura primero, escritura primero o sin cambios



Distributed RAM: Escritura Síncrona, Lectura Asíncrona

- Las memorias son matrices bidimensionales de datos

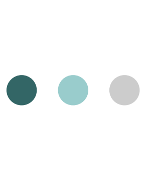
```
type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
signal RAM : ram_type;
```

- Al especificar escritura síncrona y lectura asíncrona se fuerza el uso de distributed RAM

```
process (clk)
begin
        if (clk'event and clk = '1') then
                if (we = '1') then
                        RAM(conv_integer(a)) <= di;
                end if;
        end if;
end process;
do <= RAM(conv_integer(a));
```

- Resultado:

```
Found 32x4-bit single-port distributed RAM for signal <ram>
```



Lectura síncrona

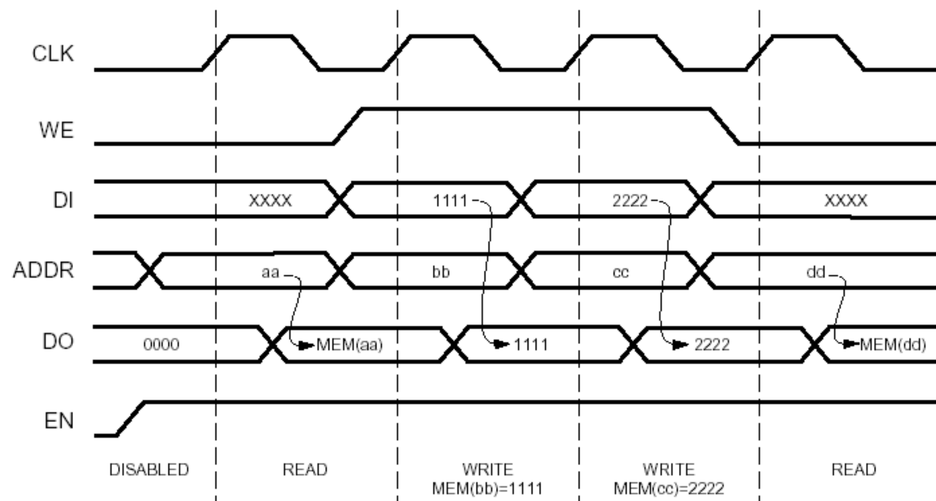
- Primera posibilidad:

```
process (clk)
begin
        if (clk'event and clk = '1') then
                if (we = '1') then
                        RAM(conv_integer(a)) <= di;
                end if;
                do <= RAM(conv_integer(a));
        end if;
end process;
```

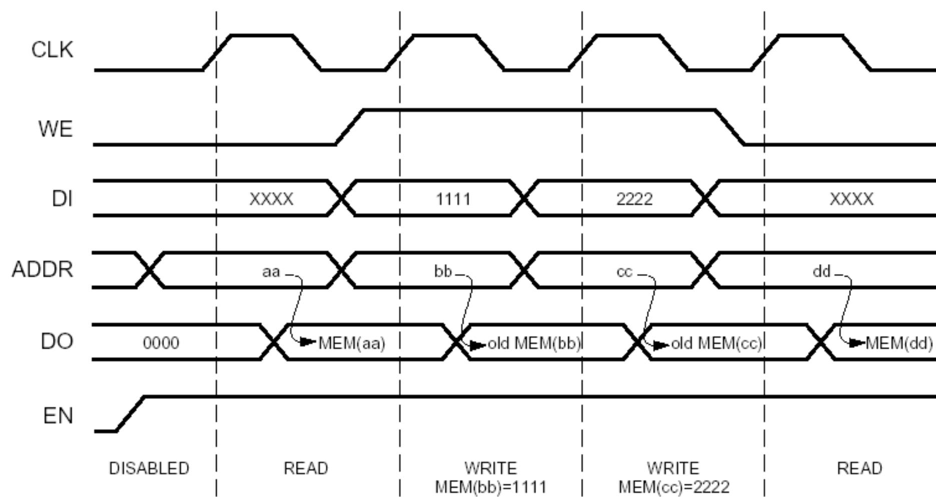
- Al examinar los resultados se comprueba que se ha generado una BlockRAM con lectura primero:

```
Found 32x4-bit single-port block RAM for signal <RAM>.
-----------------------------------------------------------------------
| mode           | read-first               |     |
| aspect ratio     | 32-word x 4-bit             |     |
```

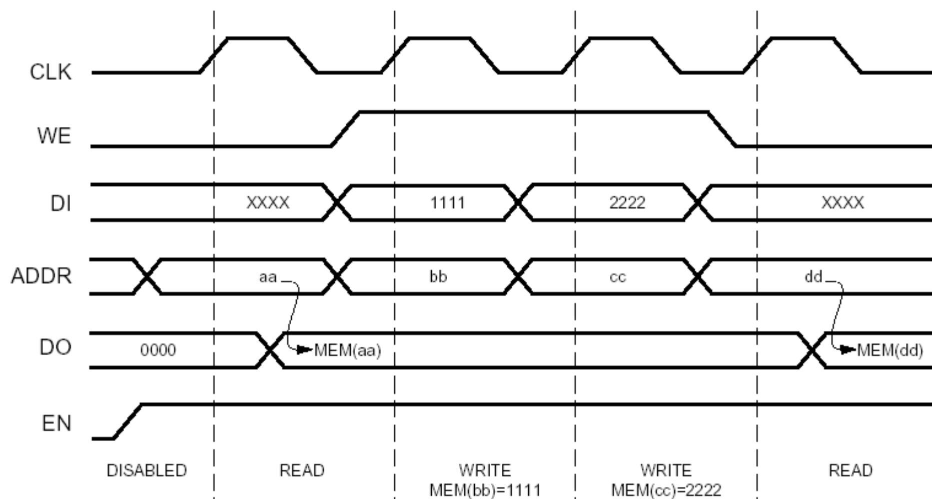
Modos de funcionamiento de la BlockRAM: Escritura primero (WRITE-FIRST)



Modos de funcionamiento de la BlockRAM: Lectura primero (READ-FIRST)



Modos de funcionamiento de la BlockRAM: Modo sin cambio (NO-CHANGE)



Lectura síncrona con escritura primero

- Probando ahora esta alternativa

```
process (clk)
begin
          if (clk'event and clk = '1') then
                    if (we = '1') then
                              RAM(conv_integer(a)) <= di;
                              do <= di;
                    else
                              do <= RAM(conv_integer(a));
                    end if;
          end if;
end process;
```

- Ahora la escritura tiene prioridad en la BlockRAM que se ha inferido:

```
Found 32x4-bit single-port block RAM for signal <RAM>.
-----------------------------------------------------------------------
| mode            | write-first                    |        |
| aspect ratio      | 32-word x 4-bit                  |        |
```

Lectura síncrona con escritura primero (2)

- Otra alternativa es considerar que lo que se está registrando es la dirección de lectura:

```
type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
signal RAM : ram_type;
signal read_a : std_logic_vector(4 downto 0);

begin

        process (clk)
        begin
                if (clk'event and clk = '1') then
                        if (we = '1') then
                                RAM(conv_integer(a)) <= di;
                        end if;
                        read_a <= a;
                end if;
        end process;

        do <= RAM(conv_integer(read_a));
```

Lectura síncrona con modo sin cambio

- Para el modo sin cambio se puede usar esta alternativa

```
process (clk)
begin
        if (clk'event and clk = '1') then
                if (we = '1') then
                        RAM(conv_integer(a)) <= di;
                else
                        do <= RAM(conv_integer(a));
                end if;
        end if;
end process;
```

- En efecto:

```
Found 32x4-bit single-port block RAM for signal <RAM>.
-----------------------------------------------------------------------    | mode          | no-change
|      |   | aspect ratio      | 32-word x 4-bit                |      |
```

Evitar la inferencia de BlockRAM

- La restricción (*constraint*) ram_style permite especificar el tipo de memoria que se inferirá.
- Se puede especificar globalmente, o como un atributo de la señal que modela la matriz de la memoria

```
type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
signal RAM : ram_type;

attribute ram_style : string;
attribute ram_style of RAM : signal is "distributed";
```

- De esta manera se evita la inferencia de BlockRAM en el ejemplo anterior:

```
Found 32x4-bit single-port block RAM for signal <ram>.
...
Cell Usage
# RAMS                  : 8
#     RAM16X1D          : 8
```

ROMs

- Se pueden inferir de un *if* o un *case*:

```
process (a)
begin
        case a is
                when "0000" => do <= "0101";
                when "0001" => do <= "1010";
                when "0010" => do <= "0000";
                ...
                when "1110" => do <= "0101";
                when "1111" => do <= "1111";
                when others => do <= "XXXX";
        end case;
end process;
```

- El resultado es:

```
Macro Statistics
# ROMs                  : 1
  16x4-bit ROM          : 1
```

ROMs: Alternativa más sencilla

- Una alternativa más compacta es modelar la ROM como una constante bidimensional:

```
architecture Behavioral of rominfr2 is

type ROM_TYPE is array(15 downto 0) of std_logic_vector(3 downto 0);
constant ROM : rom_type := (x"1", x"A", x"0", x"3",
                            x"5", x"E", x"0", x"F",
                            x"0", x"2", x"7", x"8",
                            x"C", x"3", x"1", x"0");

begin

        do <= ROM(conv_integer(a));

end Behavioral;
```

En el fondo este modelo es el que más se aproxima a la implementación física de la memoria ROM

Doble puerto

- La inferencia de la memoria de doble puerto sigue el mismo esquema que los ejemplos anteriores:

```
process (clk)
begin
        if (clk'event and clk = '1') then
                if (we = '1') then
                        RAM(conv_integer(a)) <= di;
                end if;
                read_a <= a;
                read_dpra <= dpra;
        end if;
end process;

spo <= RAM(conv_integer(read_a));
dpo <= RAM(conv_integer(read_dpra));
```

- Hay que tener en cuenta las mismas limitaciones que antes con respecto al tipo de memoria que se quiere inferir (lectura asíncrona, prioridad de la lectura...)

Instanciando memorias

- Las memorias, a parte de inferirse, se pueden instanciar:

```
library unisim;
use unisim.vcomponents.all;
...
ram_256_x_16: RAMB4_S16
port map(    DI => "0000000000000000",
             EN => '1',
             WE => '0',
             RST => '0',
             CLK => clk,
             ADDR => address,
             DO => instruction(15 downto 0));
```

- La declaración del componente se puede obtener del paquete **vcomponents** de la librería **unisim**
- Tema importante: Inicio de los contenidos de la memoria, tanto para la FPGA como simulación

Diseño para Síntesis

Sergio López Buedo, Elías Todorovich
Septiembre 2012
etodorov@exa.unicen.edu.ar

INSTITUTO DE INVESTIGACIÓN EN TECNOLOGÍA INFORMÁTICA AVANZADA

Universidad Nacional del Centro de la Provincia de Buenos Aires