

# Tema 4: VHDL para procesamiento de señal

Hipólito Guzmán Miranda  
Departamento de Ingeniería Electrónica  
Universidad de Sevilla  
[hguzman@us.es](mailto:hguzman@us.es)

## Contexto docente

### BT02: Lenguajes de descripción hardware

- Tema 3: VHDL avanzado
- Tema 4: VHDL para procesamiento de señal
- Tema 5: Verilog para diseñadores de VHDL

### Conocimientos previos requeridos:

- VHDL básico (Grado / Complementos de Electrónica)
- VHDL avanzado
- Conceptos básicos de procesamiento de señal

## Objetivos de aprendizaje

- Conocer las capacidades del lenguaje VHDL para procesamiento de señal
- Evitar fallos comunes a la hora de determinar los tipos de datos a utilizar en un diseño digital de procesamiento de señal
- Determinar cuándo es necesario utilizar datos de tipo flotante y cuándo no
- Conocer las diferencias de comportamiento entre síntesis y simulación de algunas construcciones típicas del lenguaje

# Repaso

VHDL avanzado:

- Otra forma de estructurar nuestro código
- Atención a la ocupación de recursos en síntesis

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## ¿Para qué se usan?

Principalmente para índices que sirvan para acceder a arrays/agregados:

```
signal i : integer range 0 to 7;
```

```
signal vect : std_logic_vector (7 downto 0);
```

```
vect(i) <= '1';
```

```
std_logic_vector ( integer ) = std_logic
```



## Definid el rango!

```
signal i : integer;           -- 32 bits  
signal j : integer range 0 to 255;  -- 8 bits  
signal k : integer range -128 to 127; -- 8 bits
```

Puede dar “simulation mismatch” si os salís del rango:

- en simulación se comportará como si tuviera 32 bits (al menos en Xilinx ISIM)
- en implementación no!

## Ejemplo

```
signal cont : integer range 0 to 7;
```

```
begin
```

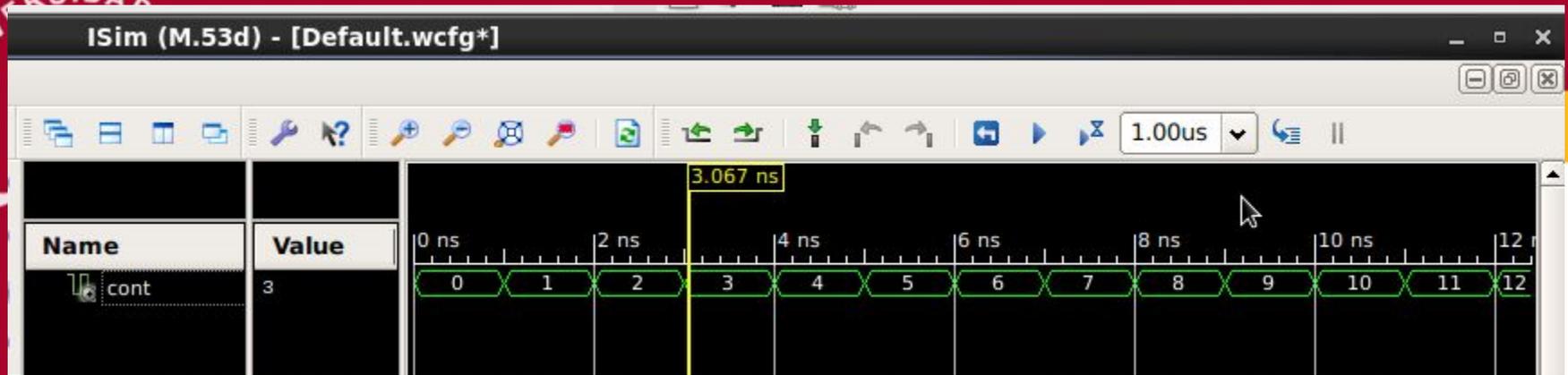
```
process
```

```
begin
```

```
    wait for 1 ns;
```

```
    cont <= cont +1;
```

```
end process;
```



```
begin
```

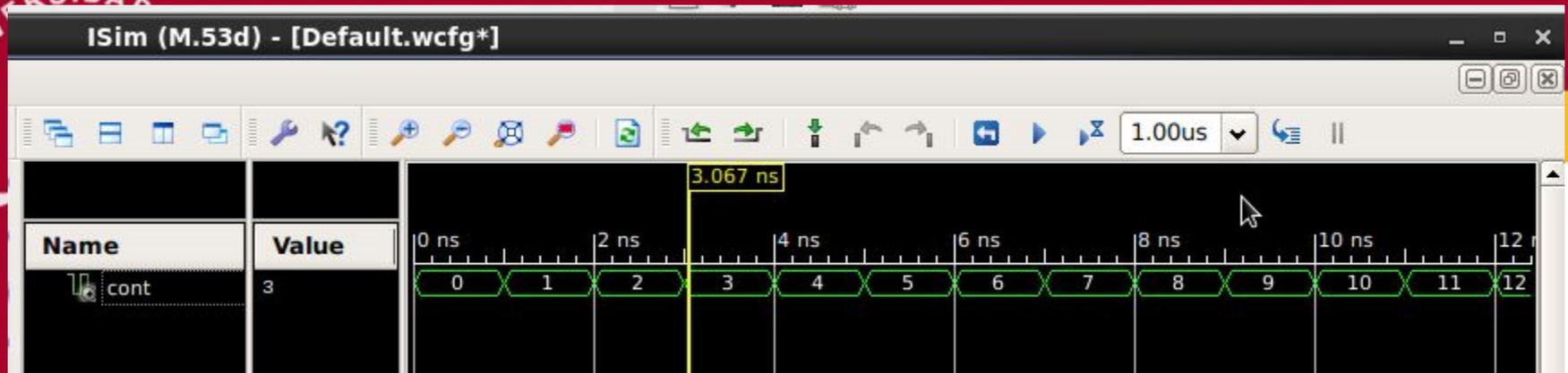
```
process
```

```
begin
```

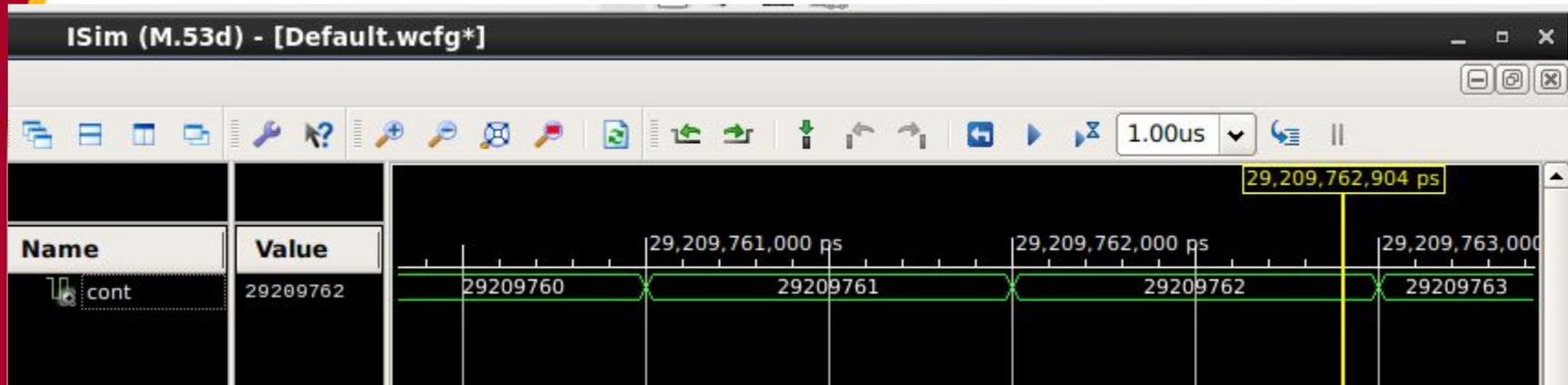
```
    wait for 1 ns;
```

```
    cont <= cont +1;
```

```
end process;
```



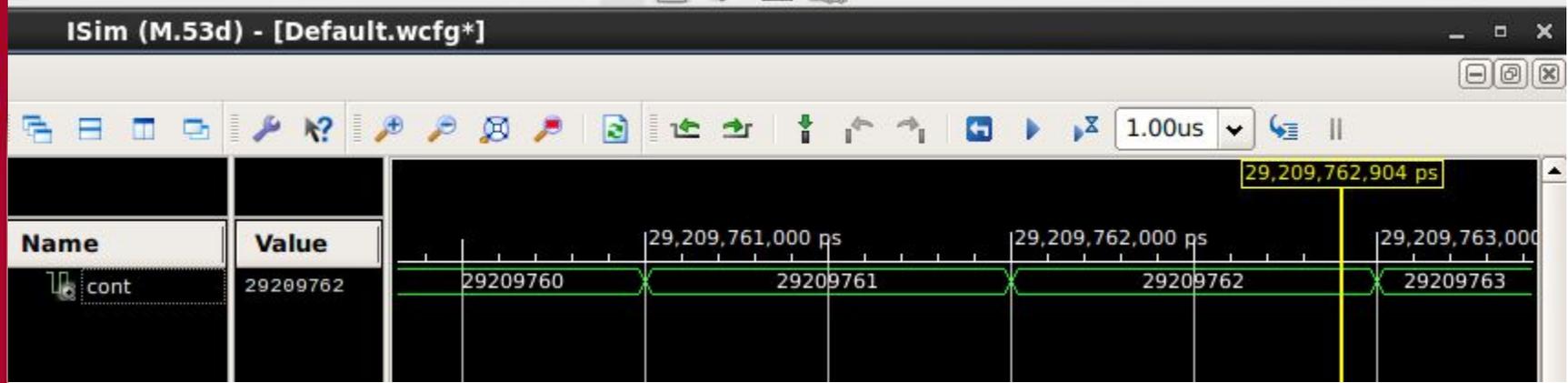
Signal cont : Integer Range (0 to 12)



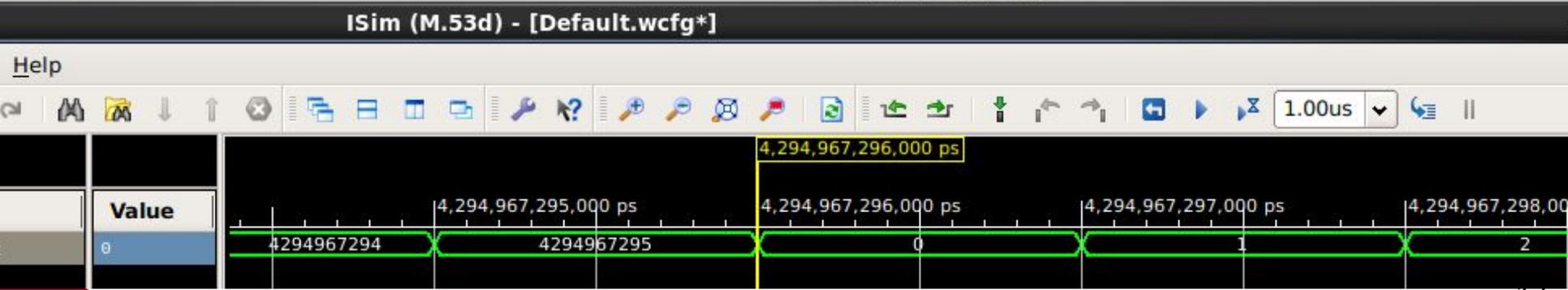
```
begin  
    wait for 1 ns;  
    cont <= cont +1;  
end process;
```



Signal cont: Integer Range (0 to 12)



hexin



## Con GHDL esto no ocurre

```
ghdl -a testbench.vhd
ghdl -e testbench
./testbench --vcd=testbench.vcd --wave=testbench.ghw
testbench.vhd:14:9:@1ns: (report note): count = 0
testbench.vhd:14:9:@2ns: (report note): count = 1
testbench.vhd:14:9:@3ns: (report note): count = 2
testbench.vhd:14:9:@4ns: (report note): count = 3
testbench.vhd:14:9:@5ns: (report note): count = 4
testbench.vhd:14:9:@6ns: (report note): count = 5
testbench.vhd:14:9:@7ns: (report note): count = 6
testbench.vhd:14:9:@8ns: (report note): count = 7
./testbench:error: bound check failure at testbench.vhd:17
./testbench:error: simulation failed
```

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Realmente, es un tipo enumerado

-- predefinido en VHDL:

```
type Boolean is (false, true);
```

Puede ser interesante en GENERICS, para parametrizar componentes

Para señales binarias de I/O, usad  
std\_ulogic/std\_logic

## Ejemplo

```
if (my_std_logic = '1') then
```

VS

```
if ( condition ) then
```

La expresión dentro del if ha de ser un boolean

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## **Sólo cuando realmente se necesite**

El float es un tradeoff entre precisión y rango

(es el 'google maps' de los números :)

Tiene sus limitaciones:

- Pérdida de precisión
- Ocupación de recursos
- Operaciones más lentas
- Necesidad de convertir desde/a otros tipos

## Pérdida de precisión

- ¿Qué pasa si sumamos mil números flotantes?

Veamos un ejemplo (en C)

```
#include <stdio.h> // for printf
#include <time.h> // for time
#include <stdlib.h> // for rand

#define MAX 10e10 // max rand number range

int main (void)
{
float table [1000];
float accfwd = 0;
float accrev = 0;

srand(time(NULL)); // set random seed to time

int i;

// Initialize table with random floats
for (i=0; i<1000; i++)
{
table[i] = ((float)rand()/((float)(RAND_MAX))) * MAX;
}
```

```
// Sum from 0 to 999
for (i=0; i<1000; i++)
{
    accfwd += table[i];
}

// Sum from 999 to 0
for (i=999; i>= 0; i--)
{
    accrev += table[i];
}

// Compare and print
printf ("sum (fwd): %f\n", accfwd);
printf ("sum (rev): %f\n", accrev);
printf ("difference: %f\n", accfwd-accrev);

return 1;
}
```

## Pérdida de precisión

Al ejecutar el programa:

```
$ ./a.out
```

```
sum (fwd): 50039409868800.000000
```

```
sum (rev): 50039435034624.000000
```

```
difference: -25165824.000000
```

## Pérdida de precisión

Al ejecutar el programa:

```
$ ./a.out
```

```
sum (fwd): 50039409868800.000000
```

```
sum (rev): 50039435034624.000000
```

```
difference: -25165824.000000
```

## Pérdida de precisión

Estándar IEEE 754

`float` (32bit: 24 mantisa + 8 exponente)

- Rango  $\sim 1.75 \cdot 10^{-38}$  a  $\sim 3.40 \cdot 10^{38}$

`double` (64bit: 53 mantisa + 11 exponente)

- Rango:  $\sim 2.22 \cdot 10^{-308}$  a  $\sim 1.79 \cdot 10^{308}$

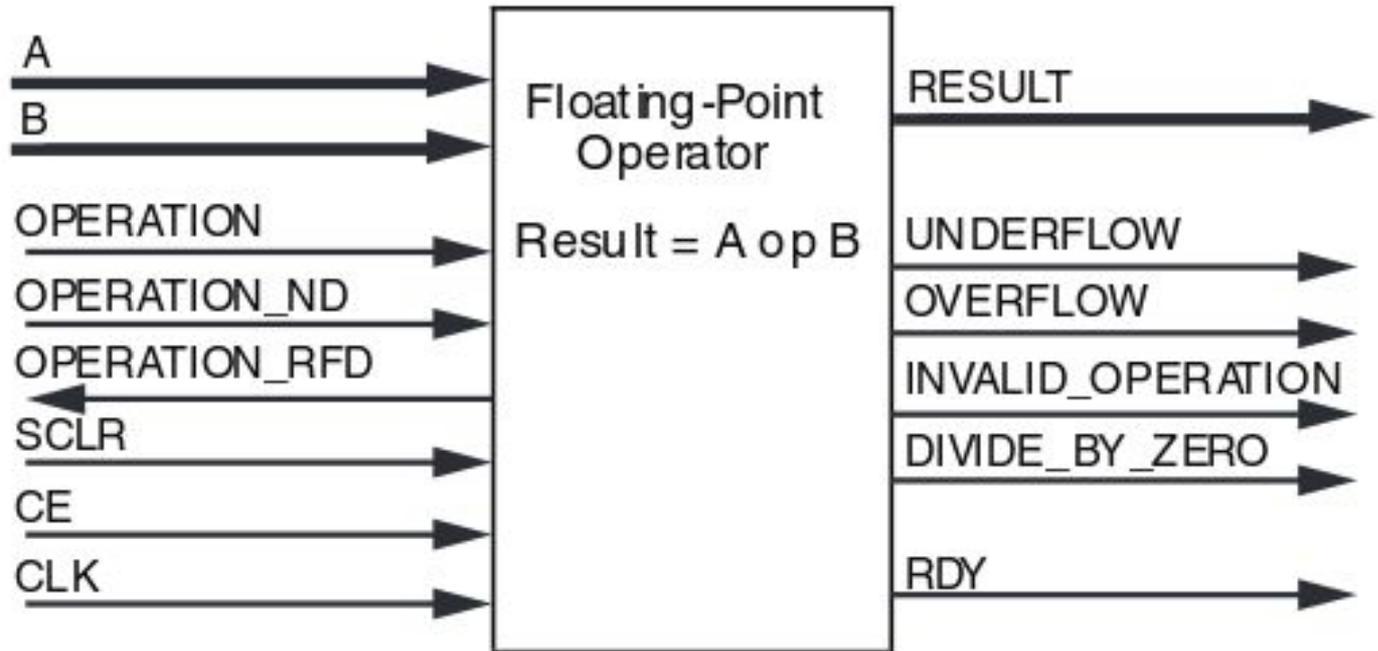
## ¿De verdad lo necesitas?

Con 32 bits (fixed) tienes hasta  $4\text{G} \sim 4 \cdot 10^9$   
¡nueve órdenes de magnitud!

Con 64 (fixed) bits tienes hasta  $1.89 \cdot 10^{19}$  !

En aplicaciones típicas, con eso suele ser  
más que suficiente

## Xilinx floating-point operator:



DS335\_01\_021508

Figure 1: Block Diagram of Generic Floating-Point Binary Operator Core

Fuente: Xilinx (DS335)

Cada fabricante proporciona sus bloques

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Varias formas:

Usando `std_logic_vector` + librerías  
`ieee.std_logic_[un]signed`

Usando librería `numeric_std` + tipos `signed`  
y `unsigned`

Usando librerías específicas (por ejemplo: el  
`package ieee_proposed.fixed_pkg`)

No soportadas por todos los fabricantes!

## Varias formas:

Usando `std_logic_vector` + librerías  
`ieee.std_logic_[un]signed`

Usando librería `numeric_std` + tipos  
`signed` y `unsigned`

Usando librerías específicas (por ejemplo: el  
`package ieee_proposed.fixed_pkg`)

No soportadas por todos los fabricantes!

## **Al final es lo mismo!**

Vector de bits  $\rightarrow$  nosotros decidimos qué interpretamos como parte entera y qué interpretamos como parte fraccional

La lógica para sumar o multiplicar es la misma independientemente de dónde interpretas que está la coma!

## Ejemplo (suma)

	4,0	4,1	4,2
0111	7	3.5	1.75
+ 0001	1	0.5	0.25
= 1000	8	4.0	2.00

Notación:  $(x, y) = (\text{bits\_totales}, \text{bits\_parte\_fraccional})$

Mismos bits de parte fraccional en el resultado  
que en los operandos

## Ejemplo (mult)

	4,0	4,1	4,2
	7	3.5	1.75
*	2	1	0.5
=	14	3.5	0.8750
	8,0	8,2	8,4

$$\text{bits\_ent}(\text{res}) = \text{bits\_ent}(a) + \text{bits\_ent}(b)$$

$$\text{bits\_frac}(\text{res}) = \text{bits\_frac}(a) + \text{bits\_frac}(b)$$

## Ejercicio (mult)

	“total, dec”	valor
	4,2	
*	4,1	
=		

¿Dónde tiene el punto el resultado?

## ¿Con qué bits nos quedamos?

- Cuando realizamos una operación en punto fijo podemos obtener más bits que los operandos
- Si nos descuidamos terminamos sacando 58 bits a la salida
- Hay que estudiar los rangos de los números con que estamos operando
- Y decidir en función de
  - Si estamos aprovechando el rango
  - La precisión que necesitemos

## Ejercicio (unsigned):

- a: entero entre 0 y 5 (3 bits)
- b: entero entre 0 y 20 (5 bits)

¿Cuántos bits necesitamos para  $a * b$ ?

## Ejercicio (signed):

- a: entero entre -5 y 5 (4 bits)
- b: entero entre -10 y 12 (5 bits)

¿Cuántos bits necesitamos para  $a * b$ ?

## Suma con acarreo

Si  $a$  y  $b$  son de 8 bits, para hacer

$c \leftarrow a + b$ ;

$c$  tiene que ser de 8 bits.

Para tener el bit de acarreo (si nos hace falta):

$d \leftarrow \text{resize}(a, 9) + \text{resize}(b, 9)$ ;

$d$  es vector de 9 bits

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Parte real + parte imaginaria

Existe una librería que define un tipo complejo, pero que no es sintetizable: tendréis que crearos los vuestros.

Por ejemplo:

```
type complex10 is record  
  re : signed (9 downto 0);  
  im : signed (9 downto 0);  
end record;
```

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

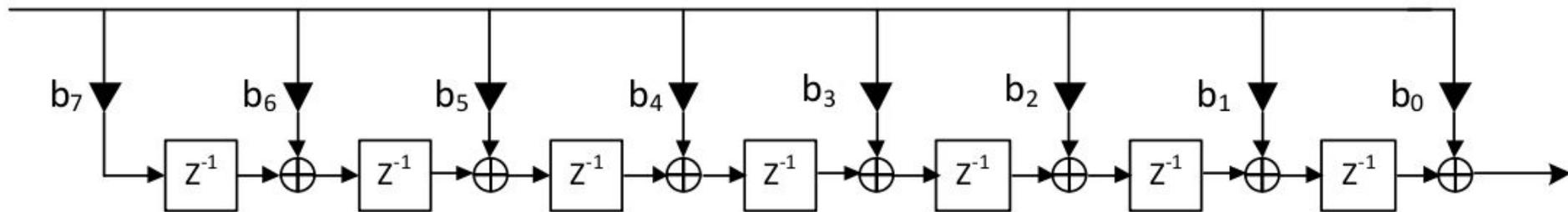
## ¿Cómo comunicamos nuestros bloques?

- No siempre (en cada ciclo de clk) las entradas tendrán un nuevo dato
- El uso de una señal “data\_valid” está altamente recomendado
- Otros interfaces: FIFOs, memorias

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Filtros como conjuntos de etapas (o 'taps')



- 1 etapa es una multiplicación, una suma, un retraso
- Retraso 1 muestra != retraso 1 ciclo
- Si usamos "data\_valid" es fácil de arreglar

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Almacenamiento de datos

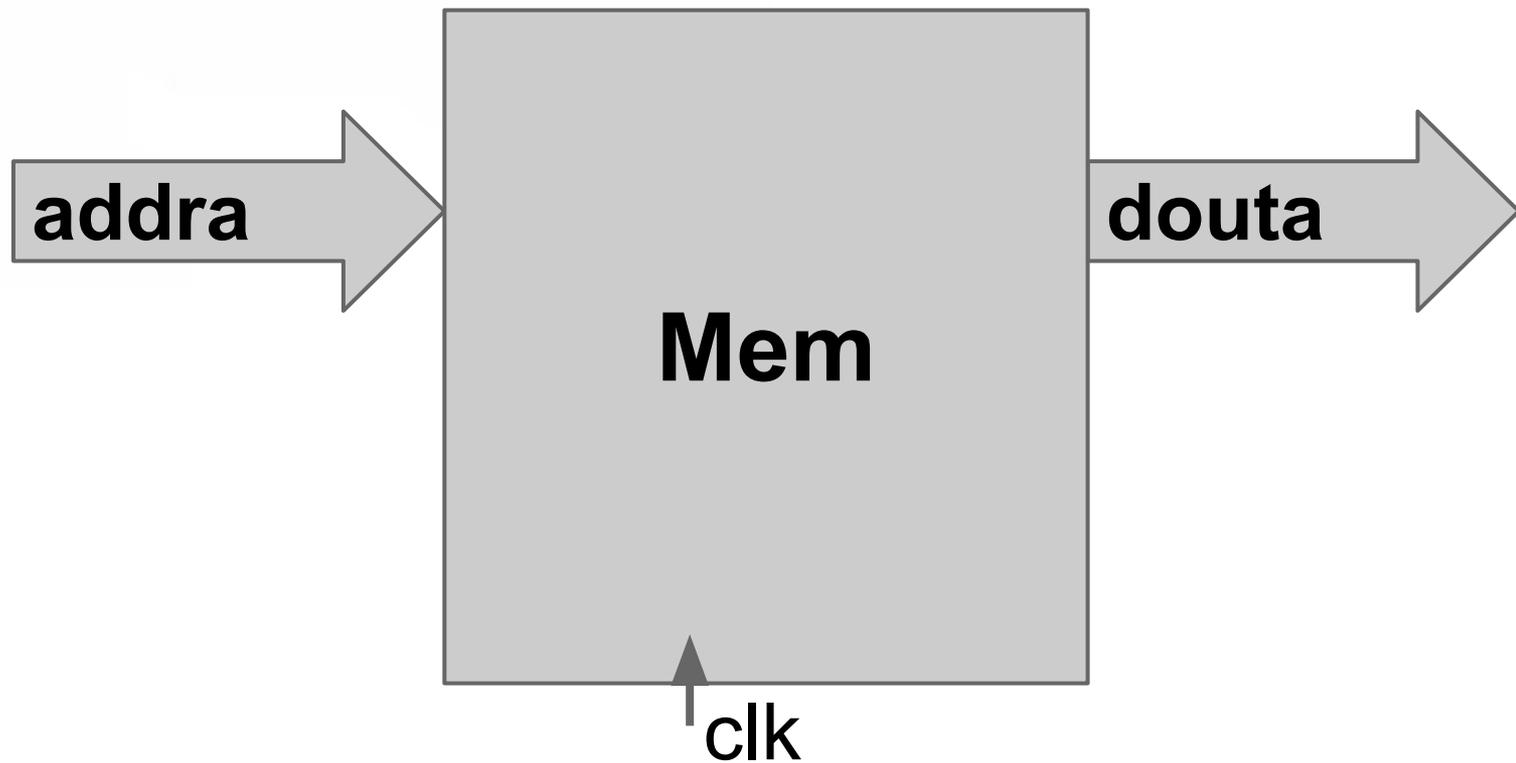
Se generan con el generador de IPs del fabricante (como Xilinx Core Generator) o usando VHDL que infiera memorias al sintetizarse

Utilizan BRAMs (Block RAMs) internas a la FPGA (no biestables)

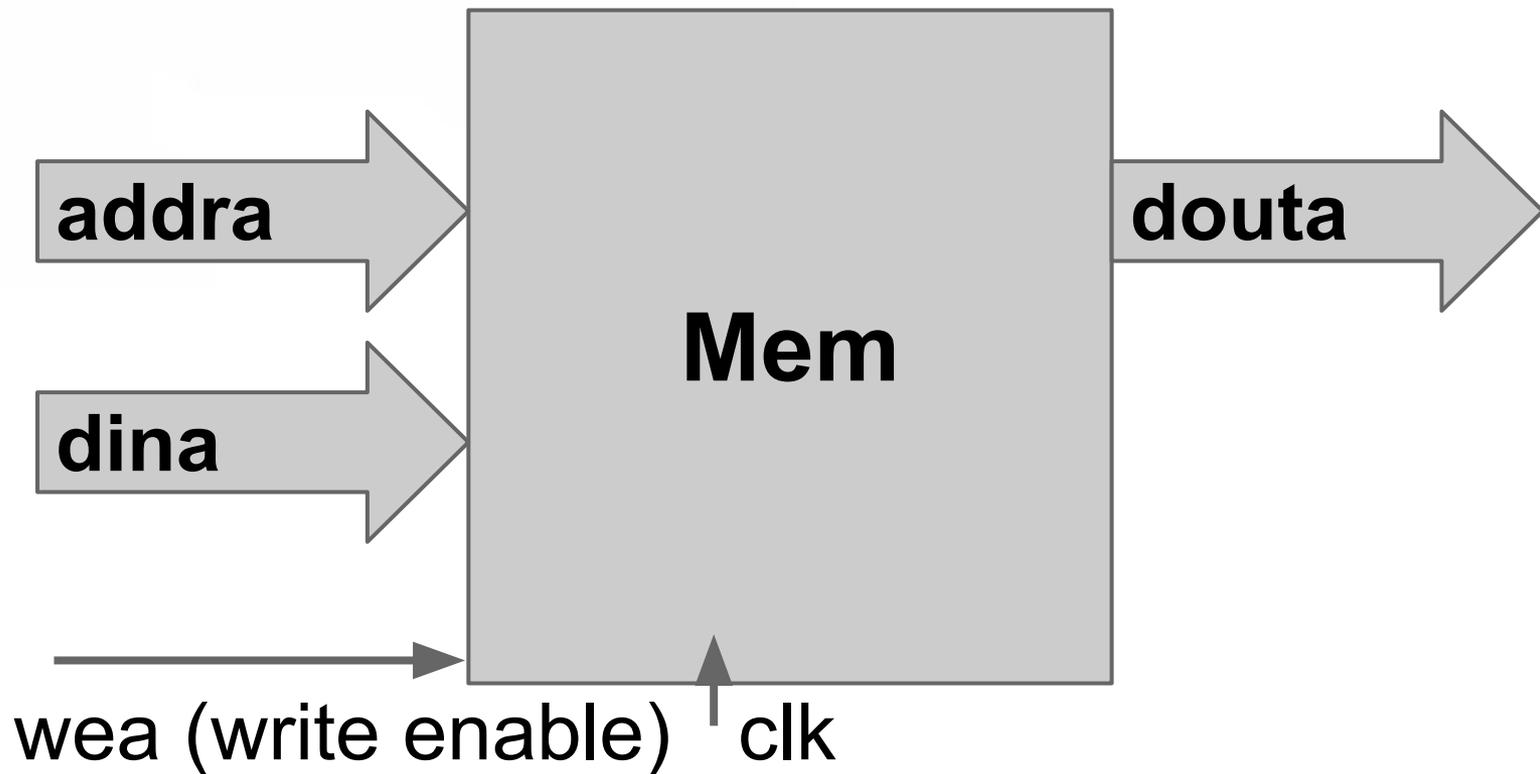
Síncronas (entrada de clk)

¿Habéis hecho alguna?

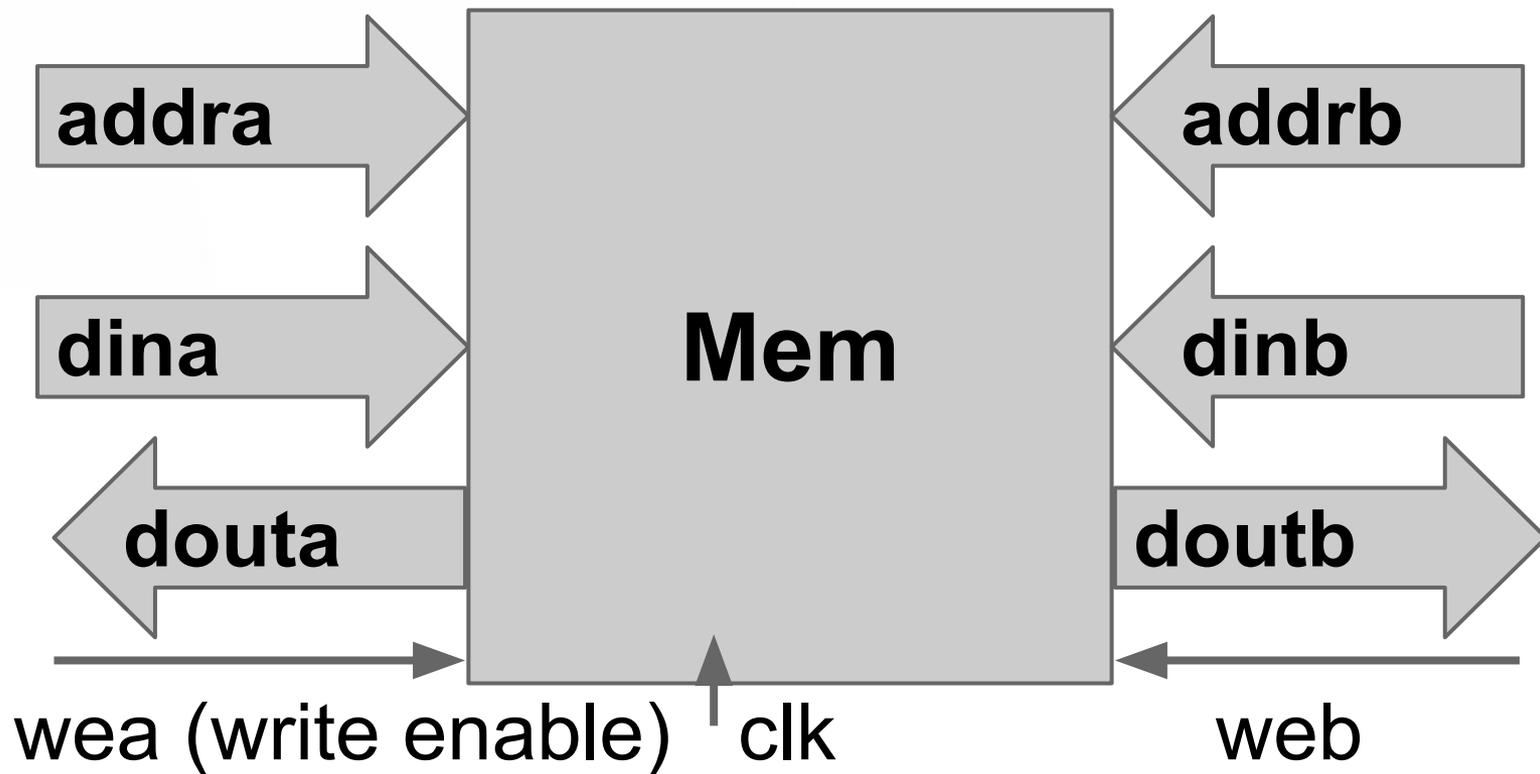
## Single port ROM



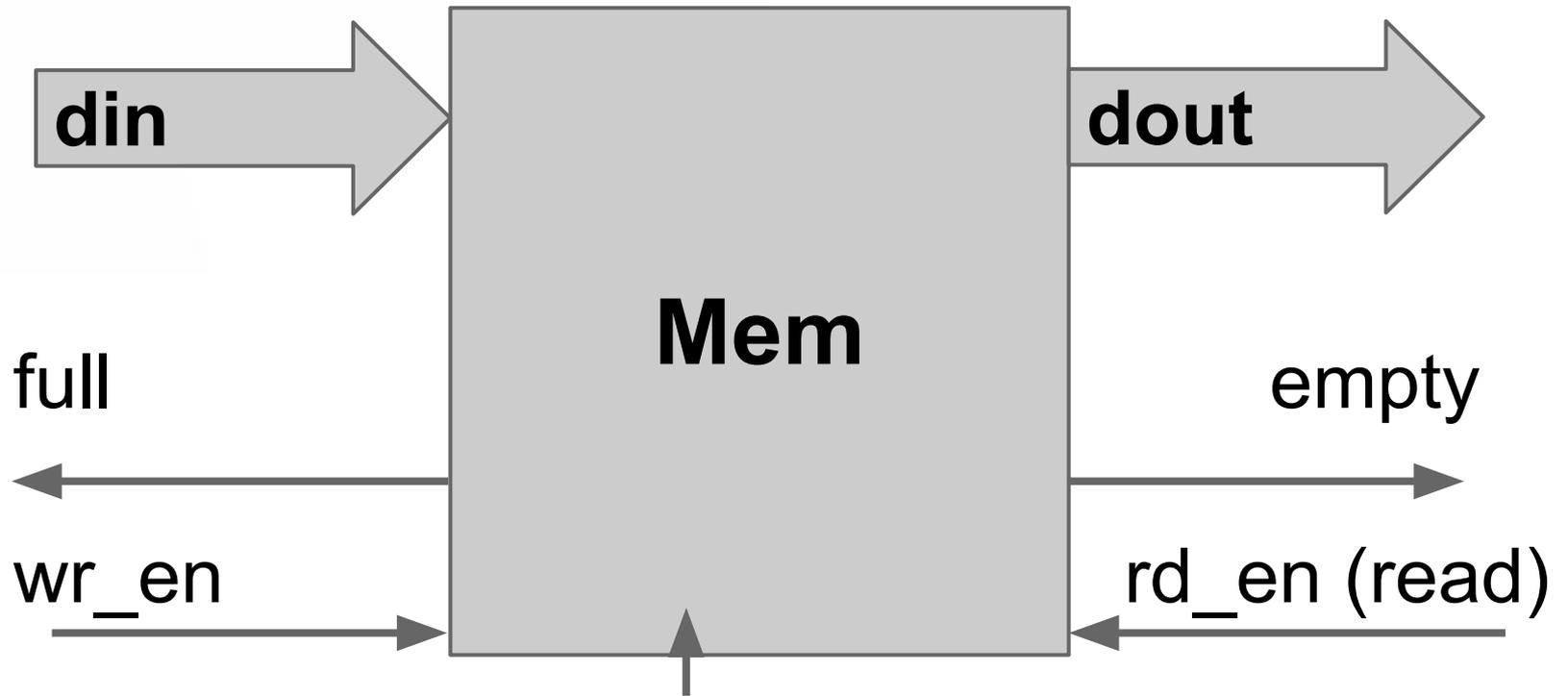
## Single port RAM



## Dual port RAM



## FIFOs



## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

# Conclusiones

## En procesamiento de señal:

- Para diseñar bien hay que conocer los tipos de datos y sus posibilidades
- No todos los ciclos de reloj vamos a tener un dato válido
- Memorias internas son útiles para almacenar datos que se procesan en bloque (ahorro de FFs)
- FIFOs y memorias de doble puerto también sirven de interfaz entre bloques

## Os recomiendo:

- Simulad cada concepto que no tengáis claro por separado
  - Siempre será mucho más sencillo que depurar un circuito complejo cuando lo tengáis hecho.

## Contenido

- Integer
- Boolean
- Sobre el punto flotante
- Punto fijo
- Tipos complejos
- Diseño de interfaces
- Filtros y etapas
- FIFOs y memorias
- Conclusiones
- Bibliografía

## Bibliografía

- Brian Mealy, Fabrizio Tappero, [Free Range VHDL](#). Free Range Factory, 2018
- *The VHDL Golden Reference Guide*. Doulos, 1995
- Ricardo Jasinski, *Effective Coding with VHDL: principles and best practice*. The MIT Press, 2016

## Resultados de aprendizaje

- Saber identificar los tipos de datos que intervienen en una expresión del tipo:  
`suma <= resize(a,9) + resize(b(7 downto 0),9);`
- Saber dimensionar un vector en punto fijo en función del rango de valores que se desea almacenar
- Comprender las ventajas y limitaciones del punto flotante
- Entender el paso de datos entre módulos utilizando señal de dato válido

# Dispositivos extra

## Ejercicio:

- a: entero entre 0 y 100
- b: entero entre 0 y 40

¿Cuántos bits necesitamos para  $a * b$ ?

## Ejercicio:

- a: entero entre -2 y 8
- b: entero entre -40 y 25

¿Cuántos bits necesitamos para  $a * b$ ?

## Ejercicio:

- a: entero entre -120 y 60
- b: entero entre -10 y 40

¿Cuántos bits necesitamos para  $a * b$ ?