

Evaluación 1 — Unidad 1

Sistemas Embebidos LA207 · lunes 7 de septiembre de 2026, 18:00

Teoría de la unidad + banco de ejercicios resuelto y explicado

Adam Del Pozo Ontiveros

Lee esto primero. El banco que te dieron está en **Verilog**, pero las clases 2, 3 y 4 fueron en **VHDL** («VHDL es el del curso», guía del 17-08). El aviso de la guía del 31-08 dice literalmente: «*La próxima sesión, lunes 7 de septiembre, es la Evaluación 1. Cubre la unidad 1 completa*».

Por eso esta guía te da **las dos versiones de cada cosa** y añade lo que el banco *no* trae pero la unidad 1 sí incluye: flip-flops, reset síncrono y asíncrono, registros y contadores. Si solo estudias el banco, te falta aproximadamente un tercio del examen.

Y es en papel. No hay simulador, ni autocompletado, ni compilador que te avise de un `endmodule` que falta. Eso cambia cómo hay que estudiar: no basta con *entender* el código, tienes que poder **escribirlo de memoria y verificarlo a mano**. Por eso todos los ejercicios de esta guía llevan una comprobación hecha a mano, sin simular, y la sección 7 es solo técnica de examen escrito.

1. Conocimiento previo (léelo aunque creas que ya lo sabes)

Si abres el banco y ves `module`, `always`, `entity` o `&` sin saber *qué estás describiendo*, el resto de la guía se siente como memorizar magia. Esta sección es el piso: vocabulario, idea de HDL y la diferencia VHDL/Verilog. No sustituye la teoría de la Unidad 1; la desbloquea.

1.1. ¿Qué es un HDL?

Un **HDL** (*Hardware Description Language*, lenguaje de descripción de hardware) **no es un lenguaje de programación** al estilo C o Python. No le dices a un procesador «haz esto paso a paso». Le describes a un compilador/síntesis **cómo está cableado un circuito**: qué señales entran, qué salen, y qué compuertas o registros las relacionan.

Software vs hardware (la frase que salva el examen).

- En software, una línea se ejecuta *después* de otra (salvo concurrencia explícita).
- En HDL, casi todo lo que escribes ocurre **en paralelo**, porque describe cables y bloques que existen al mismo tiempo en el silicio.

Por eso un **always** o un **process** no es un «bucle infinito»: es la forma de decir «este bloque reacciona cuando cambian estas señales».

1.2. ¿Qué significan VHDL y Verilog?

	VHDL	Verilog
Nombre	<i>VHSIC Hardware Description Language</i> (circuito integrado de muy alta velocidad)	<i>Verilog HDL</i> (IEEE 1364; hoy también SystemVerilog)
Origen	Años 80, defensa / estándar IEEE; muy usado en Europa y academia	Años 80 (Gateway/Cadence); muy usado en industria USA y ASICs
Estilo	Verboso, tipado fuerte, parece Ada/Pascal	Más compacto, parece C
En este curso	«VHDL es el del curso» (guía 17-08): clases 2-4	El banco del profesor está escrito en Verilog

No tienes que «elegir un bando». En la evaluación te pueden pedir cualquiera. Estudia **las dos plantillas** de cada idea. La tabla de traducción VHDL ↔ Verilog (más abajo y en la sección de chuleta) es oro.

1.3. Anatomía mínima: de qué está hecho un diseño

- **Señal / wire:** un cable. Lleva 0 o 1 (o un bus de varios bits).
- **Puerto (port):** señal que entra o sale del bloque (`input/output` en Verilog; `in/out` en VHDL).
- **Módulo / entidad:** la «cajita» del circuito. Verilog: `module ... endmodule`. VHDL: `entity ... end + architecture ... end`.
- **Compuerta:** AND, OR, NOT, XOR, NAND, NOR... operación booleana sobre bits.
- **Registro / flip-flop:** memoria de 1 bit que guarda un valor en el flanco del reloj. Sin reloj no hay «estado» secuencial.
- **Bus:** varios bits juntos, p.ej. `[3:0] = 4 bits` (Verilog) o `std_logic_vector(3 downto 0)` (VHDL).

1.4. Combinacional vs secuencial (en una frase cada uno)

Las dos familias de circuitos

- **Combinacional:** la salida depende *solo* de las entradas actuales. Ejemplos: sumador, MUX, decodificador, ALU sin registros. En código: `assign/with-select`, o `always @(*)` / `process(todas las entradas)` **sin** reloj.
- **Secuencial:** la salida también depende de un **estado** guardado (flip-flops). Ejemplos: contador, registro, FSM. En código: `always @(posedge clk)` / `process(clk)` con `rising_edge(clk)`.

El banco del profesor es casi todo combinacional. La Unidad 1 del curso también pide lo secuencial (sección 6 de esta guía).

1.5. Bits, niveles lógicos y notación

- Un bit vale **0** o **1** (en VHDL a menudo '0'/'1' entre comillas simples).
- **Activo en alto / en bajo:** «reset activo en alto» significa que el reset hace efecto cuando la señal es 1.
- Notación Verilog rápida: `1'b0` = un bit en 0; `4'b1010` = cuatro bits; `8'hFF` = ocho bits en hex.
- Negación: `~A` (Verilog) o `not A` / `A` con barra (VHDL / en papel $\bar{A}$).

1.6. Compuertas básicas (lo que dibujas o escribes en papel)

Compuerta	Símbolo / idea	Verilog	VHDL
NOT	invierte	<code>~A</code> o <code>not</code>	<code>not A</code>
AND	1 solo si todas 1	<code>A & B</code>	<code>A and B</code>
OR	1 si alguna 1	<code>A B</code>	<code>A or B</code>
XOR	1 si distintas	<code>A ^ B</code>	<code>A xor B</code>
NAND	NOT-AND	<code>~(A & B)</code>	<code>A nand B</code>
NOR	NOT-OR	<code>~(A B)</code>	<code>A nor B</code>

La trampa más cara del curso: el símbolo `&`.

- En **Verilog**, `&` es **AND** bit a bit.
- En **VHDL**, `&` es **concatenación** de vectores (pegar buses). El AND en VHDL se escribe `and`.

Si mezclas los lenguajes en el examen, esta confusión te puede tumbar un ejercicio entero. Antes de escribir `&`, pregunta: «¿estoy en Verilog o en VHDL?»

1.7. Tres estilos de escribir HDL (los verás todo el semestre)

1. **Estructural:** «instancio» compuertas o módulos como piezas de Lego (`and U1(y,a,b); / U1: entity work.and2...`).
2. **Dataflow:** ecuaciones continuas (`assign Y = A & B; / Y <= A and B;`).
3. **Comportamental:** `always/process` con `if` y `case`. Es el más flexible y donde aparecen los latches si olvidas un `else` o un valor por defecto.

1.8. Mini glosario de palabras que salen en el banco

- **Tabla de verdad:** todas las combinaciones de entradas → salida.
- **SOP / POS:** suma de productos / producto de sumas (forma canónica booleana).
- **Mapa de Karnaugh (K-map):** rejilla para simplificar a ojo agrupando unos (o ceros).
- **MUX:** multiplexor; elige una de varias entradas según un selector.
- **Decoder:** enciende una sola salida según el código de entrada.
- **Latch:** memoria *nivel-sensitiva* (mala si aparece por accidente en combinacional incompleto).
- **Flip-flop:** memoria *flanco-sensitiva* (la buena para reloj).
- **Síntesis:** traducir HDL → netlist de compuertas/FF para FPGA o ASIC.

1.9. Traducción rápida VHDL ↔ Verilog (para no ahogarte)

Idea	Verilog	VHDL
Contenedor	<code>module/endmodule</code>	<code>entity+architecture</code>
Cable	<code>wire</code> / a veces <code>reg</code>	<code>signal</code>
Asignación continua	<code>assign y = ...;</code>	<code>y <= ...;</code> (concurrente)
Bloque clocked	<code>always @(posedge clk)</code>	<code>process(clk) + rising_edge</code>
Combinacional	<code>always @(*)</code>	<code>process(todas las entradas)</code>
AND	<code>&</code>	<code>and</code>
Concatenar	<code>{a,b}</code>	<code>a & b</code>
Comentario	<code>//</code>	<code>-</code>

Cómo usar esta sección en el plan

Léela **una vez con calma** (15–20 min) antes de las plantillas. Luego no la vuelvas a leer entera: solo consulta la trampa del `&` y la tabla de traducción cuando te trabes al escribir código a mano.

2. Plan de estudio para las horas que quedan

Son seis bloques más el repaso. El orden importa: primero el vocabulario (sección de conocimiento previo), luego la teoría de la sección 3, que explica *por qué* el código del banco se escribe así.

Regla que atraviesa todo el plan: se estudia con lápiz. Como el examen es escrito, leer esta guía no sirve de nada por sí solo. Cada bloque termina escribiendo algo a mano **sin mirar**. Lo que puedas reproducir en papel es exactamente lo que tendrás en el examen; lo demás es ilusión de saberlo.

Bloque	Tiempo	Qué haces exactamente
0. Bases	20 min	Sección «Conocimiento previo»: qué es un HDL, VHDL vs Verilog, bits/compuertas, combinacional vs secuencial, y la trampa del <code>&</code> . Sin esto el banco es chino.
1. Teoría	60 min	Secciones 3 y 4 de esta guía. Al terminar, cierra el PDF y escribe en papel: el flujo de diseño de 7 pasos, la diferencia <code>signal/variable</code> y por qué aparece un latch.
2. Plantillas	45 min	Sección 2. Escribe a mano y sin mirar las seis plantillas (tres estilos × dos lenguajes). Repite hasta que salgan sin dudar. Esto es lo que más rinde por minuto.
3. Banco A	100 min	Ejercicios 1 a 5 de la sección 5. Lee la idea, tapa la solución, resuélvelo en papel, compara. No pases al siguiente sin haberlo escrito tú.
4. Banco B	100 min	Ejercicios 6 a 10. Igual. El 9 (ALU) y el 10 (NAND) son los dos más largos: déjales aire.
5. Secuencial	40 min	Sección 6. Flip-flop D, reset síncrono y asíncrono, contador y registro de desplazamiento. Son plantillas fijas: se memorizan tal cual.
Lunes, antes de entrar		
6. Repaso	30 min	Sección 7 (técnica de examen en papel y reparto del tiempo), sección 8 (los ocho errores que cuestan puntos) y sección 9 (autoevaluación de 15 preguntas). Si fallas una, vuelves solo a esa.

SI SOLO TIENES UNA HORA

Haz el bloque 0 (bases: VHDL/Verilog/`&`) en 10 min, el bloque 2 (plantillas) y luego los ejercicios **5** (full-adder), **3** (decodificador) y **10** (minimización + NAND). Esos tres tocan las tres habilidades que se repiten en todo lo demás: tabla de verdad, `case` sin latch, y simplificación algebraica.

3. Chuleta: lo que tienes que poder escribir de memoria

3.1. Las seis plantillas

Son seis y no más: tres estilos × dos lenguajes. Todo el examen práctico sale de rellenar una de estas.

Verilog — estructural (instanciar compuertas; la salida va siempre *primero* en la lista de puertos de una primitiva):

```

1 module nombre(input wire A, B, output wire Y);
2   wire n1, t1, t2; // cables internos: declararlos SIEMPRE
3   not U1(n1, B); // n1 = ~B (salida primero)
4   and U2(t1, A, n1); // t1 = A & ~B
5   or U3(Y, t1, t2); // Y = t1 | t2
6 endmodule

```

Verilog — dataflow (ecuaciones booleanas con `assign`; la salida es `wire`):

```

1 module nombre(input wire A, B, output wire Y);
2   assign Y = (A & ~B) | (A & B); // concurrente: el orden no importa
3 endmodule

```

Verilog — comportamental (la salida es `reg` aunque sea combinacional):

```

1 module nombre(input wire [1:0] S, output reg Y);
2   always @(*) begin // @(*) = lista de sensibilidad automática
3     Y = 1'b0; // valor por defecto: mata el latch
4     case (S)
5       2'b00: Y = 1'b1;
6       default: Y = 1'b0; // default SIEMPRE
7     endcase
8   end
9 endmodule

```

VHDL — dataflow:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity nombre is
5   port ( A, B : in std_logic;
6         Y : out std_logic );
7 end entity;
8
9 architecture dataflow of nombre is
10 begin
11   Y <= (A and not B) or (A and B); -- concurrente
12 end architecture;
```

VHDL — conductual (proceso combinacional):

```
1 architecture conductual of nombre is
2 begin
3   process(A, B) -- lista COMPLETA, o process(all)
4   begin
5     Y <= '0'; -- valor por defecto: mata el latch
6     if A = '1' and B = '0' then
7       Y <= '1';
8     end if;
9   end process;
10 end architecture;
```

VHDL — estructural (netlist escrito a mano):

```
1 architecture estructural of nombre is
2   component and2 port (a, b : in std_logic; y : out std_logic); end component;
3   signal t1 : std_logic;
4 begin
5   U1 : and2 port map (a => A, b => B, y => t1);
6   Y <= t1;
7 end architecture;
```

3.2. Tabla de traducción VHDL ↔ Verilog

Concepto	VHDL	Verilog
Interfaz	entity ... port(...)	lista de puertos del module
Implementación	architecture ... begin	cuerpo del module
Asignación concurrente	Y <= A and B;	assign Y = A & B;
Bloque secuencial	process(...)	always @(...)
Asignación a señal	<=	<= (no bloqueante)
Asignación a variable	:=	= (bloqueante)
NOT / AND / OR / XOR	not and or xor	~ & ^
Concatenación	&	{a,b}
Un bit	std_logic	wire / reg
Bus	std_logic_vector(3 downto 0)	[3:0]
Constante de 1 bit	'0' '1'	1'b0 1'b1
Constante de bus	"1010"	4'b1010 o 4'hA
Flanco de reloj	rising_edge(clk)	posedge clk
Comentario	--	//

La trampa que más cae aquí: el símbolo **&**. En Verilog es **AND**. En VHDL es **concatenación**. Si escribes `Y <= A & B;` en VHDL no estás haciendo un AND: estás pegando dos bits y creando un bus de dos. El AND en VHDL es la palabra **and**.

3.3. Operadores de Verilog que necesitas

Operador	Qué hace	Ejemplo
<code>~ & ^</code>	NOT, AND, OR, XOR bit a bit	<code>Y = A ^ B;</code>
<code>~^ o ^^</code>	XNOR	<code>eq = A ~^ B;</code> (1 si son iguales)
<code>^X</code>	XOR-reduction: XOR de todos los bits	<code>P = ^X;</code> (1 si X tiene un número impar de unos)
<code>&X X</code>	AND / OR de todos los bits	<code>todos = &X; alguno = X;</code>
<code>{a,b}</code>	concatenación	<code>case ({S1,S0})</code>
<code>{n{a}}</code>	replicación	<code>{4{1'b0}} = 4'b0000</code>
<code>s ? a : b</code>	multiplexor de 2 a 1	<code>assign y = s ? d1 : d0;</code>
<code>> < ==</code>	comparación (dan 1 bit)	<code>assign G = (A > B);</code>

Regla de oro de wire contra reg en Verilog. Si la señal se asigna con `assign` o la maneja una instancia, es **wire**. Si se asigna dentro de un `always`, es **reg** — *aunque el circuito sea puramente combinacional*. **reg** no significa registro: significa «variable del lenguaje». Es el nombre peor elegido de Verilog y por eso lo preguntan.

4. Teoría de la Unidad 1 (sesiones 2, 3 y 4)

Esto es lo que puede caer en preguntas de desarrollo. Cada apartado está en el orden en que lo dio el profesor.

4.1. El flujo de diseño: siete pasos

Va casi seguro. Memorízalo en orden:

1. **Especificación funcional** — qué debe hacer el sistema.
2. **Descripción en HDL** — escribir el diseño.
3. **Simulación funcional** — verificar el comportamiento con un *testbench*. Todavía no hay hardware.
4. **Síntesis** — traducir la descripción a compuertas y registros.
5. **Emplazamiento y ruteo** — asignar recursos físicos del dispositivo.
6. **Análisis de tiempos** — verificar que se cumplen las restricciones.
7. **Programación del dispositivo** y verificación en hardware.

TRUCO PARA NO PERDER EL ORDEN

Los pasos 1–3 son *qué quiero*; el 4 es la frontera (de descripción a compuertas); los 5–7 son *dónde y cuándo*. Si te trabas, recuerda que la simulación funcional va **antes** de sintetizar: se verifica la idea antes de gastar recursos.

4.2. Sintetizable y no sintetizable

No todo lo que simula se puede construir. Los **retardos explícitos** (`#10` en Verilog, `after 10 ns` en VHDL), el **acceso a archivos** y algunas construcciones existen **solo para el testbench**. La síntesis los ignora o los rechaza.

Cómo lo pueden preguntar: «da dos ejemplos de código que simula pero no sintetiza». Respuesta: un retardo explícito (`#10`) y la lectura de un archivo de vectores de prueba. Ambos describen *tiempo* o *entrada externa*, no estructura.

4.3. Concurrente contra secuencial: la idea que más cuesta

Un HDL **no es un lenguaje de programación**. No ejecuta una secuencia de instrucciones: describe una estructura de circuitos que existe toda al mismo tiempo.

- En software, dos líneas seguidas ocurren una *después* de la otra.
- En hardware, dos asignaciones concurrentes ocurren **simultáneamente**, siempre, en paralelo físico.

- Por eso el **orden en que escribes las asignaciones concurrentes no importa en absoluto**: describen conexiones, no pasos.

Dentro de un **process** (VHDL) o un **always** (Verilog) las instrucciones sí se evalúan en orden. Pero el proceso *completo* se comporta como un bloque concurrente respecto al resto del diseño. Esa doble naturaleza es la clave del lenguaje.

4.4. with-select contra when-else: sintetizan distinto

Las dos describen una selección, pero **no producen el mismo hardware**:

	with sel select	when ... else
Qué describe	selección por valor	asignación condicional
Qué sintetiza	un multiplexor plano	una cadena de multiplexores con prioridad
Coste	un nivel de lógica	tantos niveles como condiciones
Cuándo usarlo	condiciones mutuamente excluyentes	cuando de verdad quieres prioridad

Elegir mal cuesta niveles de lógica y por tanto retardo. El equivalente en Verilog es **case** (plano) frente a **if-else if** (con prioridad), y en VHDL dentro de un proceso, **case** frente a **if-elsif**.

4.5. Señal contra variable (VHDL)

	Símbolo	Cuándo se actualiza
Señal	<=	La actualización <i>se programa</i> y se hace efectiva al final del proceso . Representa un cable .
Variable	:=	Se actualiza de inmediato dentro del proceso. Representa un valor intermedio de cálculo.

«El error más frecuente del curso», palabras del profesor. Si dentro de un proceso asignas a una **señal** y la lees en la línea siguiente, **lees el valor viejo**, no el que acabas de escribir. Con una variable sí leerías el nuevo. Si te ponen un fragmento y te preguntan qué vale la salida, casi seguro es esto.

4.6. El latch que nadie pidió

Dos causas, y las dos las mencionó en clase:

1. **Lista de sensibilidad incompleta.** Describes un combinacional dentro de un proceso y olvidas una señal. La simulación se comporta distinto a la síntesis. Se corrige con la lista completa, con **process(all)** en VHDL, o con **always @(*)** en Verilog.
2. **Alguna ruta no asigna la salida.** La herramienta infiere un latch para *conservar el valor anterior*. Se evita asignando un **valor por defecto al inicio del proceso**, o poniendo **default** en el **case** y **else** en el **if**.

Por qué es malo: un latch es memoria que no pediste. Complica el análisis de tiempos, consume recursos y hace que el circuito dependa de un estado anterior en un bloque que debía ser puramente combinacional.

4.7. Flip-flop D y reinicio

El patrón es fijo y el profesor dijo literalmente que conviene memorizarlo tal cual:

```

1 process(clk)
2 begin
3   if rising_edge(clk) then
4     q <= d;
5   end if;
6 end process;
```

Lo que hace sintetizar un registro es exactamente eso: una asignación dentro de un **if de flanco**. No hay una palabra reservada «flip-flop»; el registro se *infiere* de la estructura.

	Dónde va el reset	Cuándo actúa
Asíncrono	en la lista de sensibilidad , y se evalúa <i>antes</i> del flanco	de inmediato, sin esperar al reloj
Síncrono	dentro del if de flanco	en el siguiente flanco de reloj

Pregunta casi segura: ¿cuál se prefiere en FPGA y por qué? El **síncrono**, por tres razones que dio en clase: usa **menos recursos**, evita **problemas de metaestabilidad al soltar el reset**, y **facilita el análisis de tiempos**.

4.8. Tipos de VHDL que importan

- `std_logic` — un bit con **nueve** valores posibles, entre ellos '0', '1', 'Z' de alta impedancia, 'X' de indeterminado y 'U' de no inicializado.
- `std_logic_vector` — un bus de bits.

Los valores 'X' y 'U' en la simulación **son información valiosa**: señalan que algo no se conectó o no se inicializó. No son ruido, son el diagnóstico.

4.9. VHDL contra Verilog

VHDL	Verboso, fuertemente tipado, detecta errores temprano. Es el del curso.
Verilog	Más compacto, cercano a C, más permisivo.

4.10. Circuitos combinacionales típicos que nombró

Multiplexor y demultiplexor. Decodificador y **codificador con prioridad**. Comparador. Unidad lógica de operaciones bit a bit. Todos aparecen en el banco menos el codificador con prioridad, que está en la sección 6.

5. Cómo atacar cualquier ejercicio del banco

Los diez ejercicios son variaciones de un mismo procedimiento. Si memorizas este orden, no te quedas en blanco:

1. **¿Cuántas entradas?** Con n entradas la tabla de verdad tiene 2^n filas. Con 3 entradas son 8, con 4 son 16.
2. **Escribe la tabla** contando en binario, siempre en el mismo orden (la primera columna es el bit más significativo).
3. **Saca la SOP**: un mintermino por cada fila con salida 1. Variable sin barra si vale 1, con barra si vale 0.
4. **Simplifica**: K-map o álgebra. Busca primero si es un XOR disfrazado o un MUX disfrazado.
5. **Elige el estilo que te piden** y rellena la plantilla de la sección 2.
6. **Verifica** sustituyendo dos o tres filas de la tabla en tu ecuación final.

Numeración de minterminos. m_k es la fila k contando desde 0, con la primera variable como bit más significativo. Con A, B, C : $m_5 = 101 = A\bar{B}C$. Con A, B, C, D : $m_{12} = 1100 = A\bar{B}\bar{C}\bar{D}$. Practica esta conversión: es mecánica y aparece en todos lados.

6. Banco de ejercicios, resuelto y explicado

Trece en total: los tres ejemplos que el profesor ya resolvió (porque enseñan el formato que espera) y los diez que dejó abiertos. Cada uno lleva la misma estructura: **la idea**, el **paso a paso**, el **código** en los dos lenguajes, la **trampa** y lo que hay que **memorizar**.

6.1. Los tres ejemplos resueltos

Ejemplo A — función con compuertas

estructural

Pide: para $Y = (A\overline{B}) + (CB)$, la tabla de verdad y la implementación estructural con primitivas **not**, **and**, **or**.

LA IDEA

Estructural es *dibujar el circuito con palabras*. No describes qué hace: describes qué compuertas hay y cómo se conectan. Por eso necesitas un nombre para cada cable intermedio.

La función tiene dos términos AND unidos por un OR, así que el circuito es: un inversor para $\overline{B}$, dos AND, y una OR final. Cuatro compuertas, tres cables internos.

Tabla de verdad

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Cómo se rellena

$Y = A\overline{B} + CB$. Vas fila por fila:

- $A\overline{B}$ vale 1 solo si $A = 1$ y $B = 0$: filas 100 y 101.
- CB vale 1 solo si $C = 1$ y $B = 1$: filas 011 y 111.
- Y es 1 si cualquiera de los dos lo es.

```

1 module comb_structural(
2   input wire A,
3   input wire B,
4   input wire C,
5   output wire Y
6 );
7   wire nB;
8   wire t1, t2;
9
10  not U1(nB, B); // nB = ~B
11  and U2(t1, A, nB); // t1 = A & ~B
12  and U3(t2, C, B); // t2 = C & B
13  or U4(Y, t1, t2); // Y = t1 | t2
14 endmodule

```

Y el mismo circuito en VHDL, que es el lenguaje del curso:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity comb_estructural is
5   port ( A, B, C : in std_logic;
6         Y : out std_logic );
7 end entity;
8
9 architecture estructural of comb_estructural is
10  signal nB, t1, t2 : std_logic; -- los cables internos
11 begin
12  nB <= not B;
13  t1 <= A and nB;
14  t2 <= C and B;
15  Y <= t1 or t2; -- las cuatro líneas son SIMULTANEAS
16 end architecture;

```

Matiz honesto: en VHDL, «estructural» en sentido estricto significa instanciar componentes con `port map`. Lo de arriba es flujo de datos con señales intermedias, que *describe la misma estructura* compuerta a compuerta. Si el examen pide estructural puro en VHDL, declara un `component` por compuerta e instáncialo con `port map` — el patrón está en la sección 2.

$Y = \overline{B}A + BC$ es un **multiplexor 2:1 con B de selector**: si $B = 0$ sale A , si $B = 1$ sale C . Reconocer esto te ahorra la tabla entera. Comprueba en la tabla: con $B = 0$ la salida copia a A ; con $B = 1$ copia a C .

Ejemplo B — sumador medio

dataflow

Pide: half-adder con $S = A \oplus B$ y $C_{out} = A \wedge B$, tabla y dataflow con `assign`.

LA IDEA

Sumar dos bits da como mucho 2, que en binario es 10: hacen falta dos bits de salida. El de abajo (S) es la suma módulo 2, y el de arriba (C_{out}) es el acarreo.

Por qué XOR y AND, y no otra cosa: S tiene que valer 1 cuando *exactamente uno* de los dos entra a 1 — esa es literalmente la definición del XOR. Y hay acarreo solo cuando *los dos* valen 1 — esa es la del AND.

Se llama *medio* porque no acepta acarreo de entrada. El que sí lo acepta es el sumador completo del ejercicio 5.

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Lee la última fila: $1 + 1 = 2$, que en binario es 10. Por eso $S = 0$ y $C_{out} = 1$. No es que «se pierda» el 1: se va al acarreo.

```

1 module half_adder_df(
2   input wire A,
3   input wire B,
4   output wire S,
5   output wire Cout
6 );
7   assign S = A ^ B;
8   assign Cout = A & B;
9 endmodule

```

```

1 architecture dataflow of half_adder is
2 begin
3   S <= A xor B;
4   Cout <= A and B;
5 end architecture;

```

Idea clave del estilo, en palabras del profesor: dataflow describe ecuaciones booleanas directamente; el sintetizador elige la estructura de compuertas. Tú dices *qué*, no *con qué*.

Ejemplo C — multiplexor 4:1

comportamental

Pide: MUX 4:1 con $D_0..D_3$, selectores S_1S_0 y salida Y , en comportamental con `always @(*)` y `case`.

LA IDEA

Un multiplexor es un conmutador: de varias entradas deja pasar una, la que diga el selector. Con 2 bits de selector eliges entre $2^2 = 4$ entradas.

Por qué comportamental encaja tan bien aquí: el `case` es literalmente la tabla de selección escrita en código. No dices cómo se construye el multiplexor, dices qué sale para cada valor del selector.

```

1 module mux4_beh(
2   input wire D0, D1, D2, D3,
3   input wire S1, S0,
4   output reg Y // reg porque se asigna dentro de un always
5 );
6   always @(*) begin
7     case ({S1,S0}) // {} concatena los dos selectores en un bus de 2
8       2'b00: Y = D0;
9       2'b01: Y = D1;
10      2'b10: Y = D2;
11      2'b11: Y = D3;
12      default: Y = 1'b0; // nunca sobra
13    endcase
14  end
15 endmodule

```

```

1 architecture conductual of mux4 is
2 begin
3   process(D0, D1, D2, D3, S) -- lista COMPLETA
4   begin
5     case S is
6       when "00" => Y <= D0;
7       when "01" => Y <= D1;
8       when "10" => Y <= D2;
9       when "11" => Y <= D3;
10      when others => Y <= '0'; -- obligatorio en VHDL
11    end case;
12  end process;
13 end architecture;

```

Por qué **when others** es obligatorio en VHDL y no es paranoia. `std_logic` tiene **nueve** valores, no dos. Un vector de 2 bits no tiene 4 combinaciones posibles sino $2^2 = 4$. Con solo "00", "01", "10" y "11" el **case** está *incompleto* y el compilador lo rechaza. En Verilog el **default** no es obligatorio para el compilador, pero sin él aparece un latch. Ponlos siempre los dos.

always @(*) fuerza lógica combinacional (sin memoria) porque construye la lista de sensibilidad sola. **Asigna Y en todos los casos.**

6.2. Los diez ejercicios

Ejercicio 1 — tabla, ecuación, Verilog

dataflow

Pide: dada la tabla de $F(A, B, C)$, (1) forma SOP, (2) simplificar con K-map o álgebra, (3) implementar en dataflow.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Paso 1 — SOP. Un mintérmino por cada fila con $F = 1$. Salen las filas 1, 2, 4 y 7:

$$F = \sum m(1, 2, 4, 7)$$

$$F = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

Paso 2 — simplificar. Aquí está la gracia del ejercicio. Dibuja el K-map:

A	BC			
	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Ningún 1 es adyacente a otro 1. El tablero es un ajedrez. Eso significa que **la SOP no se puede reducir**: los cuatro minterminos son implicantes primos esenciales y no se agrupan.

Pero *sí* hay una simplificación, y no es de K-map: ese patrón de ajedrez es la firma del XOR.

$$F = A \oplus B \oplus C$$

Por qué. El XOR de tres entradas vale 1 cuando hay un número **impar** de unos. Mira las filas ganadoras: 001, 010 y 100 tienen un uno; 111 tiene tres. Todas impares. Las perdedoras (000 con cero, y 011, 101, 110 con dos) son pares.

Ganancia real: pasas de **cuatro AND de 3 entradas más una OR de 4 a dos XOR**.

```
1 module comb1_df(input wire A, B, C, output wire F);
2   assign F = A ^ B ^ C; // paridad impar de tres bits
3 endmodule
```

```
1 architecture dataflow of comb1 is
2   begin
3     F <= A xor B xor C;
4   end architecture;
```

Si respondes solo «no se puede simplificar» te quedas a medias, y si respondes solo «es un XOR» te saltas lo que te piden. La respuesta completa tiene las dos partes: *en forma SOP no se reduce porque no hay adyacencias; reconocido como XOR, sí se reduce muchísimo.*

Firma del XOR en un K-map: tablero de ajedrez. Si ves los unos en diagonal sin tocarse, deja de buscar grupos y prueba XOR. Aparece otra vez en el ejercicio 5 (la suma del full-adder) y en el 8 (paridad).

Ejercicio 2 — $Y = (\overline{A}B) + (AC)$

estructural

Pide: implementarla estructural con **not/and/or**, con tabla de verdad.

LA IDEA

Misma receta que el ejemplo A, cambiando dónde va la barra. Ahora la negación está sobre A, y A aparece también sin negar en el otro término. Eso es la pista: es otro multiplexor.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$Y = \sum m(2, 3, 5, 7).$$

Cómo la rellenas rápido: con $A = 0$ el segundo término muere y queda $Y = B$; con $A = 1$ el primero muere y queda $Y = C$. Solo tienes que copiar la columna B en la mitad de arriba y la columna C en la de abajo.

Es decir: **MUX 2:1 con A de selector**.

```
1 module comb2_estructural(
2   input wire A, B, C,
3   output wire Y
4 );
5   wire nA, t1, t2; // declarar los internos: si falta uno, no compila
6
7   not U1(nA, A); // nA = ~A
8   and U2(t1, nA, B); // t1 = ~A & B
9   and U3(t2, A, C); // t2 = A & C
10  or U4(Y, t1, t2); // Y = t1 | t2
11 endmodule
```

En las primitivas de Verilog la salida va SIEMPRE primero. `and U2(t1, nA, B)` significa `t1 = nA & B`. Si escribes `and U2(nA, B, t1)` estás diciendo que la salida es `nA` — y `nA` ya lo maneja el `not`, así que tienes dos cosas manejando el mismo cable. Es el error de estructural que más se ve.

Ejercicio 3 — decodificador 2 a 4

comportamental

Pide: decodificador 2 a 4 con entradas `S1,S0` y salidas `Y0..Y3` one-hot, comportamental con `case` y `default`.

LA IDEA

Un decodificador hace lo **contrario** de un multiplexor. El MUX toma muchas entradas y saca una; el decodificador toma un número codificado en binario y **enciende exactamente una** de sus salidas.

One-hot quiere decir justo eso: una sola salida en 1, el resto en 0, siempre. Con 2 bits de entrada hay 4 salidas: $2 \rightarrow 4$.

Para qué sirve: seleccionar un chip entre varios, activar una fila de memoria, encender un dígito de un display. Es el traductor de «número» a «línea».

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Fíjate en la diagonal: la salida encendida es la que lleva el número que entra. $S = 10$ es 2 en decimal, y enciende Y_2 .

Ecuaciones, por si las piden: $Y_0 = \overline{S_1} \overline{S_0}$, $Y_1 = \overline{S_1} S_0$, $Y_2 = S_1 \overline{S_0}$, $Y_3 = S_1 S_0$.

```

1 module dec2a4_beh(
2   input wire S1, S0,
3   output reg Y0, Y1, Y2, Y3
4 );
5   always @(*) begin
6     {Y3,Y2,Y1,Y0} = 4'b0000; // por defecto TODO apagado
7     case ({S1,S0})
8       2'b00: Y0 = 1'b1;
9       2'b01: Y1 = 1'b1;
10      2'b10: Y2 = 1'b1;
11      2'b11: Y3 = 1'b1;
12      default: {Y3,Y2,Y1,Y0} = 4'b0000;
13    endcase
14  end
15 endmodule

```

Versión con bus, que es la que escribirías en la práctica:

```

1 module dec2a4_bus(
2   input wire [1:0] S,
3   output reg [3:0] Y
4 );
5   always @(*) begin
6     case (S)
7       2'b00: Y = 4'b0001;
8       2'b01: Y = 4'b0010;
9       2'b10: Y = 4'b0100;
10      2'b11: Y = 4'b1000;
11      default: Y = 4'b0000;
12    endcase
13  end
14 endmodule

```

```

1 architecture conductual of dec2a4 is
2 begin
3   process(S)
4   begin
5     Y <= "0000"; -- valor por defecto: mata el latch
6     case S is
7       when "00" => Y <= "0001";
8       when "01" => Y <= "0010";
9       when "10" => Y <= "0100";
10      when "11" => Y <= "1000";
11      when others => Y <= "0000";
12    end case;

```

```

13 end process;
14 end architecture;

```

La línea que salva el ejercicio es la asignación por defecto antes del **case**. Con las cuatro ramas cubiertas el latch no aparecería igualmente, pero el profesor pidió **default** explícitamente y es exactamente el punto que explicó en la sesión del 31-08. Escríbela siempre: cuesta un renglón y te cubre.

Ejercicio 4 — comparador de 2 bits

dataflow

Pide: entradas $A[1:0]$, $B[1:0]$, salidas G ($A > B$), E ($A = B$), L ($A < B$). Dataflow, y validar con tabla parcial de mínimo 6 casos bien elegidos.

LA IDEA

Comparar dos números binarios es lo mismo que comparar dos números decimales: **empiezas por la cifra más significativa**. Si los bits altos difieren, ya está decidido y el bit bajo no importa. Solo si los altos son iguales descienes a mirar el bajo.

Esa frase es toda la lógica del circuito, y es también lo que distingue un comparador bien hecho de uno mal hecho.

Versión corta (la que te piden en dataflow):

```

1 module comp2_df(
2   input wire [1:0] A, B,
3   output wire G, E, L
4 );
5   assign G = (A > B); // los comparadores devuelven 1 bit
6   assign E = (A == B);
7   assign L = (A < B);
8 endmodule

```

Versión con ecuaciones booleanas, por si piden nivel de compuerta. Usando $x \odot y$ para el XNOR (vale 1 si los dos bits son iguales):

$$\begin{aligned}
 E &= (A_1 \odot B_1) \cdot (A_0 \odot B_0) \\
 G &= A_1 \overline{B_1} + (A_1 \odot B_1) \cdot A_0 \overline{B_0} \\
 L &= \overline{A_1} B_1 + (A_1 \odot B_1) \cdot \overline{A_0} B_0
 \end{aligned}$$

Lee G en voz alta: « A es mayor si su bit alto es 1 y el de B es 0; o bien, si los bits altos son iguales, si su bit bajo es 1 y el de B es 0». Eso es exactamente la regla de comparar por la cifra más significativa.

Atajo válido: $L = \overline{G + E}$. Funciona porque las tres salidas son mutuamente excluyentes y exhaustivas: siempre hay exactamente una en 1. Ahorra compuertas, y explicar *por qué* es válido suma puntos.

Los 6 casos, y por qué esos:

A	B	G	E	L	Qué comprueba
00	00	0	1	0	igualdad en el extremo bajo
11	11	0	1	0	igualdad en el extremo alto
01	00	1	0	0	decide el <i>bit bajo</i> con los altos iguales
00	01	0	0	1	el simétrico del anterior
10	01	1	0	0	el bit alto manda sobre el bajo
01	10	0	0	1	el simétrico: 1 contra 2

Los dos últimos son los importantes: en 10 contra 01 el bit bajo de A es *menor* que el de B , y aun así $A > B$. Un comparador mal escrito — uno que dé prioridad al bit bajo — falla justo ahí y pasa todos los demás casos. Por eso son «bien elegidos».

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all; -- NECESARIO para comparar vectores
4
5 architecture dataflow of comp2 is
6 begin
7   G <= '1' when unsigned(A) > unsigned(B) else '0';
8   E <= '1' when A = B else '0';
9   L <= '1' when unsigned(A) < unsigned(B) else '0';
10 end architecture;

```

En VHDL **no puedes comparar dos std_logic_vector con > sin más**: un vector es una colección de bits, no un número. Hay que convertirlo con `unsigned(...)` o `signed(...)` de `ieee.numeric_std`. El `=` sí funciona directo, porque comparar igualdad bit a bit sí está definido. En Verilog no existe este problema, y esa diferencia es un ejemplo perfecto de «VHDL es fuertemente tipado».

Ejercicio 5 — sumador completo

tabla + dataflow + estructural

Pide: full-adder con A, B, C_{in} y salidas S, C_{out} . (1) tabla, (2) dataflow, (3) estructural instanciando **dos half-adders**.

LA IDEA

El sumador completo es el ladrillo con el que se construyen todos los sumadores de n bits: encadenas n de ellos pasando el C_{out} de uno al C_{in} del siguiente. Por eso necesita aceptar un acarreo de entrada, cosa que el medio no hace.

Suma tres bits. El resultado va de 0 a 3, que en binario necesita dos bits: C_{out} y S .

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truco para rellenarla sin equivocarte: cuenta cuántos unos hay en la fila. Ese número, escrito en binario con dos bits, es la salida.

Cero unos $\rightarrow 00$. Un uno $\rightarrow 01$. Dos unos $\rightarrow 10$. Tres unos $\rightarrow 11$.

Las ecuaciones, y de dónde salen.

S vale 1 cuando el número de unos es impar: 1 o 3. Eso es, otra vez, el XOR de tres — el mismo del ejercicio 1:

$$S = A \oplus B \oplus C_{in}$$

C_{out} vale 1 cuando hay **dos o más** unos. Es la función mayoría:

$$C_{out} = AB + AC_{in} + BC_{in} = AB + C_{in}(A \oplus B)$$

Las dos formas son equivalentes. La segunda es la que sale sola del montaje con dos half-adders, y por eso es la que conviene.

```

1 module full_adder_df(
2   input wire A, B, Cin,
3   output wire S, Cout
4 );
5   assign S = A ^ B ^ Cin;
6   assign Cout = (A & B) | (Cin & (A ^ B));
7 endmodule

```

Estructural con dos half-adders (lo que de verdad pide el punto 3):

```

1 module half_adder(input wire a, b, output wire s, c);
2   assign s = a ^ b;

```

```

3  assign c = a & b;
4  endmodule
5
6  module full_adder_est(
7      input wire A, B, Cin,
8      output wire S, Cout
9  );
10     wire s1, c1, c2;
11
12     // HA1 suma A y B
13     half_adder HA1 (.a(A), .b(B), .s(s1), .c(c1));
14     // HA2 suma ese resultado parcial con el acarreo de entrada
15     half_adder HA2 (.a(s1), .b(Cin), .s(S), .c(c2));
16     // hay acarreo de salida si lo genero cualquiera de los dos
17     or U1(Cout, c1, c2);
18 endmodule

```

La pregunta que suelen hacer aquí: ¿por qué OR al final y no XOR?

Porque c_1 y c_2 **no pueden valer 1 a la vez**, y es demostrable en una línea: si $c_1 = A \cdot B = 1$, entonces $A = B = 1$, luego $s_1 = A \oplus B = 0$, y por tanto $c_2 = s_1 \cdot C_{in} = 0$.

Como nunca coinciden, OR, XOR y hasta una suma darían el mismo resultado. Se usa OR por convención y porque es la compuerta más barata.

Ponerlos en fila hace un *ripple-carry adder*: el acarreo tiene que propagarse de la etapa 0 a la $n - 1$, y por eso el retardo crece con n . Es la razón de ser de los sumadores con acarreo anticipado. Si preguntan «cuál es la limitación», la respuesta es esa.

Ejercicio 6 — multiplexor 8:1

comportamental + dataflow en cascada

Pide: MUX 8:1 con $D_0 \dots D_7$ y $S[2:0]$. (1) comportamental con `case`, (2) dataflow con MUX en cascada (por ejemplo MUX2).

LA IDEA

Las dos implementaciones hacen lo mismo y sintetizan distinto — que es exactamente el tema de la sesión del 24-08.

El `case` describe un **multiplexor plano**: un nivel de lógica, todas las entradas compitiendo a la vez. La cascada describe un **árbol**: tres niveles, cada uno decidiendo con un bit del selector.

```

1  module mux8_beh(
2      input wire D0, D1, D2, D3, D4, D5, D6, D7,
3      input wire [2:0] S,
4      output reg Y
5  );
6      always @(*) begin
7          case (S)
8              3'b000: Y = D0;
9              3'b001: Y = D1;
10             3'b010: Y = D2;
11             3'b011: Y = D3;
12             3'b100: Y = D4;
13             3'b101: Y = D5;
14             3'b110: Y = D6;
15             3'b111: Y = D7;
16             default: Y = 1'b0;
17         endcase
18     end
19 endmodule

```

En cascada. Primero el ladrillo, un MUX 2:1 en una línea:

```

1  module mux2(input wire d0, d1, s, output wire y);
2      assign y = s ? d1 : d0; // si s=1 pasa d1, si s=0 pasa d0
3  endmodule

```

Y el árbol. Fíjate en qué bit del selector gobierna cada nivel:

```

1 module mux8_casc(
2   input wire D0, D1, D2, D3, D4, D5, D6, D7,
3   input wire [2:0] S,
4   output wire Y
5 );
6   wire n0, n1, n2, n3; // nivel 1: cuatro MUX2 gobernados por S[0]
7   wire m0, m1; // nivel 2: dos MUX2 gobernados por S[1]
8
9   mux2 A0 (.d0(D0), .d1(D1), .s(S[0]), .y(n0));
10  mux2 A1 (.d0(D2), .d1(D3), .s(S[0]), .y(n1));
11  mux2 A2 (.d0(D4), .d1(D5), .s(S[0]), .y(n2));
12  mux2 A3 (.d0(D6), .d1(D7), .s(S[0]), .y(n3));
13
14  mux2 B0 (.d0(n0), .d1(n1), .s(S[1]), .y(m0));
15  mux2 B1 (.d0(n2), .d1(n3), .s(S[1]), .y(m1));
16
17  mux2 C0 (.d0(m0), .d1(m1), .s(S[2]), .y(Y)); // nivel 3: S[2]
18 endmodule

```

Verifícalo con $S = 101$ ($S_2=1, S_1=0, S_0=1$), que debería sacar D_5 :

- Nivel 1, con $S_0 = 1$, cada MUX2 elige el segundo de su pareja: $n_0 = D_1, n_1 = D_3, n_2 = D_5, n_3 = D_7$.
- Nivel 2, con $S_1 = 0$, elige el primero: $m_0 = n_0 = D_1, m_1 = n_2 = D_5$.
- Nivel 3, con $S_2 = 1$, elige el segundo: $Y = m_1 = D_5$. **Correcto.**

$4 + 2 + 1 = 7$ multiplexores. En general, un **MUX $2^n:1$ se construye con $2^n - 1$ MUX2 en n niveles**. El plano gasta un nivel de lógica; el árbol gasta n . Ese es el compromiso entre **case** y cascada, y es justo lo que el profesor llamó «elegir mal cuesta niveles de lógica y por tanto retardo».

Ejercicio 7 — código Gray

dataflow

Pide: convertir binario $B[3:0]$ a Gray $G[3:0]$. Ecuaciones booleanas y módulo dataflow.

LA IDEA

El código Gray tiene una única propiedad, y de ella sale todo: **dos valores consecutivos difieren en exactamente un bit**.

Por qué importa de verdad. Imagina un encoder de posición pasando de 3 a 4. En binario eso es $011 \rightarrow 100$: cambian los tres bits. Los sensores no conmutan en el mismo instante exacto, así que durante unos nanosegundos puedes leer 111 (que es 7) o 000 (que es 0) — lecturas que no existen. En Gray solo cambia un bit, y una lectura a destiempo solo puede darte el valor viejo o el nuevo, nunca uno inventado.

Las ecuaciones. El bit más significativo se copia; cada uno de los demás es el XOR del bit binario y el que tiene encima:

$$G_3 = B_3 \quad G_2 = B_3 \oplus B_2 \quad G_1 = B_2 \oplus B_1 \quad G_0 = B_1 \oplus B_0$$

```

1 module bin2gray4(
2   input wire [3:0] B,
3   output wire [3:0] G
4 );
5   assign G[3] = B[3];
6   assign G[2] = B[3] ^ B[2];
7   assign G[1] = B[2] ^ B[1];
8   assign G[0] = B[1] ^ B[0];
9 endmodule

```

Y en una sola línea, que es lo mismo dicho con un desplazamiento:

```

1   assign G = B ^ (B >> 1); // desplazar a la derecha mete un 0 arriba,
2                             // así que G[3] = B[3] ^ 0 = B[3]. Encaja solo.

```

Comprobación con los ocho primeros valores:

B	G	B	G	B	G	B	G
0000	0000	0010	0011	0100	0110	0110	0101
0001	0001	0011	0010	0101	0111	0111	0100

La secuencia Gray es 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100. Recórrela y comprueba: entre cada dos consecutivos cambia **un solo bit**.

La vuelta (Gray a binario) es una cadena, no operaciones independientes: $B_3 = G_3$, y luego $B_2 = B_3 \oplus G_2$, $B_1 = B_2 \oplus G_1$, $B_0 = B_1 \oplus G_0$. Cada bit necesita el que acabas de calcular. Por eso ir a Gray es barato y volver es más caro.

Ejercicio 8 — generador de paridad par

dataflow

Pide: paridad **par** para $X[7:0]$. (1) expresarla con XOR-reduction, (2) implementarla, (3) proponer 4 vectores de prueba con la salida esperada.

LA IDEA

Paridad par significa: añades un bit P de forma que el **total** de unos (los ocho de X más el propio P) sea **par**. Entonces, ¿cuánto vale P ? Si X ya tiene un número par de unos, $P = 0$ (no hay que corregir nada). Si X tiene un número impar, $P = 1$ (para completar). Es decir: **P vale 1 exactamente cuando X tiene un número impar de unos** — que es, literalmente, la definición del XOR de todos los bits.

Para qué sirve: detectar un error de un bit en una transmisión. Si un bit se corrompe, la paridad deja de cuadrar.

$$P = X_7 \oplus X_6 \oplus X_5 \oplus X_4 \oplus X_3 \oplus X_2 \oplus X_1 \oplus X_0$$

```

1 module paridad_par(
2   input wire [7:0] X,
3   output wire P
4 );
5   assign P = ^X; // XOR-reduction: XOR de los 8 bits en un operador
6 endmodule
7
8 // paridad IMPAR seria: assign P = ~X;
```

Los 4 vectores de prueba:

X	unos	P	Por qué este vector
8'b0000_0000	0 (par)	0	el caso frontera: cero unos ya es par
8'b0000_0001	1 (impar)	1	el mínimo impar
8'b0000_0111	3 (impar)	1	impar con varios bits, no solo uno
8'b1111_1111	8 (par)	0	el otro extremo: todos a uno

Están elegidos para cubrir los dos extremos y las dos paridades. Con solo los dos primeros no se distingue «cuenta unos» de «mira el bit 0».

```

1 -- VHDL clasico no tiene operador de reduccion: se escribe la cadena
2 P <= X(7) xor X(6) xor X(5) xor X(4) xor
3     X(3) xor X(2) xor X(1) xor X(0);
```

Un generador de paridad detecta **un número impar de errores**. Si se corrompen **dos** bits, la paridad vuelve a cuadrar y el error pasa inadvertido. Y detectar no es corregir: sabes que algo falló, no qué bit fue. Si te preguntan por la limitación, es esta.

Ejercicio 9 — ALU de 4 bits

comportamental

Pide: ALU con $A[3:0]$, $B[3:0]$, selector $op[1:0]$ y salida $Y[3:0]$: $00 \rightarrow A + B$, $01 \rightarrow A - B$, $10 \rightarrow A \& B$, $11 \rightarrow A \oplus B$. Comportamental con case. Indicar si usas signed o unsigned y cómo tratas el overflow, documentando la decisión.

LA IDEA

Una ALU es un multiplexor con calculadoras dentro: las cuatro operaciones se calculan y el op elige cuál sale. Por eso se describe con un case, igual que un MUX.

Lo que de verdad califica en este ejercicio no es el case, que es mecánico, sino el párrafo sobre signed/unsigned y overflow. El enunciado lo pide explícitamente. Escríbelo aunque no te sobre tiempo.

```

1 module alu4(
2   input wire [3:0] A, B,
3   input wire [1:0] op,
4   output reg [3:0] Y,
5   output reg C, // acarreo (suma) o prestamo (resta)
6   output reg V // overflow interpretando en complemento a 2
7 );
8   reg [4:0] t; // 5 bits: uno mas para que quepa el acarreo
9
10  always @(*) begin
11    t = 5'd0; Y = 4'd0; C = 1'b0; V = 1'b0; // por defecto: sin latches
12    case (op)
13      2'b00: begin // SUMA
14        t = {1'b0, A} + {1'b0, B};
15        Y = t[3:0];
16        C = t[4];
17        V = (A[3] == B[3]) && (Y[3] != A[3]);
18      end
19      2'b01: begin // RESTA
20        t = {1'b0, A} - {1'b0, B};
21        Y = t[3:0];
22        C = t[4]; // 1 = hubo prestamo
23        V = (A[3] != B[3]) && (Y[3] != A[3]);
24      end
25      2'b10: Y = A & B; // AND bit a bit
26      2'b11: Y = A ^ B; // XOR bit a bit
27      default: Y = 4'd0;
28    endcase
29  end
30 endmodule

```

LA DECISIÓN DOCUMENTADA (esto es lo que se califica)

- 1. Unsigned.** En Verilog un wire [3:0] es **unsigned** por omisión, y así lo trato: A y B son números de 0 a 15. Si quisiera aritmética con signo tendría que declararlos **signed**.
- 2. Por qué el acumulador t tiene 5 bits.** Con 4 bits, $A + B$ puede llegar a 30, que no cabe. Concatenar un cero delante ($\{1'b0, A\}$) ensancha los operandos a 5 bits, así que el acarreo se guarda en $t[4]$ en vez de perderse. Si sumas directo en 4 bits, el desbordamiento desaparece sin avisar.
- 3. Dos banderas distintas.** C es el desbordamiento leyendo los números **sin signo**. V es el desbordamiento leyéndolos en **complemento a dos**. No son la misma cosa y hay casos donde una se enciende y la otra no.
- 4. Regla del overflow con signo.** La suma desborda si *los dos operandos tienen el mismo signo y el resultado tiene otro*. La resta desborda si *los operandos tienen signos distintos y el resultado no tiene el signo de A*. Es imposible que $(+) + (+)$ dé negativo, así que si pasa, desbordó.
- 5. AND y XOR nunca desbordan.** Son bit a bit: cada bit de salida depende solo del mismo bit de entrada. Por eso en esas ramas C y V se quedan en 0.

Un ejemplo que demuestra que C y V son cosas distintas.

Toma $A = 0111$ (7) y $B = 0001$ (1), operación suma. Resultado: $t = 0_1000$, luego $Y = 1000$ y $C = 0$.

- Sin signo:** $7 + 1 = 8$, y 8 cabe de sobra en 4 bits (llegan a 15). Por eso $C = 0$: correcto.
- Con signo:** en complemento a dos el rango es de -8 a $+7$. $7 + 1 = 8$ *no cabe*, y de hecho $Y = 1000$ se lee como -8 . Los dos operandos eran positivos y el resultado salió negativo, así que $V = 1$: también correcto.

La misma suma desborda o no según cómo decidas leer los bits. Esa frase es la respuesta a «documenta

tu decisión».

```

1 use ieee.numeric_std.all;
2 ...
3 process(A, B, op)
4   variable t : unsigned(4 downto 0); -- variable: se actualiza YA
5 begin
6   Y <= (others => '0'); C <= '0';
7   case op is
8     when "00" => t := ('0' & unsigned(A)) + ('0' & unsigned(B));
9                 Y <= std_logic_vector(t(3 downto 0)); C <= t(4);
10    when "01" => t := ('0' & unsigned(A)) - ('0' & unsigned(B));
11                Y <= std_logic_vector(t(3 downto 0)); C <= t(4);
12    when "10" => Y <= A and B;
13    when "11" => Y <= A xor B;
14    when others => Y <= (others => '0');
15  end case;
16 end process;

```

En ese código VHDL, `'0' & unsigned(A)` usa `&` como **concatenación**, no como AND: está pegando un cero delante para ensanchar a 5 bits, exactamente igual que `{1'b0,A}` en Verilog. Y `t` es una **variable** (`:=`), no una señal, precisamente para poder leerla en la línea siguiente con el valor nuevo.

Ejercicio 10 — minimización y NAND universal

el más largo

Pide: para $F(A, B, C, D) = \sum m(1, 3, 4, 5, 7, 9, 11, 12, 13, 15)$, (1) simplificar, (2) dataflow, (3) estructural **solo con NAND**.

Paso 1 — el K-map. Cuatro variables, 16 casillas. Recuerda que las columnas van en código Gray (00,01,11,10), no en orden binario: eso es lo que hace que las casillas contiguas difieran en un bit.

AB	CD			
	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

Paso 2 — agrupar. Dos grupos y ya está:

- **Las columnas 01 y 11 están llenas:** ocho casillas seguidas. ¿Qué tienen en común esas dos columnas? Que $D = 1$ en las dos. El grupo de 8 elimina tres variables y deja un solo literal: D .
- **Quedan sueltos m_4 y m_{12}** (columna 00, filas 01 y 11). Los agrupas con m_5 y m_{13} , que están justo al lado. Ese cuadro de 4 abarca $AB \in \{01, 11\}$ y $CD \in \{00, 01\}$: A cambia, D cambia, y lo que se mantiene es $B = 1$ y $C = 0$. Término: $B\overline{C}$.

$$F = D + B\overline{C}$$

Comprobación en dos líneas. Todos los minterminos impares (1, 3, 5, 7, 9, 11, 13, 15) tienen $D = 1$, así que los cubre el término D . Los dos pares que quedan, $m_4 = 0100$ y $m_{12} = 1100$, tienen $B = 1$ y $C = 0$: los cubre $B\overline{C}$. Y $m_6 = 0110$ no está en la lista: tiene $B = 1$ pero también $C = 1$, así que $B\overline{C} = 0$ y $D = 0$. Sale 0, correcto.

DOS COSAS QUE VALE LA PENA DECIR EN EL EXAMEN

1. **A no aparece.** La función es **independiente de A**: da igual cuánto valga. Si te dan cuatro entradas y la simplificación se come una, dilo explícitamente — demuestra que entendiste el resultado y no solo agrupaste casillas.
2. **Cuánto ahorras.** La SOP sin simplificar son **diez** AND de 4 entradas más una OR de 10. La simplificada es

un NOT, un AND de 2 y un OR de 2. De 11 compuertas grandes a 3 pequeñas.

Paso 3 — dataflow:

```
1 module f_df(input wire A, B, C, D, output wire F);
2   assign F = D | (B & ~C); // A no interviene
3 endmodule
```

Paso 4 — solo con NAND. Primero el porqué: la NAND es **universal**, es decir, con ella sola se construye cualquier función. Se demuestra fabricando las tres básicas:

Quiero	Con NAND	Por qué
NOT x	<code>nand(x,x)</code>	$\overline{x \cdot x} = \overline{x}$
AND xy	<code>nand(nand(x,y), nand(x,y))</code>	niego la NAND
OR $x + y$	<code>nand(nand(x,x), nand(y,y))</code>	De Morgan: $x + y = \overline{\overline{x} \cdot \overline{y}}$

Ahora la conversión de $F = D + B\overline{C}$, aplicando De Morgan a la OR final:

$$F = D + B\overline{C} = \overline{\overline{D} \cdot \overline{B\overline{C}}} = \text{NAND}(\overline{D}, \overline{B\overline{C}})$$

Y cada pieza es a su vez una NAND:

```
1 module f_nand(
2   input wire A, B, C, D, // A queda sin usar: la funcion no depende de el
3   output wire F
4 );
5   wire nC, p, nD;
6
7   nand U1(nC, C, C); // nC = ~C
8   nand U2(p, B, nC); // p = ~(B & ~C)
9   nand U3(nD, D, D); // nD = ~D
10  nand U4(F, nD, p); // F = ~(~D & ~(B & ~C)) = D | (B & ~C)
11 endmodule
```

Compruébalo con tres casos:

- $B=1, C=0, D=0$ (es m_4 , debe dar 1): $nC=1, p = \text{NAND}(1, 1)=0, nD=1, F = \text{NAND}(1, 0)=1$. Bien.
- $B=0, C=0, D=0$ (es m_0 , debe dar 0): $nC=1, p = \text{NAND}(0, 1)=1, nD=1, F = \text{NAND}(1, 1)=0$. Bien.
- $D=1$, lo demás da igual (debe dar 1): $nD=0$, y $\text{NAND}(0, \cdot)=1$ siempre. Bien.

La receta general SOP $\rightarrow$ NAND-NAND, que sirve para cualquier función: sustituye cada AND por una NAND y la OR final por otra NAND, e invierte los literales sueltos con `nand(x,x)`. Sale un circuito de **dos niveles**, la misma profundidad que el original. Por eso las bibliotecas reales se construyen con NAND: una sola compuerta, cualquier función, sin penalización de retardo.

7. Lo que el banco no trae pero la unidad 1 sí

El aviso del 31-08 dice que la evaluación «cubre la unidad 1 completa», y la sesión 4 entera fue de instrucciones secuenciales y flip-flops. El banco es solo combinacional. Esto es el hueco, y son plantillas fijas: se memorizan tal cual.

7.1. Flip-flop D — las tres versiones

```

1  -- SIN reset
2  process(clk)
3  begin
4      if rising_edge(clk) then
5          q <= d;
6      end if;
7  end process;
8
9  -- Reset SINCRONO: el reset se mira DENTRO del if de flanco
10 process(clk)
11 begin
12     if rising_edge(clk) then
13         if rst = '1' then q <= '0';
14         else q <= d;
15         end if;
16     end if;
17 end process;
18
19 -- Reset ASINCRONO: el reset esta en la LISTA DE SENSIBILIDAD y va primero
20 process(clk, rst)
21 begin
22     if rst = '1' then q <= '0';
23     elsif rising_edge(clk) then q <= d;
24     end if;
25 end process;

```

```

1  // Verilog, reset sincrono // Verilog, reset asincrono
2  always @(posedge clk) begin always @(posedge clk or posedge rst) begin
3      if (rst) q <= 1'b0; if (rst) q <= 1'b0;
4      else q <= d; else q <= d;
5  end end

```

La diferencia se ve en una sola cosa: dónde está **rst**. Si está en la lista de sensibilidad (`process(clk, rst)` o `or posedge rst`), es **asíncrono**. Si no está, es **síncrono**. Y en FPGA se prefiere el síncrono: menos recursos, sin metaestabilidad al soltarlo, y análisis de tiempos más fácil.

7.2. Contador ascendente con habilitación y carga

```

1  process(clk)
2  begin
3      if rising_edge(clk) then
4          if rst = '1' then q <= (others => '0'); -- prioridad 1
5          elsif load = '1' then q <= d; -- prioridad 2
6          elsif en = '1' then q <= q + 1; -- prioridad 3
7          end if; -- si no, CONSERVA
8      end if;
9  end process;

```

El `if-elsif` aquí **sí** quieres que genere prioridad: reset gana a carga, y carga gana a incrementar. Es el caso donde `if-elsif` es la elección correcta y no un descuido. Que no haya `else` final es *deliberado*: al ser un bloque de flanco, no asignar significa conservar el valor del registro, que es justo lo que se quiere.

7.3. Registro de desplazamiento

```

1  process(clk)
2  begin
3      if rising_edge(clk) then
4          if load = '1' then q <= d; -- carga paralela
5          else q <= q(2 downto 0) & serial; -- & = CONCATENA
6          end if;
7      end if;
8  end process;

```

Desplazar es tirar el bit de un extremo y meter uno nuevo por el otro. Aquí & es concatenación: pega los tres bits de abajo con el nuevo bit serie.

7.4. Codificador con prioridad

Lo nombré en la sesión 3 y no está en el banco. Es lo contrario del decodificador: recibe varias líneas y devuelve el *número* de la de mayor prioridad que esté activa.

```
1 process(x)
2 begin
3   valid <= '1';
4   if x(3) = '1' then y <= "11";
5   elsif x(2) = '1' then y <= "10";
6   elsif x(1) = '1' then y <= "01";
7   elsif x(0) = '1' then y <= "00";
8   else y <= "00"; valid <= '0'; -- ninguna activa
9   end if;
10 end process;
```

Por qué necesita la salida *valid*. Con las cuatro entradas a cero, *y* vale "00"; pero "00" también es la respuesta cuando está activa x_0 . Sin un bit extra no se distingue «la línea 0» de «ninguna línea». Es la pregunta clásica sobre codificadores.

8. Es en papel: qué cambia

No hay simulador, no hay autocompletado y no hay compilador que te avise. Eso mueve el peso del examen a tres cosas: escribir sintaxis correcta de memoria, verificar a mano, y presentar la respuesta de forma que se vea el razonamiento.

8.1. Cómo presentar cada respuesta

Siempre en este orden, aunque no te lo pidan explícitamente. Si te quedas sin tiempo, lo que llevas escrito ya vale puntos:

1. **Tabla de verdad** (o tabla de selección, si es un MUX o un decodificador). Es lo más barato de escribir y lo primero que buscan.
2. **Ecuación en SOP**, sacada de la tabla.
3. **Simplificación**, con el K-map dibujado y **los grupos rodeados**. Rodear los grupos es parte de la respuesta: demuestra el método, no solo el resultado.
4. **Código**, en el estilo que te pidan.
5. **Una línea de verificación**: sustituye una fila de la tabla en tu ecuación final y escribe que coincide.

Escribe primero la cabecera del módulo, siempre igual. `module nombre(...); ... endmodule`, o `entity ... architecture`. Aunque no sepas todavía qué va dentro, la estructura ya está puesta y no se te olvida cerrarla. En papel, un `endmodule` o un `end process`; que falta es un error tonto que sí se penaliza.

8.2. Cómo verificar sin simulador

Cuatro técnicas, todas de menos de un minuto:

- **Sustituir dos filas.** Elige una fila donde la salida sea 1 y otra donde sea 0, y evalúa tu ecuación final. Si las dos cuadran, casi seguro está bien.
- **Contar unos.** En sumadores y paridad, el número de unos de la entrada determina la salida. Es un chequeo instantáneo.
- **Casos frontera.** Todo a cero y todo a uno. Son los que más fallos destapan y los más rápidos de evaluar.
- **Contar filas.** Con n entradas la tabla tiene 2^n filas exactas. Si te salen 7 con 3 entradas, te comiste una.

8.3. Reparto del tiempo

El examen es de 18:00 a 19:40: **100 minutos**.

Minutos	Qué	Por qué
0–5	Leer todo	Marca cuáles son de teoría (rápidos) y cuáles de diseño (lentos).
5–25	La teoría	Flujo de diseño, signal/variable, latch, reset. Son puntos baratos y de memoria: cáelos primero, mientras estás fresco.
25–85	Los diseños	Reparte a partes iguales. Si uno se atasca, escribe la tabla y la cabecera del módulo y pasa al siguiente.
85–100	Repaso	Cierra <code>endmodule</code> y <code>end process</code> . Comprueba que cada <code>case</code> tiene <code>default</code> / <code>when others</code> . Sustituye una fila en cada ecuación.

La regla que más nota salva: nunca dejes una hoja en blanco por no saber terminar. La tabla de verdad, la cabecera del módulo y la ecuación SOP se pueden escribir *siempre*, aunque no sepas simplificar. Eso ya es la mayor parte del ejercicio.

9. Errores que cuestan puntos

Los ocho que más caen, en orden de frecuencia:

1. **case sin default / when others.** Latch. En VHDL ni siquiera compila con `std_logic_vector`.
2. **No asignar la salida en alguna rama.** Mismo latch. Solución: valor por defecto al principio del proceso.
3. **Lista de sensibilidad incompleta.** La simulación difiere de la síntesis. Usa `always @(*)` o `process(all)`.
4. **Confundir señal y variable en VHDL.** Asignas a una señal con `<=` y la lees en la línea siguiente: lees el **valor viejo**. El profesor lo llamó «el error más frecuente del curso».
5. **& en VHDL creyendo que es AND.** Es concatenación. El AND es `and`.
6. **Declarar como wire algo que se asigna dentro de un always.** Tiene que ser `reg`, aunque el circuito sea combinacional.
7. **Olvidar los cables internos en estructural,** o poner la salida de una primitiva en un sitio que no sea el primero.
8. **Perder el acarreo** sumando dos números de n bits en n bits. Hay que ensanchar a $n + 1$.

10. Autoevaluación: 15 preguntas

Tápate las respuestas. Si fallas una, vuelve solo a esa sección.

1. Los siete pasos del flujo de diseño, en orden.
2. Dos ejemplos de código que simula pero no sintetiza.
3. ¿Por qué el orden de las asignaciones concurrentes no importa?
4. Diferencia entre **with-select** y **when-else**, y qué sintetiza cada una.
5. Señal contra variable: símbolo y momento de actualización.
6. Las dos causas de un latch no deseado y cómo se evitan.
7. ¿Dónde va **rst** en un reset asíncrono? ¿Y en uno síncrono?
8. ¿Cuál se prefiere en FPGA y por qué (tres razones)?
9. ¿Cuántos valores tiene **std_logic** y para qué sirven 'X' y 'U'?
10. En Verilog, ¿cuándo una señal es **wire** y cuándo **reg**?
11. Simplifica $\sum m(1, 2, 4, 7)$ de tres variables.
12. Las dos ecuaciones del full-adder.
13. ¿Por qué en el full-adder estructural la última compuerta es OR y no XOR?
14. Las cuatro ecuaciones de binario a Gray de 4 bits.
15. ¿Cómo se hace un NOT, un AND y un OR usando solo NAND?

RESPUESTAS

1. Especificación funcional, descripción en HDL, simulación funcional, síntesis, emplazamiento y ruteo, análisis de tiempos, programación del dispositivo. **2.** Retardos explícitos (**#10**, **after 10 ns**) y acceso a archivos; solo existen para el testbench. **3.** Porque describen conexiones, no pasos: el circuito existe todo a la vez, en paralelo físico. **4.** **with-select** sintetiza un multiplexor plano (un nivel); **when-else** una cadena con prioridad (tantos niveles como condiciones). **5.** Señal **<=**, se actualiza al final del proceso, es un cable; variable **:=**, se actualiza de inmediato, es un valor intermedio. **6.** Lista de sensibilidad incompleta, y alguna ruta que no asigna la salida. Se evitan con la lista completa (o **process(all)** / **always @(*)**) y con un valor por defecto al inicio. **7.** Asíncrono: en la lista de sensibilidad, evaluado antes del flanco. Síncrono: dentro del **if** de flanco. **8.** El síncrono: menos recursos, evita metaestabilidad al soltar el reset, y facilita el análisis de tiempos. **9.** Nueve. 'X' indica indeterminado y 'U' no inicializado: señalan que algo no se conectó o no se inicializó. **10.** **wire** si se asigna con **assign** o desde una instancia; **reg** si se asigna dentro de un **always**, aunque sea combinacional. **11.** $F = A \oplus B \oplus C$. En SOP no se reduce: no hay adyacencias. **12.** $S = A \oplus B \oplus C_{in}$ y $C_{out} = AB + C_{in}(A \oplus B)$. **13.** Porque c_1 y c_2 nunca valen 1 a la vez: si $c_1 = AB = 1$ entonces $s_1 = 0$ y $c_2 = 0$. **14.** $G_3 = B_3$, $G_2 = B_3 \oplus B_2$, $G_1 = B_2 \oplus B_1$, $G_0 = B_1 \oplus B_0$. **15.** NOT: **nand(x,x)**. AND: negar la NAND, **nand(n,n)** con $n = \text{nand}(x,y)$. OR: **nand(nand(x,x), nand(y,y))**, por De Morgan.

Si de todo esto solo te llevas una frase:

un HDL no ejecuta pasos, describe estructuras que existen todas a la vez.

Casi todo lo demás — el latch, el orden de las asignaciones, la diferencia entre señal y variable — es una consecuencia de eso.