VERIFY THIS!

Introducción a UVM

DÍA 1 · DÍA 2 · DÍA 3 · DÍA 4 · DÍA 5 · DÍA 6 · DÍA 7 · DÍA 8

github.com/leandrotozzi/verifythis · SystemVerilog · Verilator · UVM 2020.3.1

Cómo usar este curso

Sobre todo si lo estás haciendo solo

- **Qué hay que saber antes:** Verilog o VHDL, y haber simulado algo. Programar orientado a objetos **no** hace falta — el día 2 es exactamente eso. Lo único que el curso da por sabido es **SystemVerilog**: si venís de VHDL o de Verilog-2001, media hora con `docs/systemverilog-para-el-que-viene-de-vhdl.md` antes de empezar
- Son **siete días de clase**, de 4 a 5 horas y cuarto — **≈ 31 h**, y cada agenda trae el suyo. Y un **día 8 opcional**, después del cierre. Solo, calculá el doble: la mitad se va en correr los ejemplos, y ésa es la que enseña
- `s` abre las **notas del presentador** en otra ventana · `n` las muestra acá abajo, sin salir de la página. Ahí está lo que el instructor diría en voz alta: la trampa clásica, el porqué del número, el error que comete todo el mundo la primera vez. **No las saltees**
- Cada día cierra con un **repaso** de opciones clickeables y con un **ejercicio que se corrige solo**: `cd code/ejercicios/d1 && bash run.sh`. Los nombres de los otros dieciocho están en `code/ejercicios/README.es.md`

Cómo usar este curso

Correrlo, navegarlo, y dónde buscar cuando algo no anda

- **Corré los ejemplos**: `make u4/tests` corre uno, `make u4` la unidad entera. Un curso de verificación que sólo se lee no sirve para nada. Un ejemplo con UVM tarda **un minuto y medio la primera vez y quince segundos la segunda**, si tenés `ccache` instalado — y si no tenés Linux, el camino corto es GitHub Codespaces, que ya lo trae
- `0` – `8` saltan a la portada y a cada día · `i` abre el índice · `Esc` muestra el deck entero · y el ribbon de la esquina despliega el machete del día
- **Si estás solo, leé el libro:** `libro/dia1.html` es este mismo curso de corrido, con las notas ya adentro del texto. Tené el deck al lado para los repasos
- **Cuando algo no ande, andá al final:** los apéndices *La caja de herramientas de debug* —qué mirar según el síntoma— y *Las 21 trampas mudas* —todo lo que compila, corre y miente—. Son las dos slides para imprimir

Agenda

Día 1 · ≈ 4 h · unidades 1 y 2

Unidad 1 · Por qué se verifica

- Tendencias
- Qué es UVM
- La spec del VTALU
- El plan de verificación

Unidad 2 · El testbench sin UVM

- El testbench convencional
- Cobertura funcional
- Interfaces y BFM · el `clocking block`

Dos ejercicios, y el segundo se resuelve con el visor de ondas

- **Y el cierre del día**, que vuelve a la primera slide

Al final del día podés:

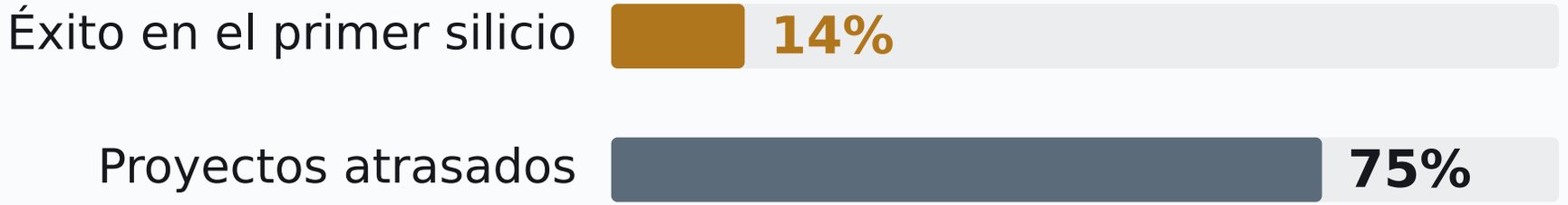
- **Escribir** un plan de verificación de cinco columnas para un DUT que no viste antes
- **Probar** que tu scoreboard chequea algo: meterle un bug al DUT a propósito y verlo fallar
- **Leer** el número que sale de un `covergroup`, decir qué fila del plan falta, y qué mirar primero cuando da 0 %
- **Explicar** qué race evita un `clocking block` — y por qué igual son opcionales

Seis de cada siete

Y el bug que llegó al silicio pasó una regresión en verde

Cómo terminan los proyectos IC/ASIC

IC/ASIC, 2024



14% es el valor más bajo en 20 años de encuesta. Atrasados: 75%, contra 66% histórico.

Datos: Wilson Research Group / Siemens EDA, estudio 2024. Gráfico propio.

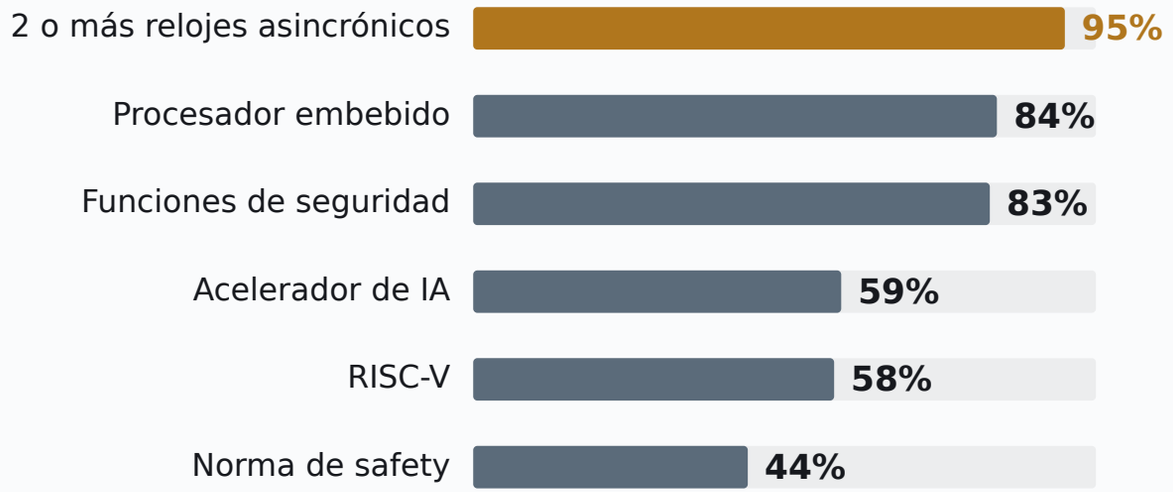
- El **14 %** de los proyectos acierta el primer silicio. Seis de cada siete vuelven a la fábrica, y es el peor valor en veinte años de encuesta
- Cada bug funcional que llegó al silicio pasó antes por un testbench. Ese testbench corrió, terminó, y **dijo que estaba bien**
- Nadie manda a fabricar con un `FAILED` en el log. Se manda con un `PASS`
- La pregunta de hoy es una sola: **¿cómo sabés que verificaste?** Y *"corrí muchos tests"* no es una respuesta

Tendencias

Por qué vuelven: lo que hay adentro de un chip hoy

Qué tienen adentro los IC/ASIC de hoy

IC/ASIC, 2024



RISC-V casi duplicó respecto del estudio anterior; los aceleradores de IA pasaron de ~30% a 59%.

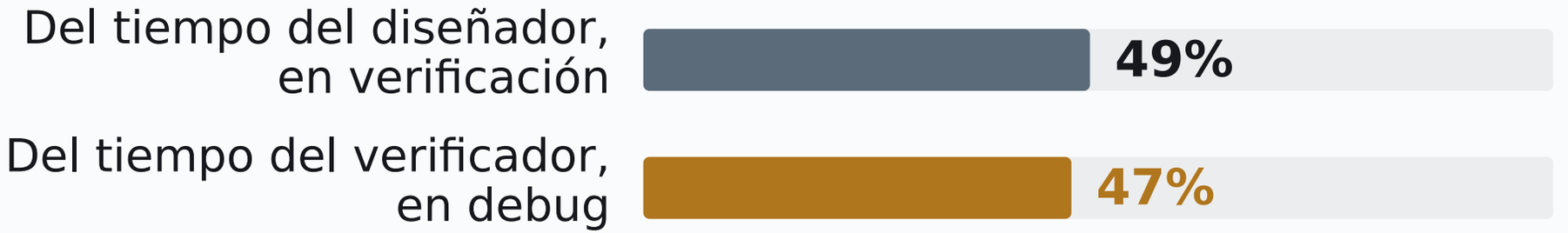
Datos: Wilson Research Group / Siemens EDA, estudio 2024. Gráfico propio.

Tendencias

Verificar no es una etapa, es la mitad del trabajo

Adónde se va el tiempo

IC/ASIC, 2024



En proyectos de procesadores la relación verificación:diseño llega a 5 a 1.

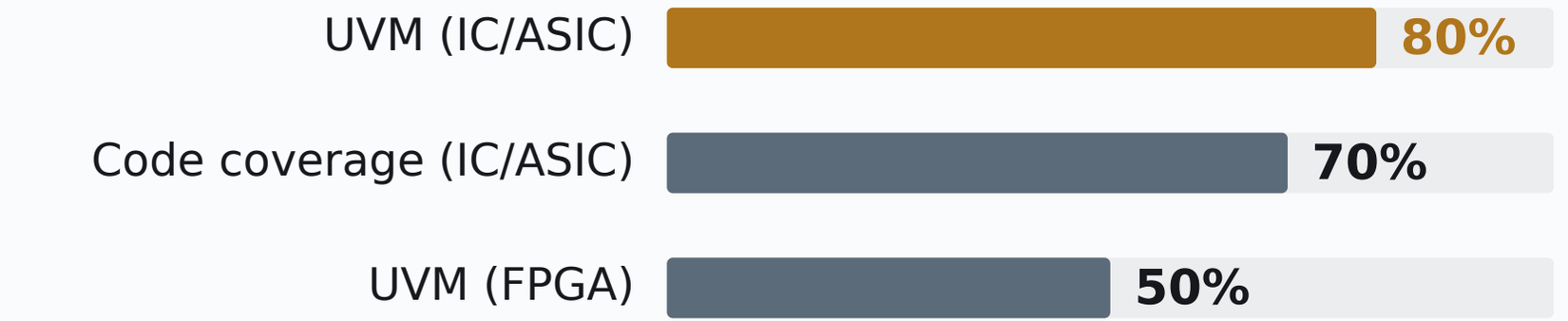
Datos: Wilson Research Group / Siemens EDA, estudio 2024. Gráfico propio.

Tendencias

Y se verifica, sobre todo, con UVM

Con qué se verifica

IC/ASIC y FPGA, 2024



Formal creciendo: 5,8% anual en property checking, 8,7% en aplicaciones automáticas.

Datos: Wilson Research Group / Siemens EDA, estudio 2024. Gráfico propio.

¿SystemVerilog es el *COBOL* de la electrónica?

Lenguaje	Reserved Keywords
ANSI COBOL 85	357
SystemVerilog	248
VHDL 2008	115
Verilog 95	102
C#	102
C++ 20	92
Python 3	35

> *" No academic computer scientists participated in the design of COBOL; all of those on the committee came from commerce or government"* ¿Parecido no?*

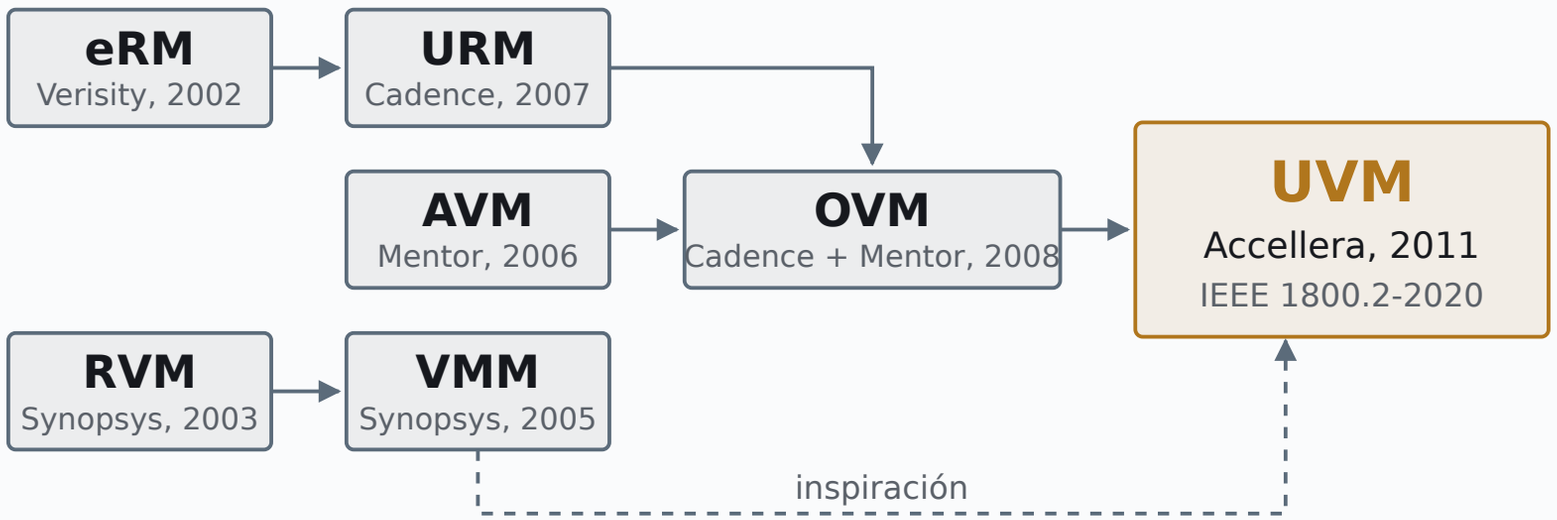
Introducción

¿Qué es UVM?

- **U**niversal **V**erification **M**ethodology: una **librería de clases** escrita en SystemVerilog, más un montón de **convenciones** sobre cómo usarla
- No es un lenguaje y no es un simulador. Todo lo que vamos a ver en el curso se puede escribir a mano — y en los días 1 y 2 lo vamos a escribir a mano
- La hace **Accellera**, el mismo consorcio que estandarizó SystemVerilog, y es **IEEE 1800.2**. El curso usa la implementación de referencia, `uvm-core 2020.3.1`
- Viene de OVM (Cadence + Mentor), con ideas de VMM (Synopsys) y de eRM (Verisity): las metodologías de cuatro vendors que competían, hasta que la industria se cansó de traducir testbenches entre herramientas
- Y ahí está el objetivo, que es uno solo: **que el testbench del de al lado se parezca al tuyo**

Introducción

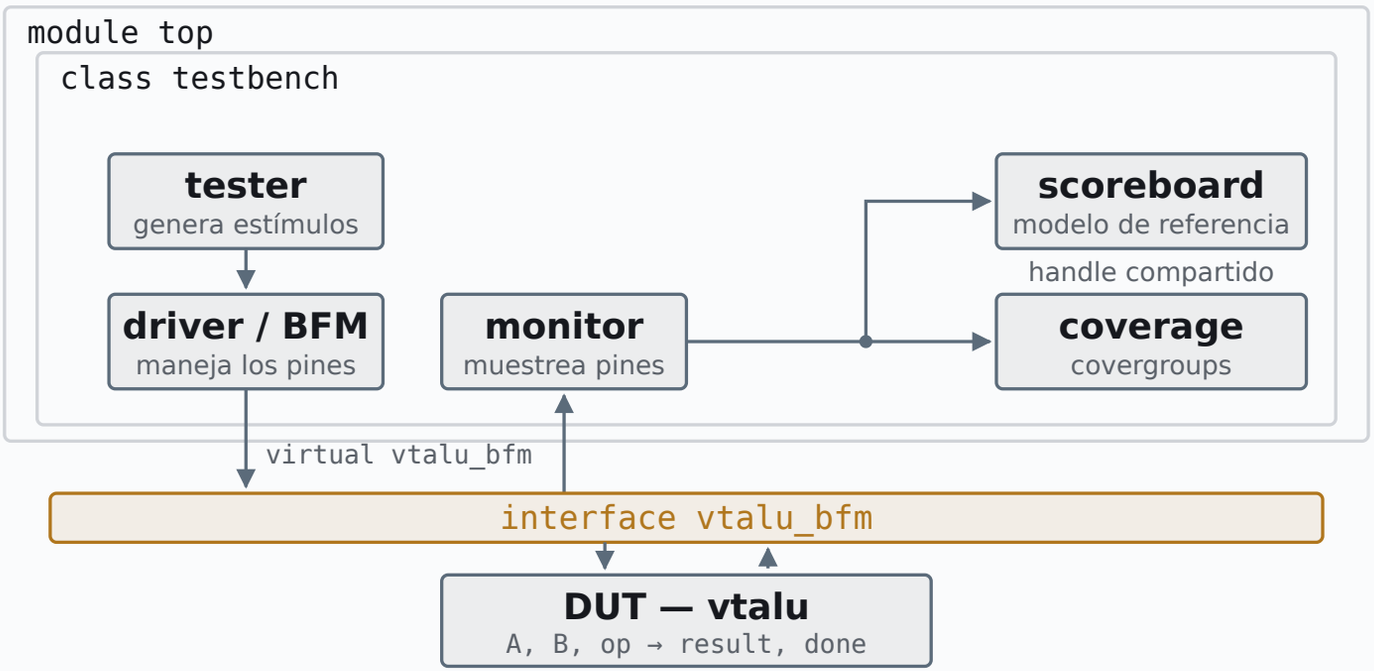
De dónde viene



OVM (2008) fusionó AVM de Mentor con URM de Cadence; Accellera tomó OVM 2.1.1 como base de U
VMM aportó ideas, no código. IEEE estandarizó UVM en 1800.2-2017 y luego en 1800.2-2020.

Introducción

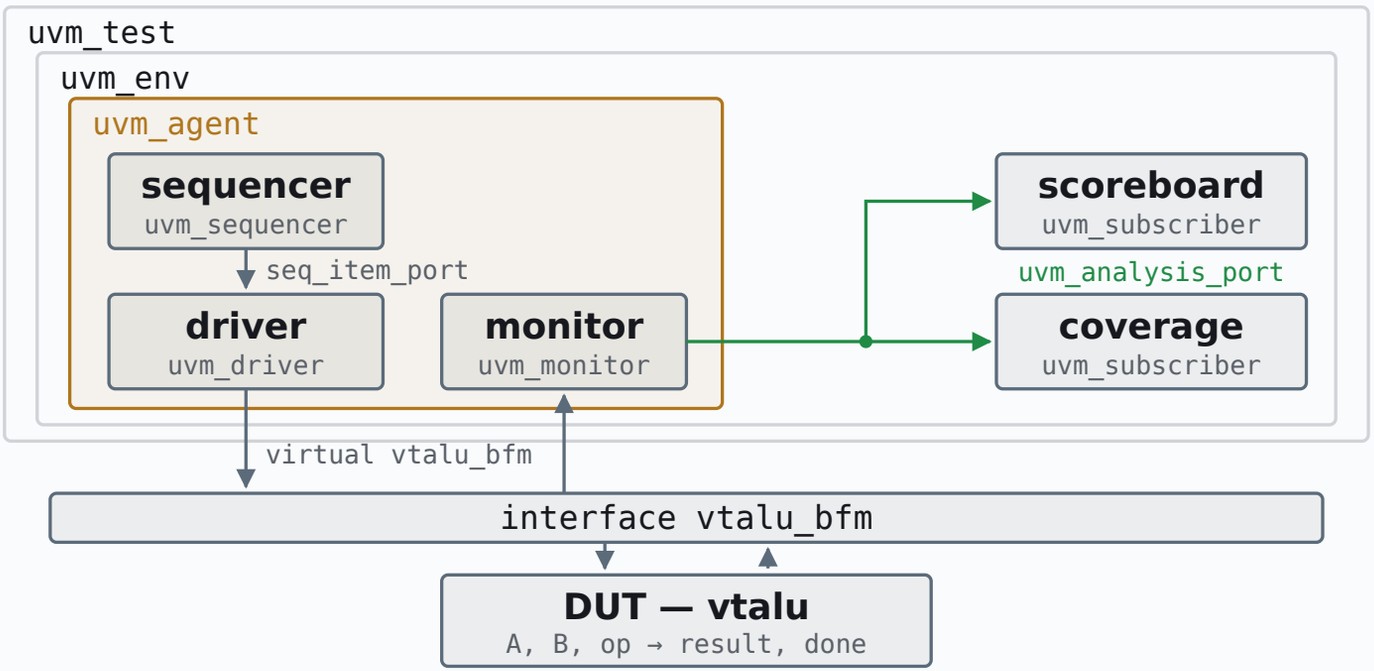
Un testbench de SystemVerilog, por dentro



La interface es la frontera: de un lado clases y transacciones, del otro pines y RTL.

Introducción

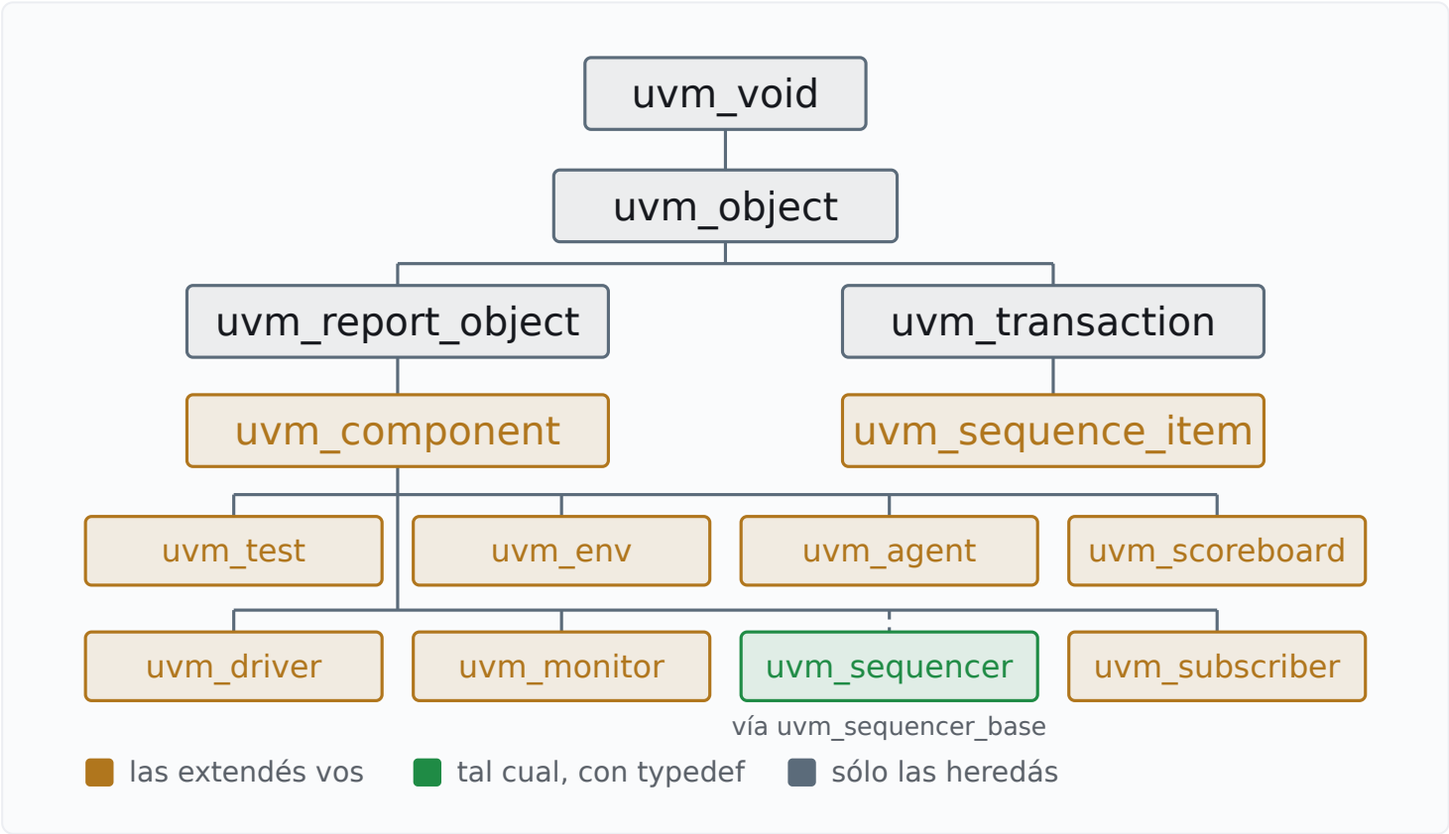
El mismo testbench, con estructura UVM



Mismas piezas, otra estructura: test → env → agent, y las conexiones pasan a ser analysis ports.

Introducción

El árbol de clases

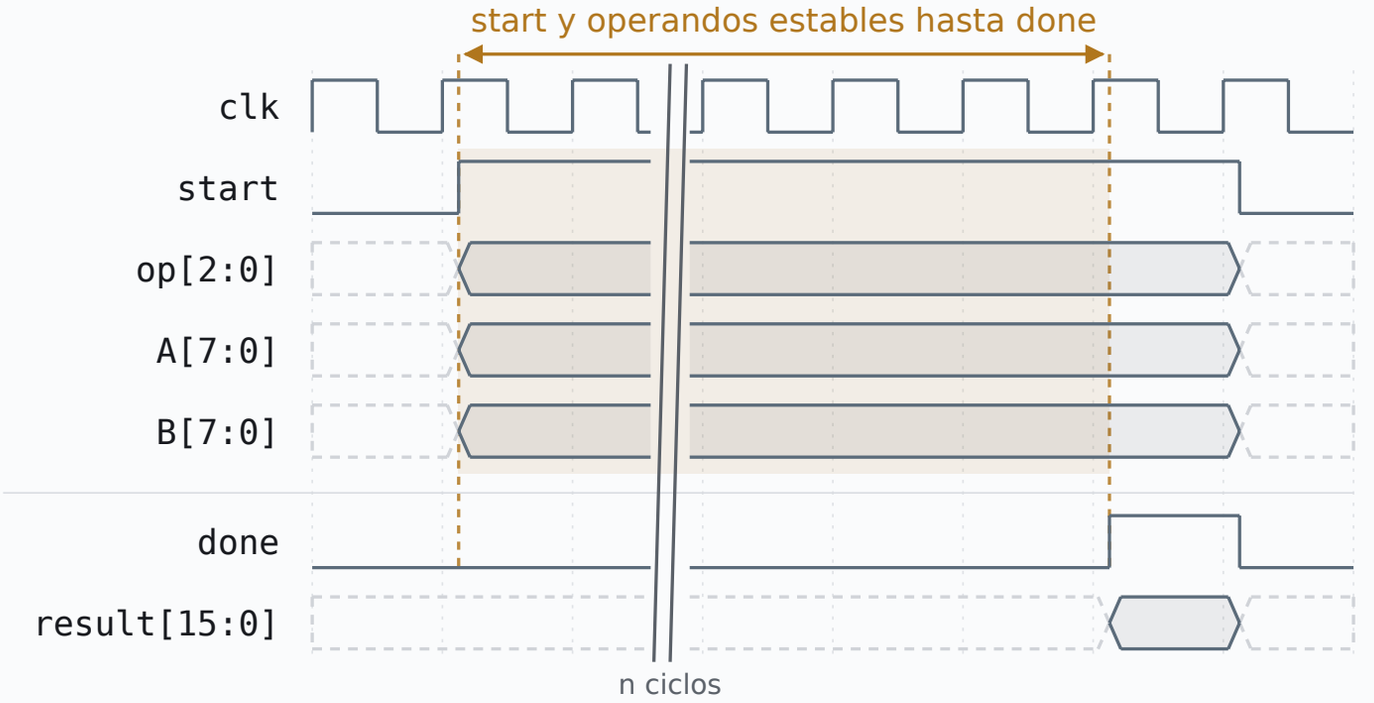


Introducción

Correr más tests escribiendo menos código

- Ésa es la promesa, y tiene una condición: **el env y los componentes casi no cambian** de un test a otro. Lo que cambia es el estímulo
- Para que eso funcione, el testbench tiene que dejar **ganchos**: métodos virtuales, la factory y los callbacks. Un test engancha ahí y no toca la estructura
- Un test nuevo entonces es una de estas tres cosas, y ninguna toca el env:
  - agregar **constraints** para llegar a un caso de borde
  - hacer un **override** de la factory para cambiar una clase por otra
  - inyectar errores o delays con **callbacks**
- Y arriba de todo eso, la palanca que más rinde: **el mismo test con cientos de semillas** distintas

La spec del VTALU



El testbench espera `done` para muestrear `result`: la latencia depende de la operación.

- *start* debe permanecer en 1 y los operandos estables hasta que termina la operación
- *done* se levanta cuando termina la operación: en la multiplicación es un pulso de un ciclo; en add/sub/and/xor es un **nivel**, y se queda arriba mientras *start* siga arriba
- *reset_n* es asíncrono y activo bajo; el diagrama arranca con el DUT ya fuera de reset
- *ovf* sale junto con *done*, y es el **borrow de la resta**: vale 1 cuando `A < B`. Para las otras operaciones vale siempre 0

La spec del VTALU

Operation	Opcode	Ciclos	ovf
no_op	3'b000	—	0
add_op	3'b001	1	0
sub_op	3'b010	1	`A < B`
and_op	3'b011	1	0
xor_op	3'b100	1	0
mul_op	3'b101	4	0
libre	3'b110	1	0
rst_op	3'b111	—	—

- **Ciclos** cuenta flancos de `clk` con `start` arriba, hasta el que levanta `done` inclusive: uno en las de un ciclo, cuatro en la multiplicación. El plan y las assertions del día 7 cuentan igual
- `rst_op` no lo decodifica el RTL — el DUT lo ve como *unused*. Es la convención del testbench para pulsar `reset_n`, y por eso está en el `operation_t` del testbench
- **`3'b110` está libre a propósito**: es el ejercicio de hoy

La spec del VTALU

Single Cycle: Add - Sub - AND - XOR

```
// The two resets are deliberately different: the one on the result is
// SYNCHRONOUS and the one on done is ASYNCHRONOUS. Not an oversight: the spec.
always_ff @(posedge clk) begin
   if (!reset_n) begin
      result_1c <= 16'h0000;
      ovf_1c    <= 1'b0;
   end else if (start) begin
      ovf_1c <= (op == 3'b010) && (A < B);
      case (op)
         3'b001:  result_1c <= {8'h00, A} + {8'h00, B};
         3'b010:  result_1c <= {8'h00, A} - {8'h00, B};
         3'b011:  result_1c <= {8'h00, A} & {8'h00, B};
         3'b100:  result_1c <= {8'h00, A} ^ {8'h00, B};
         default: ;  // 3'b110 is free: it is the day 1 exercise
      endcase
   end
end

// done goes high on the edge if start is high and the op is not no_op
always_ff @(posedge clk or negedge reset_n) begin
   if (!reset_n) done_1c <= 1'b0;
   else done_1c <= start && (op != 3'b000);
end
```

- Dos `always_ff` y nada más: uno registra `A op B`, el otro levanta `done`
- Los resets **no son iguales**: el del resultado es **síncrono** —sólo `clk` en la lista de sensibilidad—, el del `done` es **asíncrono**
- `done <= start && (op != no_op)`: por eso en estas cuatro operaciones `done` es un **nivel** y se queda arriba mientras `start` lo esté

La spec del VTALU

Multi Cycle: Multiplicación

```
// Four register stages. done is pipelined alongside the data, and each
// stage clears itself with (& ~done_mult): that is what makes done a
// one-cycle pulse instead of staying high for as long as start does. It is
// subtle -- it is copied straight from the VHDL.
always_ff @(posedge clk or negedge reset_n) begin
   if (!reset_n) begin
      done_mult <= 1'b0;
      done3 <= 1'b0;
      done2 <= 1'b0;
      done1 <= 1'b0;
      a_int <= 8'h00;
      b_int <= 8'h00;
      mult1 <= 16'h0000;
      mult2 <= 16'h0000;
      result_mult <= 16'h0000;
   end else begin
      a_int <= A;
      b_int <= B;
      mult1 <= a_int * b_int;
      mult2 <= mult1;
      result_mult <= mult2;
      done3 <= start & ~done_mult;
      done2 <= done3 & ~done_mult;
      done1 <= done2 & ~done_mult;
      done_mult <= done1 & ~done_mult;
   end
end
```

- Es un pipeline: los operandos se registran, se multiplican, y el producto atraviesa dos registros más antes de salir por `result_mult`: cuatro flancos
- El `done` viaja por **la misma cadena** —`done3`, `done2`, `done1`— así que llega exactamente con el dato y no antes
- El `& ~done_mult` de cada etapa es lo que apaga la cadena sola: por eso acá `done` es un **pulso de un ciclo**, aunque `start` siga arriba

La spec del VTALU

Top Level

```
assign es_mult = (op == 3'b101);

assign start_1c = es_mult ? 1'b0 : start;
assign start_mult = es_mult ? start : 1'b0;
assign result_mux = es_mult ? result_mult : result_1c;
assign done = es_mult ? done_mult : done_1c;
```

- El top no calcula nada: instancia los dos bloques y **decodifica** el opcode
- `es_mult = (op == mul_op)`. El `start` se rutea a uno solo, y `result_mux` y `done` salen del mismo lado. El `ovf` sigue el mismo patrón justo abajo del recorte — `assign ovf = es_mult ? 1'b0 : ovf_1c;` — y `result_mux` sale por `result` recién después del XOR de `+VTALU_BUG`
- Por eso el protocolo del bus es uno solo aunque adentro haya dos latencias distintas: lo que el testbench ve es `start` → `done`

La spec del VTALU

Cómo se corre

- El DUT vive en `code/vtalu_dut/`, en **SystemVerilog**
- El simulador del curso es **Verilator** (libre, sin licencia):

```
make u4/tests          # un ejemplo con UVM:  ~1 min 30 la primera vez
make u4/tests          # la segunda, con ccache:      ~15 segundos
make               # los 38 ejemplos
```

- Los que usan UVM compilan **la librería entera** la primera vez. Instalá `ccache` antes de empezar: el curso lo detecta solo y no se recompila dos veces
- Cada ejemplo tiene su `run.sh`; los flags comunes están en `code/verilator/common.sh`

La spec del VTALU

Ver las ondas de verdad

- El diagrama de la primera slide está **dibujado**. El de verdad lo generás vos, y es la herramienta con la que se hace el 47 % del trabajo: debuggear

```
cd code/u2/convencional
VLT_TRACE=1 bash run.sh      # compila con --trace y vuelca vtalu.vcd
gtkwave vtalu.vcd            # o surfer, o el visor que uses
```

- `VLT_TRACE=1` prende dos cosas a la vez: el `--trace` de Verilator y el `$dumpfile`/`$dumpvars` del top, que sin el flag no compilaría
- Sale gratis y es libre, igual que el simulador. Cuando el testbench se cuelgue en `while (done == 0)`, esto es lo que te va a decir por qué

La spec del VTALU

Resumen de la unidad

- El **protocolo** es una sola frase, y hay que respetarla: `start` en 1 y los operandos estables **hasta que sube** `done`
- La letra chica que va a colgar al primer testbench: `done` es un **pulso** de un ciclo en la multiplicación y un **nivel** en las de un ciclo
- Adentro hay **dos bloques** con latencias distintas —uno y cuatro ciclos— y el top **decodifica** el opcode. Afuera, el bus es uno solo
- `ovf` es una salida más, y es del `sub_op` y de nadie más
- `rst_op` **no existe para el RTL**: es una convención del testbench para pulsar `reset_n`, y por eso entra en la cobertura
- El DUT no valida el opcode: el que está libre **levanta** `done` **igual**, con el `result` de la operación anterior. Atajarlo es trabajo del **plan de verificación**, no del diseño
- Y la herramienta que se usa el 47 % del tiempo ya está instalada: `VLT_TRACE=1` y GTKWave

El plan de verificación

El entregable más profesional de la disciplina, y el más barato

- Es una **tabla.** Se escribe **antes** que el testbench — si se escribe después, describe lo que el testbench ya hace en vez de lo que la spec pide
- Sale de la spec que acabás de leer, no del testbench que todavía no existe
- Y es lo que se entrega: el capstone del día 7 se entrega con el plan lleno, que es exactamente lo que se entrega en un proyecto

El plan de verificación

Las cinco columnas

Columna	La pregunta que contesta	Qué no va
Feature	¿de qué parte de la spec sale esta fila?	el nombre de un archivo del testbench
Escenario	¿qué situación concreta hay que provocar?	*"probar la ALU"*
Estímulo	¿quién la provoca: el random o un caso dirigido?	*"a mano"*
Chequeo	¿quién dice que estuvo bien?	*"se mira la onda"*
Medida	¿qué bin se llena cuando pasa?	*"se ve en el log"*

- Una fila por **escenario**, no por feature: una feature con tres casos borde son tres filas, y se cierran de a una

El plan de verificación

Dos reglas que salen de las columnas

- **Si la columna de chequeo dice "a ojo", el escenario no está verificado.** Está *simulado*, que es otra cosa
- **Si la columna de medida está vacía, nadie se va a enterar de que el escenario nunca pasó.** Un caso que el random no tocó y que no tiene bin es indistinguible de uno que pasó mil veces
- Y una distinción que ordena la columna de chequeo: **el scoreboard chequea *qué* calcula el DUT; las assertions chequean *cómo* se habla con él**

El testbench convencional

El plan del VTALU, con tres columnas por llenar

Feature	Escenario	Estímulo	Chequeo	Medida
ALU	las seis operaciones			
ALU	operandos en `00` y en FF			
reset	operar después de un reset			
mult	una mult después de una de un ciclo			
ALU	la misma operación dos veces seguidas			
sub	restar de menos: `A < B`, y el `ovf` sube			

- Seis filas que salen de la spec, y todavía ninguna columna de la derecha. Esta sección llena **dos**: el estímulo las provoca y el scoreboard dice si el resultado estuvo bien
- La tercera —**qué bin se llena cuando pasa**— es la sección que sigue
- El testbench tiene tres partes porque el plan tiene tres columnas por llenar

El testbench convencional

El estímulo: mil operaciones, y el protocolo a mano

```
repeat (1000) begin
   @(negedge clk);
   op_set = get_op();
   A = get_data();
   B = get_data();
   start = 1'b1;
   case (op_set) // handle the start signal
     no_op: begin
        @(posedge clk);
        start = 1'b0;
     end
     rst_op: begin
        reset_n = 1'b0;
        start = 1'b0;
        @(negedge clk);
        reset_n = 1'b1;
     end
     default: begin
        // Wait for an EDGE with done up, not for done itself: on the
        // one-cycle ops done is a level still up from the previous op,
        // and a plain wait(done) drops this one without a sound.
        enviadas++;
        do @(negedge clk); while (done == 0);
        start = 1'b0;
     end
   endcase // case (op_set)
end
$finish;
```

- Mil vueltas: elegir operación, elegir operandos, levantar `start`, esperar **un flanco con `done` arriba**, bajar `start`
- El `case` está ahí por la letra chica de la spec: `no_op` no levanta `done`, `rst_op` pulsa `reset_n`, y las demás esperan
- `get_op()` y `get_data()` sesgan el random hacia los bordes —00 y FF— para llegar a los casos que el plan pide. Es *constrained random* escrito a mano
- `enviadas` se cuenta acá y `chequeadas` en el scoreboard, y un `final` exige que sean iguales: es el testbench chequeándose a sí mismo
- Fijate quién sabe del protocolo acá: **el tester**. En interfaces y BFM eso se muda a la BFM y en el cuerpo del `repeat` se achica a una línea

El testbench convencional

El self-checking: predecir y comparar

```
// Sent by the tester, compared by the scoreboard: the final block below
// demands that the two match.
int enviadas, chequeadas;

// Scoreboard loop: checks that everything came out right
always @(posedge done) begin : scoreboard
   shortint predicted_result;
   bit      predicted_ovf;
   #1;
   case (op_set)
     add_op: predicted_result = A + B;
     sub_op: predicted_result = A - B;
     and_op: predicted_result = A & B;
     xor_op: predicted_result = A ^ B;
     mul_op: predicted_result = A * B;
   endcase // case (op_set)

   // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
   // output, neither the addition nor the multiplication can overflow.
   predicted_ovf = (op_set == sub_op) && (A < B);

   if ((op_set != no_op) && (op_set != rst_op)) begin
     chequeadas++;
     if (predicted_result != result || predicted_ovf != ovf)
       $error ("FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
               A, B, op_set.name(), result, ovf);
   end
end : scoreboard
```

- Un `always @(posedge done)`: cada vez que el DUT dice que terminó, el scoreboard predice el resultado y lo compara
- El `#1` no es un adorno: sin él se leen las señales en el mismo instante en que `done` sube y se puede comer una carrera de deltas
- `no_op` y `rst_op` no levantan `done`, así que este bloque no debería correr con ellas. El `if` es defensivo —sin predicción para esas dos compararía basura— y adentro va `chequeadas++`, la otra mitad del contador
- El covergroup —la tercera pata— es la sección que sigue

El testbench convencional

¿Cómo sabés que el scoreboard chequea algo?

```
FAILED: A: 15  B: ce  op: xor_op result: da ovf: 0
```

- Mil operaciones y ningún error. Ahora **el mismo testbench con el DUT roto a propósito**: `VTALU_BUG=1 bash run.sh` da vuelta el bit 0 del resultado
- Falla en la primera comparación. Eso —y no el `PASS` de antes— es lo que prueba que el scoreboard mira el resultado
- Un scoreboard que **nunca vio un error** no está probado: pudo haber comparado contra sí mismo, o no haber comparado nada
- `make mutante` hace esto con los tres testbenches de los días 1 y 2, y falla si alguno **no** falla. Es la primera respuesta a la pregunta del día

El testbench convencional

Resumen de la unidad

- El testbench tiene **tres partes** porque el plan tiene tres columnas por llenar: estímulo, self-checking y medida. Hoy están sueltas en un archivo
- El estímulo **sesga el random hacia los bordes** — `00` y `FF` — porque el azar uniforme casi nunca los visita
- El `#1` antes de leer las señales no es adorno: sin él se lee en el mismo instante en que el DUT escribe, y el resultado es indefinido
- Un `PASS` no prueba nada hasta que viste el `FAILED`: **`VTALU_BUG=1`** es la forma de verlo, y `make mutante` la de no olvidarse
- Y lo que hay que mirar para interfaces y BFM: **el tester sabe del protocolo**. Mueve `start` y espera `done` con la mano

Cobertura funcional

¿Cuándo terminás de verificar?

- *Cobertura de código* —líneas, ramas, toggle— la da el simulador gratis. Dice qué RTL se **ejecutó**
- *Cobertura funcional* la escribís vos. Dice qué **escenarios de la spec** pasaron por el DUT
- No son lo mismo, y la primera engaña: un test que manda siempre `add_op` con `A=B=8'h01` ejecuta casi todas las líneas del sumador y no verificó nada
- La cobertura funcional es el *contrato* entre la spec y el testbench: si un caso de la spec no tiene un bin, nadie se va a enterar de que falta

Cobertura funcional

Del plan de verificación al covergroup

Las seis filas del plan de la sección anterior, traducidas:

Del plan	Cómo se mide
las seis operaciones	`coverpoint op_set` con un bin por operación
operandos en `00` y en FF	`cross` de A, B y la operación
operar después de un reset	bin de transición `rst_op => op`
una mult después de una de un ciclo	bin de transición
la misma operación dos veces seguidas	bin de repetición `[* 2]`
restar de menos: `A < B`, y el `ovf` sube	`coverpoint borrow` con el bin `hubo_borrow`

- Las seis filas dejan de ser buenas intenciones: cada una es una línea de código, y el simulador te dice cuáles todavía no pasaron

Cobertura funcional

Anatomía: son tres pasos, no uno

```
covergroup op_cov;                      // 1. QUE se mide
   coverpoint op_set;
endgroup

op_cov oc;                              // 2. se INSTANCIA: es un tipo, como una clase
initial oc = new();

always @(posedge clk) oc.sample();      // 3. CUANDO se cuenta
```

- Un `covergroup` es un **tipo**: declararlo no mide nada, hay que hacerle `new()`
- `sample()` es el que cuenta. Sin `sample()` el reporte da **0 %** y el código compila igual
- *Dónde* muestrear es una decisión de diseño. Acá, el flanco en el que **lee el DUT**, nunca el mismo en el que escribe el tester; cuando el TB tenga monitores va a ser *"cada vez que el monitor ve un comando"*

Cobertura funcional

bins: los casilleros que hay que llenar

```
coverpoint op_set;                    // bins automaticos: uno por valor del enum

coverpoint op_set {
   bins single_cycle[] = {[add_op : xor_op], rst_op, no_op};  // [] -> uno por valor
   bins multi_cycle    = {mul_op};                            // sin [] -> uno solo
}
```

- Sin `{ }`, SystemVerilog crea un bin por valor. Anda para un `enum`; para un `int` la herramienta corta el rango en **64** casilleros — `auto_bin_max` — que no corresponden a ninguna fila del plan
- Los corchetes `[]` **reparten**: un casillero por valor del rango. Sin ellos, todos los valores caen en el mismo casillero
- Un coverpoint está cubierto cuando **todos** sus bins tienen al menos un hit. Un bin que no se puede llenar te clava el número para siempre

Cobertura funcional

El covergroup del VTALU

```
   //op_cov: make sure every operation, and every interaction between them, is covered
   covergroup op_cov;
      coverpoint op_set {
         bins single_cycle[] = {[add_op : xor_op], rst_op,no_op};
         bins multi_cycle = {mul_op};

         // => are transitional coverage. From => To
         // Transition bins. Verilator 5.052 does NOT implement them yet: any form
         // with "=>" or "[* n]" aborts with an Internal Error. Minimal repro in
         // code/verilator/repro-cg-transition.sv. They stay because they are part
         // of the topic; when Verilator supports them, drop the ifndef, nothing else.
`ifndef VERILATOR
         bins opn_rst[] = ([add_op:mul_op] => rst_op);
         bins rst_opn[] = (rst_op => [add_op:mul_op]);

         bins sngl_mul[] = ([add_op:xor_op],no_op => mul_op);
         bins mul_sngl[] = (mul_op => [add_op:xor_op], no_op);

         bins twoops[] = ([add_op:mul_op] [* 2]);
         bins manymult = (mul_op [* 3:5]);
`endif
      }
   endgroup
```

- `single_cycle[]` genera seis bins —uno por operación—, `multi_cycle` uno solo
- Lo que está entre `` `ifndef VERILATOR `` son los bins de transición, que vienen más adelante en esta misma sección

Cobertura funcional

ignore_bins: decir en voz alta lo que no se cubre

```
// Check the corners where the inputs are all 0s or all 1s
all_ops : coverpoint op_set {
   ignore_bins null_ops = {rst_op, no_op};}

a_leg: coverpoint A {
   bins zeros = {8'h00};
   bins others= {[8'h01:8'hFE]};
   bins ones  = {8'hFF};
}

b_leg: coverpoint B {
   bins zeros = {8'h00};
   bins others= {[8'h01:8'hFE]};
   bins ones  = {8'hFF};
}
// VTALU rev2: the borrow of the subtraction, the row of the plan that the
// scoreboard can only close by looking at TWO outputs. Without this bin, a
// subtraction that never went negative looks exactly like one that did.
borrow: coverpoint ((op_set == sub_op) && (A < B)) {
   bins hubo_borrow = {1};
   ignore_bins resto = {0};
}
```

- `all_ops` **ignora** `rst_op` y `no_op`: no tiene sentido pedir "una suma con los operandos en 0x00" cuando la operación es un reset
- `a_leg` y `b_leg` bajan 256 valores a los tres casilleros que importan: todo ceros, el medio, todo unos
- Un bin imposible que dejás puesto te deja la cobertura clavada abajo del 100 % y nadie sabe por qué. `ignore_bins` lo vuelve una decisión escrita

Cobertura funcional

cross: las combinaciones, y cómo no ahogarse

```
op_00_FF : cross a_leg, b_leg, all_ops;    // 3 x 3 x 5 = 45 bins, casi todos sin sentido
```

- Un `cross` es el **producto cartesiano** de sus coverpoints: crece rápido y la mayoría de las combinaciones no está en ninguna spec
- `binsof(x) intersect {v}` se lee *"los bins de `x` que contienen `v`"*
- Se combinan con `&&` y `||`, y lo que sobra se tira con `ignore_bins`
- Así los 45 bins quedan en los 11 que el plan pidió de verdad

Cobertura funcional

El cross del VTALU

```
bins mul_00 = binsof (all_ops) intersect {mul_op} &&
              (binsof (a_leg.zeros) || binsof (b_leg.zeros));

bins mul_FF = binsof (all_ops) intersect {mul_op} &&
              (binsof (a_leg.ones) || binsof (b_leg.ones));

bins mul_max = binsof (all_ops) intersect {mul_op} &&
               (binsof (a_leg.ones) && binsof (b_leg.ones));

ignore_bins others_only =
               binsof(a_leg.others) && binsof(b_leg.others);
```

- `mul_00` se lee de corrido: *una multiplicación en la que A **o** B valgan 0x00*. Los otros ocho bins del cross son estas dos líneas con otro `op`
- `mul_max` es el único con `&&`: pide las **dos** patas en 0xFF: el **producto máximo**, `FF` × `FF` = `FE01`. No desborda — 8 bits por 8 entran en 16
- Verilator 5.052 **ignora** `binsof` / `intersect` (`%Warning-COVERIGN`) y mide el cross completo: por eso el número no es el de una herramienta comercial

Cobertura funcional

Bins de transición: cuando importa el orden

```
bins opn_rst[] = ([add_op:mul_op] => rst_op);   // A y despues B
bins sngl_mul[] = ([add_op:xor_op], no_op => mul_op);
bins twoops[]  = ([add_op:mul_op] [* 2]);       // dos veces seguidas
bins manymult  = (mul_op [* 3:5]);              // entre 3 y 5 veces seguidas
```

- Tres filas del plan no son valores, son **secuencias**: "después de un reset", "una mult después de una de un ciclo"
- Verilator 5.052 todavía **no los compila** (Internal Error). En el código están entre `` `ifndef VERILATOR ``: son parte del tema y se leen igual
- Repro mínimo en `code/verilator/repro-cg-transition.sv`. El día que Verilator los soporte, se borra el `ifndef` y nada más

Cobertura funcional

Las perillas: cuándo un bin cuenta como cubierto

```
coverpoint op_set {
   option.at_least = 10;                  // un bin cuenta recien con 10 hits
   bins single[] = {[add_op:xor_op]};
}
coverpoint A { option.auto_bin_max = 8; } // 8 casilleros automaticos, no 64
```

- `option.at_least` es la que más importa: por defecto vale **1**, así que un bin que pasó **una sola vez** ya figura cubierto. El caso borde que salió una vez en mil corridas no está verificado, y el 100 % dice que sí
- `option.auto_bin_max` acota los bins automáticos. Sin `{ }` y sin esta perilla, un coverpoint de 8 bits no da 256 casilleros: da **64**, repartidos por la herramienta y no por la spec
- `option.weight` cambia cuánto pesa un coverpoint en el total, y `type_option.merge_instances` suma todas las instancias en un solo número en vez de reportarlas por separado
- Verilator 5.052 respeta `auto_bin_max`, y `at_least` **sólo escrito en el coverpoint**: en el covergroup lo ignora sin avisar. Está en `docs/verilator.md`

Cobertura funcional

Leer el número

```
$ make u2/convencional
Coverage Summary:
  covergroup : 86.8% (66/76)
```

- 66 de 76 bins llenos, con **1000 randomizaciones** y **615 operaciones al bus** —el `no_op` y el `rst_op` se sortean y no se envían, y el testbench lo imprime justo antes del reporte de Verilator—. Las otras filas salen `0/0`: `--coverage-user` deja afuera la cobertura de código
- Los 10 que faltan son **un solo valor**: `all_ops.auto_5` y sus nueve cruces. Es `3'b110`, que el enum **no tiene** — Verilator reparte los bins automáticos por el tipo de base, `bit [2:0]`, y no por miembro del enum
- O sea que ninguna fila del plan quedó sin cubrir, y en Questa esto da 100 %. El número no es la meta: la pregunta que sirve es **cuál** bin falta, y eso sólo lo dice el reporte por bin, `obj_dir/top/coverage.dat`
- Correr → mirar qué falta → escribir el test dirigido → volver a correr. Eso se llama *coverage closure*, y es a lo que un verificador le dedica el día

Cobertura funcional

El plan de verificación, entero y en una tabla

#	Feature	Escenario	Estímulo	Chequeo	Medida
1	ALU	las seis operaciones	random	scoreboard	`coverpoint op_set`
2	ALU	operandos en `00` y en FF	`dist` sesgado a los bordes	scoreboard	cross `op_00_FF`
3	mult	producto máximo: FF × FF	caso dirigido	scoreboard, 16 bits	bin `mul_max`
4	reset	operar después de un reset	`rst_op` intercalado	scoreboard	bin `rst_op => op`
5	mult	una mult después de una de un ciclo	random	scoreboard	bin de transición
6	ALU	la misma operación dos veces seguidas	random	scoreboard	bin `[* 2]`

#	Feature	Escenario	Estímulo	Chequeo	Medida
7	sub	restar de menos: `A < B`, y el `ovf` sube	random	scoreboard, **dos salidas**	bin `hubo_borrow`
8	sub	`A == B`: el resultado es 0 y el `ovf` **no** sube	random	scoreboard, dos salidas	bin `sub_00`/`sub_FF`
9	ovf	`ovf` no se levanta para ninguna otra operación	random	assertion	`c_ovf`
10	protocolo	los operandos no se tocan con `start` arriba	random	assertion	`cover property`
11	protocolo	done llega, y antes de 5 ciclos	random	assertion	`cover property`
12	protocolo	no_op es la única que no contesta	random	assertion	`cover property`

- Las tres columnas de la derecha son **las tres partes del testbench** de esta sección. Lo que las ata es el plan, y hasta ahora no lo habías visto junto
- Las dos filas de la resta son las que obligan al scoreboard a mirar **dos salidas**: si sólo compara `result`, pasa en verde con el `ovf` clavado en cero
- Las **cuatro últimas** no las chequea ningún scoreboard: son reglas de `ovf` y de **protocolo**, y se chequean donde ocurren —las assertions
- Las doce filas van numeradas igual que en `docs/plan-de-verificacion.md`, que además dice en qué archivo vive cada una y trae la plantilla del capstone

Cobertura funcional

Resumen de la unidad

- **Cobertura de código ≠ cobertura funcional.** La primera mide qué RTL se ejecutó; la segunda, qué escenarios de la spec pasaron
- Un `covergroup` es un **tipo**: hay que instanciarlo con `new()` y **alguien tiene que llamar a** `sample()`. Sin eso el reporte da 0 % sin una advertencia
- Los **bins** son los casilleros. `[]` reparte uno por valor; sin corchetes, todo el rango entra en uno solo
- `ignore_bins` **es documentación ejecutable**: dice en voz alta qué no se cubre y por qué. Un bin imposible que dejás puesto te clava el número abajo
- Un `cross` es el producto cartesiano y crece rapidísimo: sin acotarlo, la cobertura no cierra nunca
- `option.at_least` **vale 1 por defecto**: un bin que pasó una sola vez ya figura cubierto, y un caso raro con un hit no está verificado
- El número no es la meta. La meta es el **plan**; la cobertura sólo dice cuánto del plan se cumplió

Interfaces y BFM

Primero: las señales dejan de estar sueltas

```
interface vtalu_bfm;
   // brings in operation_t. e.g. (no_op  = 3'b000)
   import vtalu_pkg::*;

   byte unsigned        A;
   byte unsigned        B;
   bit                  clk;
   bit                  reset_n;
   wire          [ 2:0] op;
   bit                  start;
   wire                 done;
   wire          [15:0] result;
   wire                 ovf;
   operation_t          op_set;

   assign op = op_set;

   // The BFM is a first step towards modularity
   // It takes care of driving every low-level signal
   // and keeps the protocol in one single place

   // DUT clock
   initial begin
      clk = 0;
      forever begin
         #10;
         clk = ~clk;
      end
   end
```

- Una `interface` es un **bundle** de señales con nombre propio. Nueve cables que antes estaban declarados en el `top` ahora viven juntos
- El reloj se genera adentro: la interface no es un cable, es un modelo del bus
- Conectar el DUT pasa a ser `.A(bfm.A)`, `.clk(bfm.clk)`… y una señal nueva se declara **una vez**: los módulos que la usan la ven aparecer sin tocar sus puertos
- Lo que acá no hace falta y en un RTL ajeno vas a ver: el `modport`, que es la misma interface con las direcciones puestas para un lado del cable

Interfaces y BFM

Después: el protocolo se esconde en una task

```
// The protocol handling can now be changed in one single place.
// Fix it here once, and the fix propagates through the rest of the code
task send_op(input byte iA, input byte iB, input operation_t iop,
             output shortint alu_result);

   op_set = iop;

   if (iop == rst_op) begin
      @(posedge clk);
      reset_n = 1'b0;
      start = 1'b0;
      @(posedge clk);
      #1;
      reset_n = 1'b1;
   end else begin
      @(negedge clk);
      A = iA;
      B = iB;
      start = 1'b1;
      if (iop == no_op) begin
         @(posedge clk);
         #1;
         start = 1'b0;
      end else begin
         do @(negedge clk); while (done == 0);
         start = 1'b0;
      end
   end

endtask : send_op
```

- `send_op(A, B, op, result)` traduce *"hacé una suma"* al meneo de señales que el DUT espera. Eso es un **Bus Functional Model**
- Adentro está toda la letra chica de la spec: el pulso de `reset_n`, el `no_op` que no espera `done`, y el `do ... while (done == 0)` de las demás
- De acá en adelante **nadie mueve `bfm.start` a mano**. El que quiere una operación la pide

Interfaces y BFM

El top, con la BFM en el medio

- El testbench sigue partido en las mismas tres piezas —tester, scoreboard y cobertura—, y ninguna cambió de nombre
- Lo que cambió es **cómo se conectan al DUT**: ya no hay señales sueltas viajando, hay una sola interface que las tres comparten

```
module top;
   vtalu_bfm bfm ();
   // Tester: generates the stimulus
   tester tester_i (bfm);
   // Coverage: provides functional coverage
   coverage coverage_i (bfm);
   // ScoreBoard: checks the results
   scoreboard scoreboard_i (bfm);

   vtalu DUT (
       .A(bfm.A),
       .B(bfm.B),
       .op(bfm.op),
       .clk(bfm.clk),
       .reset_n(bfm.reset_n),
       .start(bfm.start),
       .done(bfm.done),
       .ovf(bfm.ovf),
       .result(bfm.result)
   );
endmodule : top
```

Interfaces y BFM

Scoreboard: el mismo chequeo, ahora hablando con la BFM

```
module scoreboard (
    vtalu_bfm bfm
);
   import vtalu_pkg::*;

   always @(posedge bfm.done) begin
      shortint predicted_result;
      bit      predicted_ovf;
      #1;
      case (bfm.op_set)
         add_op: predicted_result = bfm.A + bfm.B;
         sub_op: predicted_result = bfm.A - bfm.B;
         and_op: predicted_result = bfm.A & bfm.B;
         xor_op: predicted_result = bfm.A ^ bfm.B;
         mul_op: predicted_result = bfm.A * bfm.B;
      endcase  // case (op_set)

      // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
      // output, neither the addition nor the multiplication can overflow.
      predicted_ovf = (bfm.op_set == sub_op) && (bfm.A < bfm.B);

      if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
         if (predicted_result != bfm.result || predicted_ovf != bfm.ovf)
            $error(
                "FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
                bfm.A,
                bfm.B,
                bfm.op_set.name(),
                bfm.result,
                bfm.ovf
            );

   end
endmodule : scoreboard
```

- Es el `scoreboard` del testbench convencional: predice y compara con la misma lógica
- Lo único que cambió es de dónde saca las señales: antes eran variables del mismo módulo, ahora son `bfm.A`, `bfm.B`, `bfm.op_set`
- Y por eso ahora es un **módulo aparte**: puede vivir en su propio archivo porque no comparte nada con el tester salvo la interface

Interfaces y BFM

Tester: pide operaciones, no mueve cables

```
initial begin
   byte unsigned iA;
   byte unsigned iB;
   operation_t   op_set;
   shortint      result;

   bfm.reset_alu();
   repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set, result);
   end : random_loop
   $finish;
end  // initial begin
```

- El tester ya no toca `start` ni espera `done`: llama a `bfm.send_op(...)` y se olvida del protocolo
- El handshake quedó **escrito una sola vez**, adentro de la BFM. Si mañana el DUT pide dos ciclos de `start`, se toca un archivo
- Ésa es la definición de *Bus Functional Model*: la tarea traduce *"hacé una suma"* al meneo de señales que el DUT espera

Interfaces y BFM

La regla del protocolo, escrita en el idioma del simulador

```
// INMEDIATA: una sentencia. Chequea el presente, donde el hilo pasa.
assert (op_set != no_op || done == 0)
   else $error("no_op no deberia levantar done");

// CONCURRENTE: una declaracion con reloj. Se enchufa una vez y
// chequea CADA flanco de la simulacion, sola, para siempre.
a_operandos_estables :
assert property (@(negedge clk) start |=> $stable(A) && $stable(B));
```

- La regla —*mientras* `start` *está arriba, los operandos no se tocan*— ya está escrita **dos veces**: en prosa en la spec, y encapsulada adentro de `send_op()` acá
- Una **assertion** es esa misma frase por tercera vez, pero **ejecutable**: no la respeta, la **chequea**
- Y la chequea donde ocurre —en la interface, en el flanco exacto— y no al final, comparando resultados

Interfaces y BFM

Dónde se chequea cada cosa

- El **scoreboard** del testbench convencional chequea el **resultado**: `A op B` contra lo que devolvió el DUT. Es la mitad que este curso desarrolla hasta el final
- Una **assertion** chequea el **protocolo**: quién se mueve, cuándo, y por cuántos ciclos. Es la otra mitad
- Un DUT que devuelve bien el `result` pero baja `done` un ciclo antes **pasa todos los tests** de los próximos seis días. Nadie lo está mirando
- Se ve entero en **la sección de assertions**, con el mismo VTALU y el mismo `run.sh`

Interfaces y BFM

El agujero que deja la BFM: el temporizado

```
// 1. Sample AT the edge. The DUT always_ff updates d_out with a
//    nonblocking assignment, which lands after this initial already ran:
//    you read the OLD value.
@(posedge clk);
en_el_flanco = d_out;

// 2. The same edge, one delta later. The nonblocking assignment landed
//    already and you read the NEW value. That "#1" is visible nowhere and
//    changes the result.
@(posedge clk);
#1;
un_delta_despues = d_out;

// 3. The opposite edge: the trick the whole course BFM uses. It works,
//    but it asks whoever reads it to know why.
@(negedge clk);
en_el_flanco_opuesto = d_out;
```

- La BFM encapsuló **el protocolo**. Lo que **no** encapsuló es *cuándo* se maneja y *cuándo* se muestrea: eso sigue decidido task por task
- Tres formas de leer el mismo dato: `0`, `11` y `11`. Dos valores, pero **tres mecanismos**, y el del medio depende de un `#1` que no se ve en ningún lado
- El `@(negedge clk)` de nuestra BFM es la tercera. Anda — y le pide al que la lee que sepa por qué

Interfaces y BFM

Por qué el flanco solo no alcanza: las regiones



Tres formas de leer el mismo dato, con el DUT calculando `d_in + 1 = 11`:

1	`@(posedge clk);`	leído en Active, antes del NBA	`0`
2	`@(posedge clk); #1;`	el NBA ya aterrizó	`11`
3	`@(negedge clk);`	el flanco opuesto, medio ciclo después	`11`

El `#1` del medio funciona, y funciona por accidente: no dice qué carrera evita.

- Un flanco no es un instante indivisible: adentro hay **regiones**, en orden fijo
- El `<=` del `always_ff` **aterriza en NBA**: el que lee en *Active* lee el viejo
- El `#1` cruza el NBA, el flanco opuesto lo cruza sobrado, y SVA no necesita ninguno de los dos: muestrea en *preponed*, antes de todo

Interfaces y BFM

El clocking block: el flanco y el delta, declarados una vez

```
interface reg_bfm (input bit clk);
   byte unsigned d_in, d_out;

   // The only two timing lines in the testbench:
   //   input  #1step  samples the STABLE value right before the edge, which is
   //                  what the hardware sees. Never what the nonblocking
   //                  assignment just wrote on that same edge.
   //   output #0      drives ON the edge, in the nonblocking region, so the DUT
   //                  does not see it until the next edge.
   clocking cb @(posedge clk);
      default input #1step output #0;
      output d_in;
      input  d_out;
   endclocking

   modport tb (clocking cb);
endinterface : reg_bfm
```

- `input #1step` — muestreá el valor **estable justo antes** del flanco, que es el que ve el hardware. Nunca el que la no bloqueante acaba de escribir
- `output #0` — manejá **en** el flanco, con las no bloqueantes, así el DUT no lo ve hasta el flanco siguiente
- Manejar pasa a ser `cb.d_in <= valor`; muestrear, leer `cb.d_out`; esperar, `@(cb)`. **Ningún `#1` en ninguna task**
- Y el dato leído es el mismo en cualquier momento del ciclo: la carrera se cerró por declaración, no por costumbre

Interfaces y BFM

¿Son obligatorios? No — y conviene saber por qué

La pregunta	La respuesta
¿Se puede escribir un TB correcto **sin** clocking blocks?	**Sí.** Con NBA en el driver y disciplina de scheduler, no hay race
¿Qué conviene que use **un equipo**?	**Usalos** en interfaces síncronas — es lo que dice el style guide de lowRISC/OpenTitan

- No son de UVM: son de **SystemVerilog**. UVM no los pide ni los conoce
- **Dave Rich** —la referencia de Verification Academy— sostiene desde 2014 que son *opcionales*: lo que evita la race es entender el scheduler, no el clocking block
- Las dos posturas contestan preguntas distintas: una es **corrección del lenguaje**, la otra es **política de equipo**. Las dos son correctas

Interfaces y BFM

El precio: dos nombres para el mismo cable

```
// The two reads, at the SAME instant, on the SAME wire.
por_cb = bfm.cb.d_out;
crudo  = bfm.d_out;

$display("at the same instant, on the same wire:");
$display("  bfm.cb.d_out (what the clocking block sampled) : %0d", por_cb);
$display("  bfm.d_out    (the live wire)                   : %0d", crudo);
$display("");
$display("They differ by one cycle, and no warning says so. A monitor that");
$display("reads through the clocking block and a scoreboard that reads the");
$display("raw wire do not fail every time: they fail when the data changes.");
```

- Con la señal adentro de un clocking block hay **dos** formas de leerla: `bfm.d_out` es el cable en vivo, `bfm.cb.d_out` es lo que se muestreó `#1step` antes del flanco
- En una señal que cambia cada ciclo **difieren en un ciclo**: el ejemplo imprime `3` y `4`, en el mismo instante y sobre el mismo cable
- Nadie avisa. Compila, corre, y el scoreboard falla **sólo cuando el dato cambia**
- La regla: si una señal entró al clocking block, **todo el protocolo la lee por ahí** — y se espera `@(bfm.cb)`, no `@(posedge bfm.clk)`

Interfaces y BFM

Resumen de la unidad

- Una `interface` es un **bundle de señales con nombre**: agregar un cable deja de ser tocar seis port lists
- Un **BFM** es la interface **más el protocolo**: `send_op(A, B, op, result)` traduce *"hacé una suma"* al meneo de señales
- Toda la letra chica de la spec —el pulso de reset, esperar `done`, el `no_op` que no responde— queda escrita **una sola vez**
- Desde acá **nadie mueve `bfm.start` a mano**. El que quiere una operación, la pide
- El tester, el scoreboard y la cobertura pasan a ser **módulos aparte**, cada uno en su archivo, conectados por la BFM
- Y esto no se abandona: el driver del agent sigue llamando a **esta misma task**
- La regla que el BFM **cumple** se puede además **chequear**, con una assertion que vive en esta misma interface. Ésa es la sección de assertions
- Y el `clocking block` cierra el último agujero: el flanco y el delta dejan de decidirse task por task y se declaran **una vez**

Repaso · Día 1

1 de 7 · La pregunta del día

Mil operaciones al azar, el scoreboard no reportó ningún error y el log termina en `PASS`. ¿Qué te falta para decir que el DUT está verificado?

a. Nada: mil operaciones sin un error es un DUT verificado

b. Correr más semillas hasta que la cobertura de código llegue al 100 %

c. Saber qué habría dicho el log con un bug adentro: correrlo con el DUT mutado ✓

d. Reemplazar el scoreboard por assertions, que chequean el protocolo en el flanco exacto y no al final

Verlo fallar — el `PASS` de un scoreboard que nunca vio un error no dice nada: pudo no haber comparado. `VTALU_BUG=1` da vuelta un bit del resultado y el testbench tiene que fallar; `make mutante` lo exige para los tres testbenches de los días 1 y 2.

Repaso · Día 1

2 de 7 · La spec de la ALU

Mientras la VTALU está ejecutando una operación, ¿qué tienen que hacer `start` y los operandos?

a `start` baja enseguida: es un pulso de arranque, y el DUT ya latcheó todo

b Da igual: el DUT los registra en el primer flanco

c Los operandos tienen que cambiar en cada ciclo

d `start` en 1 y los operandos quietos hasta que sube `done` ✓

Estables hasta `done` — es el protocolo del DUT, y es exactamente el motivo por el que existe el BFM: encapsular esa regla en un solo lugar para que ningún test se la olvide. El primer distractor describe un protocolo real —arranque por pulso, operandos latcheados— que este DUT no tiene.

Repaso · Día 1

3 de 7 · Cobertura funcional

El test corre 1000 operaciones al azar y la cobertura de *código* da 100 %. ¿Qué te dice eso sobre la verificación?

a Que el DUT está verificado y el plan de verificación se puede cerrar

b Que el testbench no tiene bugs

c Que faltan pocos escenarios: el 100 % ya recorrió el diseño entero

d Muy poco: mide el RTL que se ejecutó, no la spec ✓

Muy poco — la cobertura de código mide el DUT; la funcional mide la spec. Una feature que el diseñador nunca escribió da 100 % de líneas y 0 % de lo que importa, y el reporte no te lo va a decir.

Repaso · Día 1

4 de 7 · covergroup

Declarás un `covergroup`, le hacés `new()`, corrés mil operaciones y el reporte da 0 %. ¿Qué es lo primero que hay que mirar?

- **a** Que nadie esté llamando a `sample()` ✓
- **b** Que los bins estén mal definidos y no matcheen ningún valor
- **c** Que el DUT no esté respondiendo
- **d** Que falten `ignore_bins`

El `sample()` — el covergroup no se muestrea solo: alguien tiene que llamarlo, en el flanco o cuando llega una transacción. Sin esa llamada el código compila, corre, y el reporte da 0 sin una sola advertencia.

Repaso · Día 1

5 de 7 · Interfaces y BFM

¿Qué gana el testbench cuando el protocolo se mueve a un BFM?

- **a** Deja de hablar en señales y pasa a hablar en operaciones ✓
- **b** Simula más rápido: una tarea del BFM le cuesta menos al simulador que mover cables
- **c** Se puede sintetizar el testbench
- **d** Se ahorra tener que declarar un `clk`

Deja de hablar en señales — el BFM traduce *una operación* a *un handshake de señales*. El tester no vuelve a tocar un cable; el scoreboard y el coverage siguen leyéndolos, por la interface, hasta que un monitor les mande transactions. Es el primer paso hacia UVM.

Repaso · Día 1

6 de 7 · El plan de verificación

En el plan de verificación, ¿qué va en la columna *Medida*?

a Cuánto tarda el escenario en correr, para poder estimar la regresión

b El nombre del archivo del testbench que cubre esa fila

c Qué bin se llena cuando el escenario pasa ✓

d Cuántas veces hay que correr el test para darlo por cubierto

Qué bin se llena — las cinco columnas son *Feature*, *Escenario*, *Estímulo*, *Chequeo* y *Medida*, y ésta es la que más cuesta: obliga a decidir **antes** qué se va a contar. Si queda vacía, nadie se va a enterar de que el escenario nunca pasó — un caso que el random no tocó y que no tiene bin es indistinguible de uno que pasó mil veces.

Repaso · Día 1

7 de 7 · clocking block

¿Hacen falta los clocking blocks para escribir un testbench UVM sin races?

a Sí: sin clocking block, driver y DUT compiten siempre en el mismo flanco

b Sí, y además son parte de la librería UVM

c No: alcanza con NBA en el driver y disciplina de scheduler ✓

d No: los reemplaza el `uvm_driver`, que ya muestrea en la región correcta

No hacen falta, y conviene usarlos igual — son de **SystemVerilog**, no de UVM, y ningún `uvm_driver` muestrea por vos. Lo que evita la race es entender el scheduler: un driver que maneja con `<=` contra un DUT que registra con `<=` ya es determinista. El clocking block no reemplaza ese entendimiento, lo **encapsula** — y ahí es donde paga: agents reutilizables, VIP, gate-level y protocolos con setup/hold en la spec.

Ejercicio · Día 1 · 1 de 2

Una operación nueva, de punta a punta

```
$ cd code/ejercicios/d1 && bash run.sh
```

- La VTALU tiene un **opcode libre**, `3'b110`. Enseñale a desplazar, y enseñale también al TB que lo tiene que verificar
- Se toca en **tres** lugares: el RTL, el estímulo con su chequeo, y **la medida** — el bin que hoy no existe
- El `run.sh` falla hasta que imprime `EXERCISE OK` **y el reporte cierra con 77 bins cubiertos**; la solución está al lado

Ejercicio · Día 1 · 2 de 2

Las ondas: cuando el log no alcanza

```
$ cd code/ejercicios/d1b && bash run.sh
```

- La corrida aborta con **una sola línea**: `FAILED: A: e5 B: 0 op: mul_op result: fe01 ovf: 0`. `e5 * 00` es `0`, no `fe01`. El scoreboard tiene razón, y el DUT también: el que se equivocó es el **handshake** de la BFM
- Ahí se termina lo que el log te da. El `run.sh` deja **`ondas.vcd`** al lado: `gtkwave ondas.vcd`
- **Etapa 1:** dos tiempos que sólo están en el visor, en `respuesta.txt`. **Etapa 2:** la línea de la BFM que los explica
- Es el único ejercicio del curso que **no se resuelve leyendo el log**

Seis de cada siete, otra vez

Cierre del día

- La regresión que mandó el bug a fabricar dijo `PASS`. Hoy tenés tres formas de saber si ese `PASS` valía algo, y las tres son columnas de una tabla
- **¿Qué había que probar?** Un plan: una fila por escenario, escrito antes que el testbench, desde la spec
- **¿Quién dijo que estuvo bien?** Un scoreboard que predice y compara — y que viste fallar con un bug adentro
- **¿Cuánto del plan pasó?** Un `covergroup` con un bin por fila, y el reporte por bin, no el porcentaje
- El tester ya pide operaciones en vez de mover cables: eso fue la BFM. Lo que todavía no se puede es **cambiar el estímulo sin editar el archivo**. Mañana es eso

Ayer quedó...

Dónde dejamos el testbench

- Un testbench que **anda**: manda estímulo, chequea el resultado y mide cobertura funcional. El plan de verificación dice qué fila cierra cada cosa
- La BFM se llevó las señales adentro de una `interface`, y el testbench pasó a llamar `send_op()` en vez de mover cables
- Y quedó un problema abierto: **cambiar el estímulo es editar el archivo**. Un tester que sólo multiplica es copiar el que ya está y borrarle cinco líneas

Agenda

Día 2 · ≈ 4 h · unidad 3 · la OOP que UVM da por sabida

- Clases y extensiones
- Polimorfismo
- Variables y métodos estáticos
- Clases paramétricas
- El patrón factory
- Un testbench sin un solo módulo

Al final del día podés:

- **Extender** una clase y cambiarle el comportamiento **sin copiarla**, y decir qué hace `super`
- **Cambiar** la clase que usa un testbench **sin tocar** el código que la usa, con una factory
- **Escribir** el testbench del día 1 sin un solo módulo

Clases y extensiones

¿Por qué OOP en un testbench?

- El testbench convencional es un **módulo**: se elabora una vez, existe desde el `time 0` hasta el `$finish`, y siempre es el mismo
- Un estímulo no se parece en nada a eso: aparece, viaja por el testbench, se compara y se tira. Eso no es un módulo, es un **objeto**
- Módulos = *estructura*, se elaboran. Clases = *datos y comportamiento*, se crean y se destruyen mientras la simulación corre
- Y no es opcional: **UVM es una librería de clases**. `uvm_test`, `uvm_component`, `uvm_object`. Sin esto, el resto del curso no se puede leer

Clases y extensiones

Lo que una `struct` no puede hacer

```
typedef struct {
   int length;
   int width;
} rectangle_struct;

typedef struct {int side;} square_struct;

module top_struct;
   rectangle_struct rectangle_s;
   square_struct    square_s;
   initial begin
      rectangle_s.length = 50;
      rectangle_s.width = 20;
      $display("rectangle area: %0d", rectangle_s.length * rectangle_s.width);
      square_s.side = 50;
      $display("square area: %0d", square_s.side ** 2);
   end
endmodule
```

- La `struct` guarda datos y **nada más**: la cuenta del área hay que escribirla afuera, y de nuevo en cada lugar donde se necesite
- `square_struct` no tiene ninguna relación con `rectangle_struct`, aunque un cuadrado sea un caso particular de un rectángulo
- No se randomiza sola, no se compara sola, no se imprime sola

Clases y extensiones

La primera clase: datos, métodos y constructor

```
class rectangle;
   int length;
   int width;

   function new(int l, int w);
      length = l;
      width = w;
   endfunction

   function int area;
      return length * width;
   endfunction
endclass : rectangle

module top_rectangle;

   rectangle rectangle_h;

   initial begin

      rectangle_h = new(.l(50), .w(20));

      $display("rectangle area: %0d", rectangle_h.area());

   end
endmodule : top_rectangle
```

- La clase junta los **datos** (`length`, `width`) con **lo que se hace con ellos** (`area()`). El área se calcula en un solo lugar
- `new()` es el *constructor*: corre una vez, cuando se crea el objeto
- `.l(50)` es pasaje de argumentos **por nombre**. UVM lo usa todo el tiempo

Clases y extensiones

Handle y objeto no son la misma cosa

```
rectangle rectangle_h;              // 1. un HANDLE. Todavia no hay objeto: vale null
rectangle_h = new(.l(50), .w(20));  // 2. new() crea el OBJETO y devuelve su handle
```

- Declarar un handle **no crea nada**. Hasta el `new()` vale `null`, y usarlo ahí revienta en tiempo de simulación, no de compilación
- El handle se parece a un puntero, pero no admite aritmética: no hay `h + 1`
- La memoria de una `struct` la reserva el simulador apenas la ve; la de un objeto, recién en el `new()`
- Y no hay `free()`: cuando no queda ningún handle apuntándolo, el objeto se recolecta solo

Clases y extensiones

Extender: extends y super

```
class rectangle;
   // Data members
   int length;
   int width;

   function new(int l, int w);
      length = l;
      width = w;
   endfunction

   function int area();
      return length * width;
   endfunction
endclass

// Extending classes!
class square extends rectangle;
   // Override Constructor
   function new(int side);
      // super: the call to the parent method
      super.new(.l(side), .w(side));
   endfunction

endclass
```

- `square extends rectangle` hereda `length`, `width` y `area()` **sin copiar una línea**. Un cuadrado es un rectángulo con una restricción, y el código lo dice así
- El constructor del hijo llama a `super.new()`: la parte de arriba del objeto también hay que construirla, y va **primero**
- Extender no es copiar: si mañana `area()` cambia, cambia para los dos

Clases y extensiones

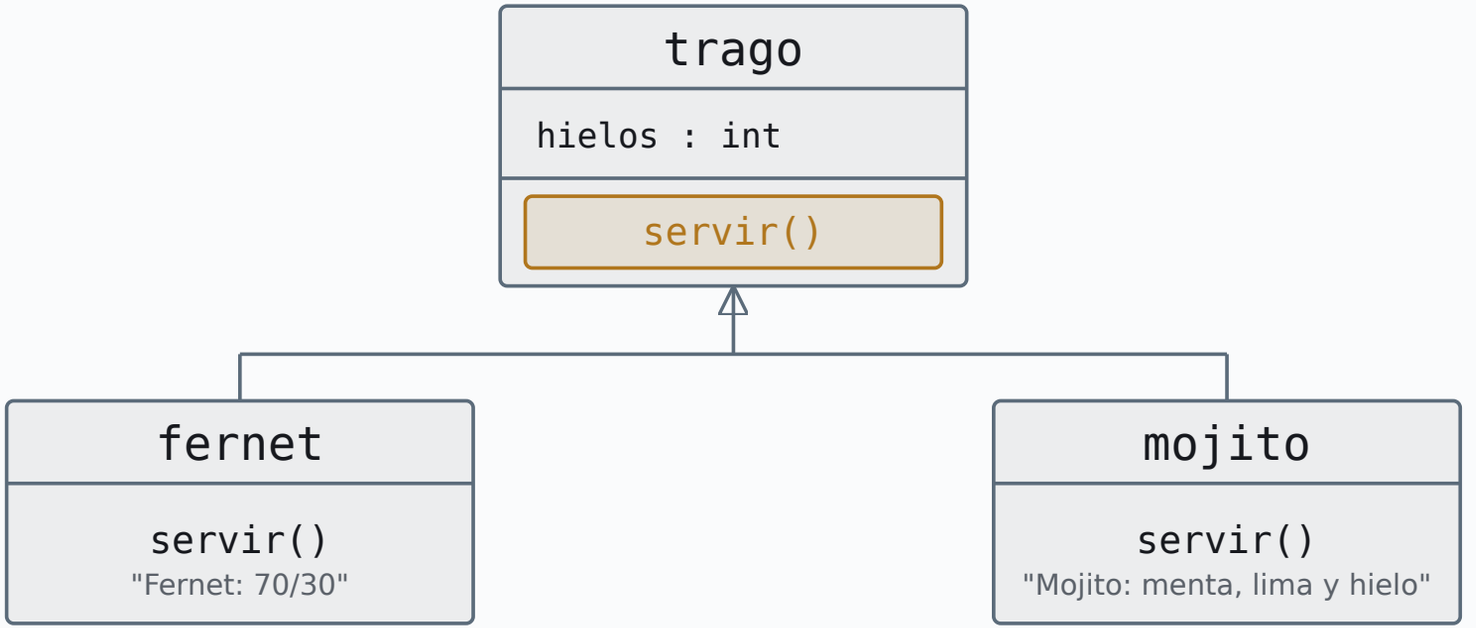
Dónde reaparece todo esto

Lo que acabás de ver	Cómo se llama en UVM
clase con datos y métodos	`uvm_object` — las *transactions* del día 5
clase que además vive en el árbol del TB	`uvm_component` — driver, monitor, scoreboard
`new()`	el constructor, con `name` (y `parent` si es un component)
`extends`	cada test, cada tester, cada agent del curso
`super.new()`	la primera línea obligatoria de todo constructor de UVM

- Falta una pieza: guardar un objeto `square` en una variable `rectangle`. Eso es la unidad que sigue

Polimorfismo

Una variable `trago`, un objeto `fernet`: ¿cuál `servir()` corre?




```
trago_h = fernet_h; trago_h.servir();
```

Sin virtual manda el tipo de la variable (trago, $fatal). Con virtual manda el objeto (fernet).

- Un `fernet` **es** un `trago`, así que guardarlo en una variable `trago` compila y es legal
- La pregunta es la otra: cuando llamás al método por esa variable, ¿SystemVerilog mira el **tipo de la variable** o el **tipo del objeto**?
- La respuesta por defecto es la que no querés, y la palabra que la cambia es una sola

Polimorfismo

Las tres clases: un `trago` que no se sabe servir

```
class trago;
   int hielos = -1;

   function new(int a);
      hielos = a;
   endfunction : new

   function void servir();
      $fatal(1, "A generic trago cannot be served: order a real one.");
   endfunction : servir

endclass : trago

class fernet extends trago;

   function new(int hielos);
      super.new(hielos);
   endfunction : new

   function void servir();
      $display("Fernet: 70/30, and the coke last");
   endfunction : servir

endclass : fernet
```

- `trago` define `servir()` con un `$fatal`: la clase base no sabe con qué se sirve, y lo dice fuerte
- `fernet` y `mojito` extienden `trago` y **redefinen** `servir()`, cada uno con el suyo
- `mojito` —que no entra en pantalla— es igual que `fernet`, con otro `$display`

Polimorfismo

Sin `virtual` manda la variable, y eso rompe

```
fernet_h = new(15);
fernet_h.servir();
$display("The fernet has %0d ice cubes", fernet_h.hielos);

mojito_h = new(1);
mojito_h.servir();
$display("The mojito has %0d ice cubes", mojito_h.hielos);

// The variable is trago, the object is fernet.
trago_h = fernet_h;
trago_h.servir();  // ** Fatal: A generic trago cannot be served
$display("The trago has %0d ice cubes", trago_h.hielos);

trago_h = mojito_h;
trago_h.servir();
$display("The trago has %0d ice cubes", trago_h.hielos);
```

- Las dos primeras llamadas andan: la variable `fernet_h` es de tipo `fernet` y el objeto también
- La tercera **explota**. `trago_h` apunta a un `fernet`, pero como el método no es virtual, SystemVerilog resuelve en **compilación** por el tipo de la variable, y llama al `servir()` de `trago`
- Se llama *static binding*: la decisión de qué código correr ya está tomada antes de que exista el objeto

Polimorfismo

Métodos virtuales: una palabra, y ahora manda el objeto

```
virtual function void servir();
   $fatal(1, "A generic trago cannot be served: order a real one.");
endfunction : servir
```

- Es la **única** diferencia entre `01-sin-virtual` y `02-virtual`: la keyword `virtual` en el método de la clase base
- Con eso la llamada se resuelve en **simulación**, mirando el objeto que hay adentro de la variable — *dynamic binding*
- `trago_h.servir()` ahora sirve un fernet o un mojito, sin que el que escribe esa línea sepa cuál le va a tocar
- Ésa es toda la idea: **el que usa el objeto deja de ser el que elige el tipo**

Polimorfismo

Clases abstractas: mover el error a la compilación

- El `$fatal` de `trago` se entera **tarde**: compila, corre, y explota en el medio de la simulación. En una regresión de la noche eso es una corrida perdida
- SystemVerilog permite declarar una **abstract class** —`virtual class`—: sólo sirve como base, **no se puede instanciar**
- Adentro puede declarar **pure virtual methods**: métodos sin cuerpo
- Extender la clase **obliga** a redefinirlos. Si no, es un **error de compilación**
- El error no desapareció: se mudó a un momento en el que sale barato

Polimorfismo

`pure virtual`: el mismo ejemplo, sin `$fatal` posible

```
// Abstract class: it only works as a base, it cannot be instantiated.
virtual class trago;
   int hielos = -1;

   function new(int a);
      hielos = a;
   endfunction : new

   // pure virtual: no body. Extending it forces you to redefine it.
   pure virtual function void servir();

endclass : trago
```

- `virtual class trago` ya no tiene un `servir()` que explote: **no tiene cuerpo**, y por eso no hay nada que pueda correr mal
- La línea comentada del `top` — `trago_h = new(3)` — no compila: una clase abstracta no se instancia
- Si a `mojito` le borrás el override, el compilador dice *"Class 'mojito' extends 'trago' but is missing implementation for 'servir'"* y ahí termina el asunto

Polimorfismo

Resumen de la unidad

- **Sin `virtual` manda la variable**, y se decide al compilar. **Con `virtual` manda el objeto**, y se resuelve en simulación. Una palabra, y cambia todo
- La idea de fondo, que vuelve en cada sección que sigue: **el que usa el objeto deja de ser el que elige el tipo**
- En SystemVerilog `virtual` está **sobrecargado**: virtual method, virtual class y virtual interface son tres cosas distintas — y en UVM vas a escribir las tres

Variables estáticas

La variable global que sí podés defender

- En un testbench siempre hay algo que es **uno solo**: el contador de errores, la lista de transacciones en vuelo, el handle a la configuración
- La respuesta fácil es una variable global, y es la que después nadie puede debuggear: no se sabe quién la escribió, ni desde dónde, ni cuándo
- `static` es la misma idea con apellido. La variable vive **en la clase**, no en el objeto, y para tocarla hay que nombrar la clase: `bandeja_de_fernet::vasos`
- Ese `::` es todo el argumento. El día que el valor sea raro, un grep te dice exactamente quién lo puso
- No es un tema suelto de OOP: la **factory** y el `uvm_config_db` son las dos variables estáticas más importantes de un testbench UVM

Variables estáticas

- `bandeja_de_fernet` no tiene métodos ni constructor: es una clase que existe **sólo** para ser el lugar donde vive la cola
- Nadie la instancia. Se la nombra con el operador `::`, que le dice al compilador *"buscá esta variable en el namespace de esa clase"*
- Y ahí está la ventaja sobre la global: `bandeja_de_fernet::vasos` dice dónde está declarada. Un `vasos` suelto, no

```
class bandeja_de_fernet;
   // Static queue
   static fernet vasos[$];

endclass : bandeja_de_fernet

module top;
   initial begin
      fernet fernet_h;
      fernet_h = new(2, "the one at table 7");
      // :: operator
      //----------------
      //   The variable in the bandeja_de_fernet class is reached by
      //   naming the class and putting the :: operator before the name
      //   That tells the compiler we want a static variable inside
      //   the namespace of the class
      bandeja_de_fernet::vasos.push_back(fernet_h);
      fernet_h = new(3, "the one at table 4");
      bandeja_de_fernet::vasos.push_back(fernet_h);
      fernet_h = new(15, "the one at the bar");

      // class::static_variable. You know right away where it is declared
      bandeja_de_fernet::vasos.push_back(fernet_h);
```

```
      $display("Fernets on the tray:");
      foreach (bandeja_de_fernet::vasos[i]) $display(bandeja_de_fernet::vasos[i].get_name());
   end
endmodule : top
```

Variables estáticas

- Hay **una sola** copia en memoria, sin importar cuántos objetos se instancien
- Y existe aunque no se instancie ninguno: la `static` no espera al `new()`

```
class pepe;
   static    int    cant;
   int    val = -1;

   function new(int a);
      // Static Variable
      cant++;
      val = a;
   endfunction : new

endclass : pepe

//---------------------------------------
module top_pepes;
   initial begin
      pepe pepe1_h, pepe2_h;

      pepe1_h = new(10);
      pepe2_h = new(254654);

      $display("PEPE1 VAL = %0d", pepe1_h.val);
      $display("PEPE2 VAL = %0d", pepe2_h.val);

      // There is only one copy of a static variable in memory,
      // no matter how many instances of the class exist
      $display("NUMBER OF PEPES = %0d", pepe::cant);

   end
endmodule : top_pepes
```

Variables estáticas

Métodos estáticos

- En los dos ejemplos anteriores la cola estaba a la vista, así que el día que se cambia por otra estructura hay que salir a corregir todo el testbench
- La versión que se defiende en un code review son dos palabras: la variable `protected`, y **métodos de acceso estáticos** que sean la única puerta
- Es lo mismo que hace UVM con la factory: nadie toca el diccionario de tipos, se le pide con `type_id::create()`

```
class bandeja_de_fernet;

   protected static fernet vasos[$];

   static function void bandeja_fernet(fernet l);
      vasos.push_back(l);
   endfunction : bandeja_fernet

   static function void lista_fernets();
      $display("Fernets on the tray:");
      foreach (vasos[i]) $display(vasos[i].get_name());
   endfunction : lista_fernets

endclass : bandeja_de_fernet

module top;

   initial begin
      fernet fernet_h;
```

```
      fernet_h = new(2, "the one at table 7");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      fernet_h = new(3, "the one at table 4");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      fernet_h = new(15, "the one at the bar");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      bandeja_de_fernet::lista_fernets();
   end

endmodule : top
```

Variables estáticas

Resumen de la unidad

- **Una sola copia** por más objetos que instancies, **existe aunque no instancies ninguno**, y para tocarla hay que nombrar la clase con `::`
- Ese `::` es todo el argumento: el día que el valor sea raro, un `grep` te dice quién lo puso. Pero `static` resuelve *dónde vive*, no *quién lo toca* — eso es `protected` más métodos de acceso estáticos
- Dónde reaparece: `uvm_config_db#(T)::set(...)` y `type_id::create()`, las dos estáticas más importantes de un testbench UVM

Clases paramétricas

No es una clase: es un molde

- La `bandeja_de_fernet` de la sección anterior sirve para fernets. Para mojitos hay que escribir otra igual, con una palabra cambiada
- Eso ya lo resolviste en RTL y no le decís OOP: una FIFO no se copia por cada ancho de dato, se parametriza. `#(.awidth(8), .dwidth(16))`
- Una clase paramétrica es lo mismo, con **tipos** en vez de números: `bandeja #(fernet)` y `bandeja #(mojito)`
- Y hay una consecuencia que sorprende: cada `#(...)` distinto es una **clase distinta**, generada en compilación. No comparten nada, ni siquiera lo `static`
- Por qué importa: cada `#(...)` que veas en el resto del curso es esto. `uvm_analysis_port #(T)`, `uvm_subscriber #(T)`, `uvm_sequence #(T)`

Clases paramétricas

- Antes de la sintaxis nueva, el parámetro que ya conocés: una RAM que no se copia por cada ancho, se instancia con `#(.awidth(8), .dwidth(16))`
- Cada instanciación con parámetros distintos produce **hardware distinto**, y nadie lo discute
- Las *parameterized class definitions* son eso mismo del lado del software, con una diferencia: el parámetro puede ser un **tipo**

```
module RAM #(
    awidth,
    dwidth
) (
    input wire [awidth-1:0] address,
    inout wire [dwidth-1:0] data,
    input we
);

   initial $display("awidth: %0d  dwidth %0d", awidth, dwidth);
   // code to implement RAM

   //...

endmodule  // RAM

module top;
   wire [ 7:0] address;
   wire [15:0] data;
   wire        write_enable;
   RAM #(
       .awidth(8),
       .dwidth(16)
```

```
    ) my_ram (
        address,
        data,
        write_enable
    );
endmodule  // top
```

Clases paramétricas

- Versión con métodos estáticos: la bandeja no se instancia, se le pide con `::`
- La cola es `static` y aun así hay **dos**: `bandeja#(fernet)` y `bandeja#(mojito)` son dos clases distintas, cada una con la suya

```
// Parameterized class
// the T (type) parameter is what says which
// kind of queue this is.
class bandeja #(
    type T
);

   protected static T vasos[$];

   static function void bandeja_trago(T l);
      vasos.push_back(l);
   endfunction : bandeja_trago

   static function void lista_tragos();
      $display("Drinks on the tray:");
      foreach (vasos[i]) $display(vasos[i].get_name());
   endfunction : lista_tragos

endclass : bandeja

module top;

   initial begin
      fernet fernet_h;
      mojito mojito_h;
      fernet_h = new(15, "the one at the bar");
      bandeja#(fernet)::bandeja_trago(fernet_h);
      fernet_h = new(15, "the one at table 4");
      bandeja#(fernet)::bandeja_trago(fernet_h);
```

```
      mojito_h = new(1, "the one at table 7");
      bandeja#(mojito)::bandeja_trago(mojito_h);
      mojito_h = new(1, "the one at the counter");
      bandeja#(mojito)::bandeja_trago(mojito_h);

      $display("-- Fernets --");
      bandeja#(fernet)::lista_tragos();
      $display("-- Mojitos --");
      bandeja#(mojito)::lista_tragos();
   end

endmodule : top
```

Clases paramétricas

Variables con parámetros

- Acá la bandeja se instancia como cualquier objeto y hay que tener el handle para usarla — deja de verse desde todo el testbench
- Hace exactamente lo mismo que la versión anterior. Lo que cambia es el alcance, y ésa es la decisión: `static` **cuando de verdad hay uno solo, instanciado cuando puede haber más de uno**

```
module top;

   // Since bandeja is not static, it has to be instantiated
   // the parameter is a variable type
   // Using a type that does not have the get_name()
   // method is a syntax error
   fernet fernet_h;
   mojito mojito_h;

   bandeja #(fernet) bandeja_de_fernet;
   bandeja #(mojito) bandeja_de_mojito;

   initial begin
      bandeja_de_fernet = new();
      fernet_h = new(15, "the one at the bar");
      bandeja_de_fernet.bandeja_trago(fernet_h);
      fernet_h = new(15, "the one at table 4");
      bandeja_de_fernet.bandeja_trago(fernet_h);

      bandeja_de_mojito = new();
      mojito_h = new(1, "the one at table 7");
      bandeja_de_mojito.bandeja_trago(mojito_h);
```

```
      mojito_h = new(1, "the one on the sidewalk");
      bandeja_de_mojito.bandeja_trago(mojito_h);

      $display("-- Fernets --");
      bandeja_de_fernet.lista_tragos();
      $display("-- Mojitos --");
      bandeja_de_mojito.lista_tragos();
   end

endmodule : top
```

Clases paramétricas

Resumen de la unidad

- Una clase paramétrica **no es una clase**: es la receta. Y cada `#(...)` distinto es una **clase distinta**, que no comparte nada con las otras — **ni siquiera lo** `static`
- El límite: se resuelve **al compilar**, así que el tipo tiene que conocerse ahí. Elegir en runtime es la sección que sigue
- Dónde reaparece: `uvm_analysis_port #(T)`, `uvm_subscriber #(T)`, `uvm_sequence #(T)` — cada `#(...)` del resto del curso es esto

El patrón factory

¿Quién decide el tipo?

- Cada `new()` que escribís es una decisión **hardcodeada**: esa línea siempre va a construir esa clase, y para cambiarla hay que editarla y recompilar
- En un testbench eso no escala. Querés el mismo `env` mandando sumas hoy y multiplicaciones mañana, elegido por línea de comandos
- Las clases paramétricas de la unidad anterior tampoco alcanzan: el `#(...)` se resuelve al compilar, y acá hay que decidir **en runtime**
- La factory es la respuesta: en vez de construir vos, le **pedís** el objeto a alguien que sabe qué tipo entregar
- Es el patrón más visible de UVM, y con él se entienden `type_id::create()` y `set_type_override()`, que van a estar en cada archivo del resto del curso

El patrón factory

- Un patrón de diseño, no una feature del lenguaje: una clase cuyo único trabajo es **construir objetos por vos**
- Le pedís un objeto describiendo cuál querés —un string, un enum— y te lo devuelve ya construido, del subtipo que corresponda
- Antes de la solución, el problema. Mirá qué pasa cuando el `new()` está escrito en el lugar donde se usa el objeto:

```
// ILLUSTRATION -- not compiled. In the previous example, new() gets called every
// time an object is needed: building a new set of drinks means editing the source.
bandeja_de_fernet = new();
fernet_h = new(15, "the one at the bar");
bandeja_de_fernet.bandeja_trago(fernet_h);
fernet_h = new(15, "the one at table 4");
bandeja_de_fernet.bandeja_trago(fernet_h);

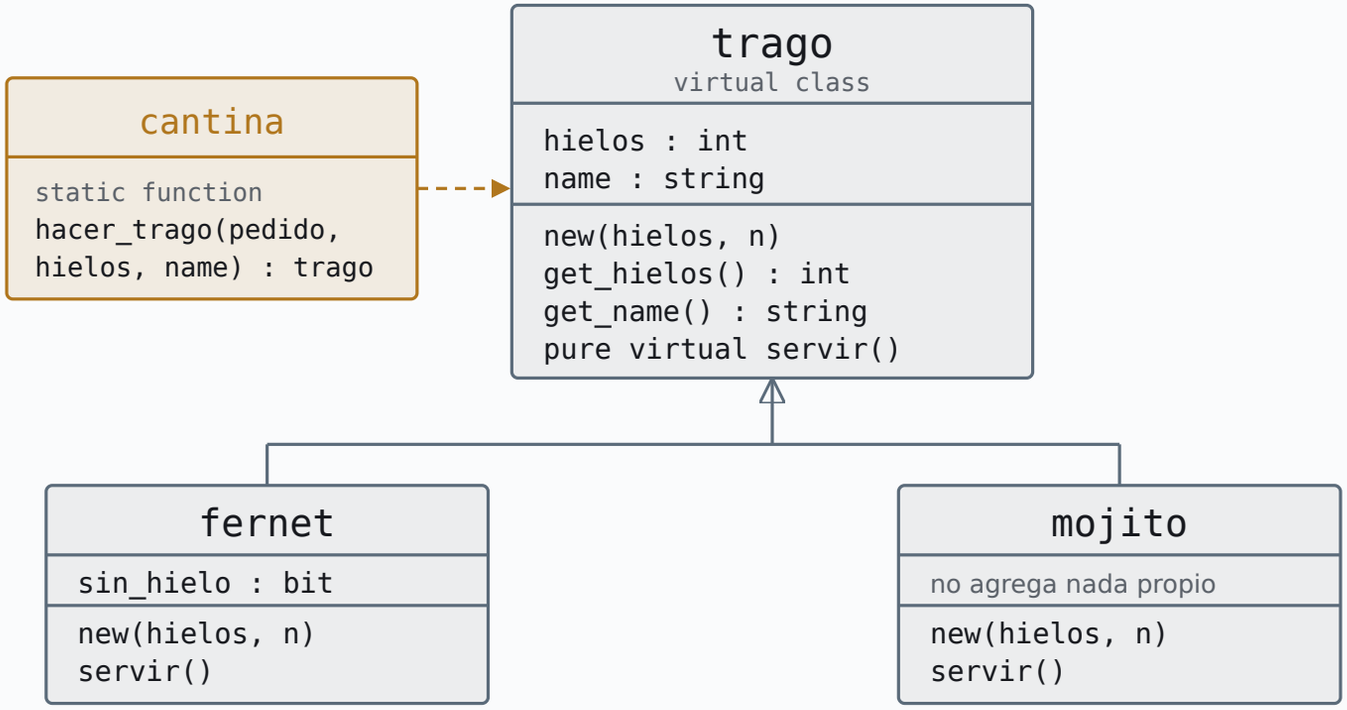
bandeja_de_mojito = new();
mojito_h = new(1, "the one at table 7");
bandeja_de_mojito.bandeja_trago(mojito_h);

mojito_h = new(1, "the one on the sidewalk");
bandeja_de_mojito.bandeja_trago(mojito_h);

// Consider the case where a set of random drinks has to be read from a file
// dynamically.
//    --> Doing that with NON hardcoded code is impossible
//        You would need a script that generates the source, recompile it...
```

La idea es crear datos de tipos aleatorios sin cambiar el código.

El patrón factory



`$cast(fernet_h, trago_h)` — el handle sale como trago; sin castear no ves sin_hielo.

- **La factory** es el método de arriba: recibe un argumento y devuelve un objeto del tipo que ese argumento pide
- **El polimorfismo** es lo que la hace posible: `fernet` y `mojito` extienden `trago`, así que los dos entran en una variable `trago`
- Sin lo segundo no hay dónde poner lo que devuelve lo primero. Son las dos mitades de la misma herramienta

El patrón factory

- `virtual` alcanza mientras uses métodos que la clase base ya declara. Pero `fernet` agrega un campo propio — `sin_hielo` — que `trago` no conoce
- Para llegar a él hay que bajar el handle a su tipo real, y eso es `$cast`: convierte la variable del segundo argumento a la clase del primero
- Chequea **en runtime** y devuelve 0 si no da, así que nunca va solo: siempre adentro de un `if` con su `$fatal`

```
//************************************************************************
// Part of the factory Class!
//************************************************************************
class cantina;

   // Static method that builds the drinks
   static function trago hacer_trago(string pedido, int hielos, string name);
      mojito nuevo_mojito;
      fernet nuevo_fernet;
      case (pedido)
         "fernet": begin
            nuevo_fernet = new(hielos, name);
            return nuevo_fernet;
         end

         "mojito": begin
            nuevo_mojito = new(hielos, name);
            return nuevo_mojito;
         end

         default: begin
            $fatal(1, {"No such drink: ", pedido});
            return null;
```

```
        end

      endcase  // case (pedido)

   endfunction : hacer_trago

endclass : cantina
```

- Y el `$cast` en uso, en el `top`: la factory siempre devuelve un `trago`, y para llegar a `sin_hielo` hay que bajarlo a `fernet`

```
    // Using the factory!
    trago_h = cantina::hacer_trago("fernet", 15, "the one at the bar");
    trago_h.servir();

    // Reaching a member of fernet (derived from trago) through a trago variable
    // requires casting it to fernet.
    if (!$cast(fernet_h, trago_h)) $fatal(1, "Could not cast trago_h to fernet_h");

    if (fernet_h.sin_hielo) $display("And on top of that it comes warm!");
    bandeja#(fernet)::bandeja_trago(fernet_h);

    if (!$cast(fernet_h, cantina::hacer_trago("fernet", 2, "the one at table 4")))
       $fatal(1, "Could not cast the cantina drink to fernet_h");
```

El patrón factory

Python Style

```
class Trago:
    """Factory class that builds a Trago."""

    @staticmethod
    def factory(kind):
        """Factory Method."""
        if kind == "CubaLibre":
            return CubaLibre()
        if kind == "Fernet":
            return Fernet()
        if kind == "Mojito":
            return Mojito()
        if kind == "Whiscola":
            return Whiscola()
        raise ValueError("No such trago: " + kind)
```

- La misma fábrica, en un lenguaje que no tiene nada que ver con hardware: un método estático con un `if` por tipo y un `raise` para lo que no existe
- Y la misma molestia: agregar un trago obliga a **editar la fábrica**
- El archivo completo está en `code/u3/factory/factory.py` y se corre con `python3`: genera veinte tragos al azar y los sirve a todos igual

El patrón factory

Resumen de la unidad

- Cada `new()` es una **decisión hardcodeada**. La factory la mueve a runtime: en vez de construir, **pedís** el objeto a alguien que sabe qué tipo entregar
- **No funciona sin polimorfismo** —hacen falta alguien que fabrique y una variable base donde guardarlo— y el **`$cast` devuelve 0** en vez de abortar, así que nunca va solo: siempre adentro de un `if` con su `$fatal`
- Dónde reaparece: `` `uvm_component_utils `` **registra la clase sola** —el `case` que acabás de escribir a mano— y `+UVM_TESTNAME` la elige desde la línea de comandos

Un testbench sin un solo módulo

El mismo testbench, sin un solo módulo

- Esta sección **no agrega ninguna funcionalidad**: verifica lo mismo que el testbench de interfaces y BFM
- Lo que cambia es de qué está hecho. Los tres módulos —tester, scoreboard, coverage— pasan a ser tres **clases**, y el `initial` pasa a ser un método
- Aparece la pieza que los une: una clase `testbench` que instancia a las otras, les pasa la BFM y lanza sus threads. Nadie la instancia por vos
- Es la **última versión sin UVM**, y por eso vale mirarla con atención: todo lo que hace a mano esta clase, desde los tests lo va a hacer la librería
- La pregunta con la que hay que salir de la sección: *¿qué parte reemplaza a qué?*

Un testbench sin un solo módulo

- Un testbench de módulos funciona hasta que hay que reusarlo: para cambiar el estímulo hay que editar el módulo, y para tener dos versiones hay que copiarlo
- Con clases, cambiar el estímulo es **extender una clase**, y las dos versiones conviven. Eso es todo lo que compra la OOP acá

El VTALU TB, pieza por pieza

- top: Instancia la clase testbench
- testbench: Top-Level Class
- tester: Genera los estímulos
- scoreboard: Chequea que la VTALU está funcionando
- coverage: Captura la información de cobertura funcional



Los tres reciben la misma BFM por el constructor, como una `virtual vtalu_bfm`.
Desde los tests, este árbol lo arma UVM: `build_phase`, `config_db` y un `run_phase` por pieza.

Un testbench sin un solo módulo

El top: lo único que sigue siendo un módulo

- El `top` sigue siendo un módulo, y va a seguir siéndolo hasta el final del curso: las clases no se elaboran, alguien tiene que existir antes del tiempo 0
- Importa el package con las clases, instancia el DUT y la BFM, y declara el handle al `testbench`
- Adentro del `initial`, dos líneas: `new(bfm)` y `execute()`

```
module top;

    // Class definitions and shared resources are kept in a package
    import vtalu_pkg::*;
    // The macros are defined in common too. UVM does this as well
    `include "vtalu_macros.svh"

    // The DUT and the BFM get instantiated
    vtalu DUT (
        .A(bfm.A),
        .B(bfm.B),
        .op(bfm.op_set),
        .clk(bfm.clk),
        .reset_n(bfm.reset_n),
        .start(bfm.start),
        .done(bfm.done),
        .ovf(bfm.ovf),
        .result(bfm.result)
    );

    vtalu_bfm bfm ();
```

```
    // A variable to hold our TB
    testbench testbench_h;

    initial begin
        // The testbench object gets built and launched
        testbench_h = new(bfm);
        testbench_h.execute();
    end

endmodule : top
```

Un testbench sin un solo módulo

La clase `testbench`: quien arma y arranca

- Hay **un** objeto arriba de todo que instancia a los demás, les pasa lo que necesitan y lanza sus threads. Todo testbench OOP tiene uno
- Acá lo escribimos a mano: tres `new()` y un `fork ... join_none`
- Desde el `env` esa clase se va a llamar `uvm_env`, los `new()` van a ser `type_id::create()` en el `build_phase`, y el `fork` lo va a hacer UVM sola

```
class testbench;

   // virtual: the object-world equivalent of a module's port list. It tells
   // the compiler that this variable is going to receive a handle to an
   // interface at some point in the future. See the slide in the TB-in-objects section.
   virtual vtalu_bfm bfm;

   tester    tester_h;
   coverage  coverage_h;
   scoreboard scoreboard_h;

   function new(virtual vtalu_bfm b);
      bfm = b;
   endfunction : new

   task execute();
      tester_h = new(bfm);
      coverage_h = new(bfm);
      scoreboard_h = new(bfm);

      // One thread per object. join_none: execute() returns and the three go on.
      fork
```

```
            tester_h.execute();
            coverage_h.execute();
            scoreboard_h.execute();
        join_none
    endtask : execute
endclass : testbench
```

Un testbench sin un solo módulo

La clase `tester`: el estímulo, sin cables

- Manda mil operaciones al azar, con el mismo sesgo a los bordes del testbench convencional: el objetivo sigue siendo llenar los bins del covergroup
- Contra la versión modular cambian dos cosas y nada más:
  - la BFM llega por una **variable** —una *virtual interface*— en vez de por una port list
  - el `initial` es ahora un método, `execute()`, que alguien tiene que llamar

```
// The initial block is replaced by an execute() task
task execute();
   byte unsigned     iA;
   byte unsigned     iB;
   shortint unsigned result;
   operation_t       op_set;
   bfm.reset_alu();
   op_set = rst_op;
   iA = get_data();
   iB = get_data();
   bfm.send_op(iA, iB, op_set, result);
   op_set = mul_op;
   bfm.send_op(iA, iB, op_set, result);
   bfm.send_op(iA, iB, op_set, result);
   op_set = rst_op;
   bfm.send_op(iA, iB, op_set, result);
   repeat (1000) begin : random_loop
```

```
        op_set = get_op();
        iA = get_data();
        iB = get_data();
        bfm.send_op(iA, iB, op_set, result);
        $display("%2h %6s %2h = %4h", iA, op_set.name(), iB, result);
      end : random_loop
      $finish;
    endtask : execute
```

Un testbench sin un solo módulo

El scoreboard: el `always` se volvió una task

- La lista de sensibilidad no existe en una clase. Lo que la reemplaza es un `forever` con la espera adentro — misma semántica, escrita como código secuencial

```
// The always @(posedge done) of the modular version, as a forever loop: an
// object has no sensitivity list, it has a task that blocks.
task execute();
   shortint predicted_result;
   bit      predicted_ovf;
   forever begin : self_checker
      @(posedge bfm.done) #1;
      case (bfm.op_set)
         add_op: predicted_result = bfm.A + bfm.B;
         sub_op: predicted_result = bfm.A - bfm.B;
         and_op: predicted_result = bfm.A & bfm.B;
         xor_op: predicted_result = bfm.A ^ bfm.B;
         mul_op: predicted_result = bfm.A * bfm.B;
      endcase  // case (op_set)

      // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
      // output, neither the addition nor the multiplication can overflow.
      predicted_ovf = (bfm.op_set == sub_op) && (bfm.A < bfm.B);

      if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
         if (predicted_result != bfm.result || predicted_ovf != bfm.ovf)
            $error(
                "FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
                bfm.A,
                bfm.B,
                bfm.op_set.name(),
                bfm.result,
```

```
                bfm.ovf
            );

    end : self_checker
endtask : execute
```

Un testbench sin un solo módulo

La clase coverage: el covergroup adentro de un objeto

```
// In a class, we don't need to declare variables to hold the covergroups, but we do need
// to call a constructor to create them.
 function new (virtual vtalu_bfm b);
   op_cov = new();
   zeros_or_ones_on_ops = new();
   bfm = b;
 endfunction : new

 task execute();
    forever begin  : sampling_block
       @(negedge bfm.clk);
       A = bfm.A;
       B = bfm.B;
       op_set = bfm.op_set;
       op_cov.sample();
       zeros_or_ones_on_ops.sample();
    end : sampling_block
 endtask : execute
```

- El covergroup es **el mismo** del testbench convencional, línea por línea. Lo que cambió es dónde vive
- En un módulo, declarar el covergroup alcanzaba. En una clase hay que construirlo: el `new()` de la clase hace `op_cov = new()`
- El `execute()` es el `always @(negedge clk)` de antes, escrito como `forever`: copia las señales de la BFM y llama a `sample()`

Un testbench sin un solo módulo

El mapa del día 3: qué reemplaza a qué

Acá, a mano	Desde los tests
`testbench.execute()` con tres `new()`	el `build_phase` del `env`
pasar la BFM por el constructor	`uvm_config_db`
`fork ... join_none`	UVM corre un `run_phase` por componente
`$finish` adentro del tester	objections
elegir el tester editando el código	`set_type_override` de la factory
`$display` que se pone y se borra	`` `uvm_info `` y el techo de verbosidad

- Éste es el testbench **completo**, sin UVM y sin nada que falte. Funciona
- Todo lo que viene en el día 3 reemplaza una fila de esta tabla, **y nada más**
- Volvemos a esta slide al cerrar el día, para tacharlas

Un testbench sin un solo módulo

Resumen de la unidad

- El testbench del día 1 **sin un solo módulo**: las mismas tres piezas, más la clase que las une — `testbench` —, que es el antepasado directo del `uvm_env`
- El `top` sigue siendo un **módulo** y lo va a ser hasta el final, y la BFM llega por una **virtual interface**: es la única forma que tiene una clase de tocar señales
- Es la **última versión sin UVM**. De acá en adelante, cada pieza que escribimos a mano tiene una clase de la librería que la reemplaza — `execute()` va a ser el `run_phase`

Repaso · Día 2

1 de 8 · Handle y objeto

`rectangle rectangle_h;` — después de esa línea, ¿cuántos objetos hay?

a Ninguno: `rectangle_h` vale `null` hasta que alguien llame a `new()` ✓

b Uno, con `length` y `width` en 0

c Uno a medio construir: los campos existen pero el constructor no corrió

d Depende de si la clase tiene constructor

Ninguno — declarar un handle no reserva nada. Ahí está la diferencia con una `struct`, que el simulador reserva apenas la ve. Y usar el handle antes del `new()` no falla al compilar: revienta en medio de la simulación.

Repaso · Día 2

2 de 8 · Polimorfismo

Una variable de tipo `trago` guarda un objeto `fernet`. Si `servir()` no es `virtual`, ¿qué se ejecuta?

a El `servir()` de `fernet`

b Error de compilación

c El `servir()` de `trago` ✓

d Los dos, primero el de la clase base

El de `trago` — sin `virtual`, SystemVerilog mira el **tipo de la variable**, no el del objeto. Es literal lo que imprime `code/u3/polimorfismo/01-sin-virtual`: *"A generic trago cannot be served"*.

Repaso · Día 2

3 de 8 · Clases abstractas

¿Qué gano usando una clase abstracta con métodos `pure virtual` en vez de una clase base cuyo método hace `$fatal`?

a Nada, es cuestión de estilo

b Que el error pasa de la simulación al compilador ✓

c Que se puede instanciar la clase base

d Que el simulador puede elegir el override más específico en tiempo de ejecución

El error pasa al compilador — con `$fatal` te enterás a mitad de la simulación de que faltaba un override. Con `pure virtual` no compila. Atajar antes siempre es más barato.

Repaso · Día 2

4 de 8 · Variables estáticas

Instancio 10 objetos de una clase que tiene una variable `static`. ¿Cuántas copias hay en memoria?

a 10, una por objeto

b 0 hasta que alguien la escribe: la memoria se reserva en el primer acceso

c 1, compartida por todas las instancias ✓

d Depende del simulador

Una sola — y existe aunque no instancies ningún objeto. Eso es lo que la hace útil para datos globales del TB, y lo que la hace peligrosa si la dejás pública.

Repaso · Día 2

5 de 8 · Métodos estáticos

¿Por qué conviene declarar `protected` la variable estática y exponerla con métodos estáticos?

- **a** Para poder cambiar la estructura de datos sin tocar a quien la usa ✓
- **b** Porque una `static` sin `protected` se reserva una vez por instancia
- **c** Para que ocupe menos memoria
- **d** Porque UVM lo exige

Para poder cambiarla después — si la cola está a la vista, el día que la cambiás por otra estructura tenés que salir a corregir todos los lugares que la tocaban. Encapsular es poder cambiar de opinión.

Repaso · Día 2

6 de 8 · Clases paramétricas

`bandeja#(fernet)` y `bandeja#(mojito)` tienen adentro una queue `static`. ¿Comparten la queue?

a Sí, `static` es una sola para todos

b Sí: el parámetro cambia el tipo de los métodos, no el almacenamiento estático

c Depende de si se instancian o no

d No: cada especialización es una clase distinta, con su propia queue ✓

No la comparten — SystemVerilog genera **una clase por combinación de parámetros**. `static` es único dentro de cada una de esas clases, no entre todas. UVM se apoya en esto todo el tiempo.

Repaso · Día 2

7 de 8 · El patrón factory

¿Qué problema resuelve el patrón factory?

a Crear objetos más rápido

b Decidir en tiempo de ejecución qué subtipo construir ✓

c Evitar tener que declarar clases

d Centralizar los `new()` en una sola clase, para poder contarlos y liberarlos

Decidir el subtipo en runtime — sin hardcodear el `new`: le pedís un objeto a la fábrica y ella decide cuál. Es la pieza que después te va a dejar cambiar el estímulo de un test entero sin tocar el código del `env`.

Repaso · Día 2

8 de 8 · $cast

¿Cuándo tiene éxito un `$cast(destino, origen)`?

a. Siempre: convierte cualquier clase en cualquier otra

b. Sólo entre clases sin herencia

c. Cuando las dos clases tienen los mismos campos, aunque no estén emparentadas

d. Sólo si el objeto de `origen` es de la clase de `destino` o de una derivada ✓

Sólo si el objeto lo permite — `$cast` chequea **en runtime** y devuelve 0 si no da. Por eso la factory de UVM es más cómoda que la `cantina` de la sección: el `type_id::create()` devuelve el tipo correcto y te ahorra el casteo.

Ejercicio · Día 2

Un tester que sólo multiplica

```
$ cd code/ejercicios/d2 && bash run.sh
```

- Extendé `tester`, redefiní `get_op()`, y que el testbench lo instancie
- Va a seguir mandando operaciones al azar. **Ahí empieza el ejercicio**
- Pista: polimorfismo

Ayer quedó...

Dónde dejamos el testbench

- El testbench del día 1 **sin un solo módulo**: tres clases, una que las une, y un `top` que sólo instancia el DUT y la BFM
- La BFM entra a las clases por una **virtual interface**, que es la única forma que tiene una clase de tocar señales
- Y la factory ya está escrita **a mano**: un `case` que hay que editar cada vez que aparece un tipo nuevo. Hoy aparece la que no hay que editar

Agenda

Día 3 · ≈ 4 h 30 · unidad 4 · entra UVM

- Tests
- Components y fases
- El env: estructura y estímulo
- Reporting: verbosidad y *actions*

Al final del día podés:

- **Explicar** por qué una simulación de UVM termina en `t=0` si nadie levanta un objection
- **Decir** por qué `build_phase` corre de arriba hacia abajo y `connect_phase` al revés
- **Hacer** que un scoreboard que falla diga **algo más** que "falló"

Tests

Compilar una vez, elegir el test por línea de comandos

- El testbench en objetos tiene el estímulo **adentro**: cambiar de test es cambiar código y volver a compilar
- Mil tests a cinco minutos de compilación cada uno son **5000 minutos**: tres días y medio de máquina para no correr nada nuevo
- UVM da vuelta el orden: el testbench se compila **una vez**, y el test que corre se elige al arrancar la simulación

```
$ ./obj_dir/top/sim +UVM_TESTNAME=random_test
$ ./obj_dir/top/sim +UVM_TESTNAME=add_test     # el mismo binario
```

- Eso es lo que se cobra en esta unidad, y lo que se paga con la ceremonia que viene: registro en la factory, constructor con firma fija y fases

Tests

El top: instanciar, publicar la interface, arrancar

```
initial begin
   // null and "*": from the root and visible to the whole tree. See the
   // uvm_config_db slide.
   uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "bfm", bfm);

   // UVM reads +UVM_TESTNAME, and builds THAT test with the factory.
   run_test();
```

- El `top` sigue siendo un **módulo**: instancia el BFM y el DUT igual que antes. Lo único nuevo son estas dos líneas
- `run_test()` sin argumento significa *"el que diga `+UVM_TESTNAME`"*. Con un argumento — `run_test("random_test")` — ése pasa a ser el **default**, y `+UVM_TESTNAME` lo sigue pisando desde la línea de comandos
- Antes de `run_test()` no hay ningún objeto de UVM vivo: por eso el `set` va acá

Tests

uvm_config_db: el buzón del testbench

- Base de datos global y **tipada**: el `#(...)` es parte de la llave
- Cuatro argumentos: *ámbito* (`cntxt`, `inst_name`), *nombre* y *dato*
- El **ámbito** es una ruta jerárquica, y ahí está toda la gracia

```
// ILLUSTRATION -- not compiled. The two halves of the config_db, side by side:
// the set() lives in top.sv and the get() in tb_classes/driver.svh.
// -- IN THE TOP -- top is a module, not a uvm_component --------------------
//   cntxt = null : the scope starts at the UVM root
//   inst  = "*"  : visible to the whole component tree
uvm_config_db #(virtual vtalu_bfm)::set(null, "*", "bfm", bfm);

// -- IN A COMPONENT -- always inside build_phase ----------------------------
//   cntxt = this : the scope is the hierarchical path of THIS component
//   inst  = ""   : with no suffix, the path is exactly the component's
if (!uvm_config_db #(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
  `uvm_fatal("DRIVER", "Failed to get BFM")

// The 3rd argument ("bfm") is the NAME of the datum inside the database.
// If the string in the set and the one in the get are not identical, this still
// compiles and blows up at run time with a null pointer.
```

Tests

Un test en tres partes: 1 · registrarlo

```
class random_test extends uvm_test;
   `uvm_component_utils(random_test);

   virtual vtalu_bfm bfm;
```

- `random_test` extiende `uvm_test`, que extiende `uvm_component`: es un nodo del árbol, no un objeto suelto
- La macro `` `uvm_component_utils `` lo **inscribe en la factory**. Sin esa línea, `+UVM_TESTNAME=random_test` no encuentra nada y UVM aborta
- El handle a la virtual interface se **declara** acá y se **llena** en el `build_phase`: son dos momentos distintos

Tests

2 · el constructor y el `build_phase`

```
function new(string name, uvm_component parent);
   super.new(name, parent);
endfunction : new

// The get goes in build_phase and not in the constructor: when the
// constructor runs, the tree does not exist yet.
function void build_phase(uvm_phase phase);
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
      `uvm_fatal("RANDOM TEST", "Failed to get BFM")
endfunction : build_phase
```

- El constructor de un `uvm_component` tiene **firma fija**: `name` y `parent`, en ese orden, y `super.new(name, parent)` como primera línea
- El `get` del config_db va en `build_phase`, **no** en el constructor: cuando corre el constructor el árbol todavía se está armando
- `if (!get(...)) uvm_fatal`: el `get` devuelve un bit. Ignorarlo deja el handle en `null` y la falla aparece cien líneas después, en otro archivo

Tests

3 · el run_phase: acá pasa la simulación

```
task run_phase(uvm_phase phase);
   random_tester random_tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   phase.raise_objection(this);

   random_tester_h = new(bfm);
   coverage_h = new(bfm);
   scoreboard_h = new(bfm);

   fork
      coverage_h.execute();
      scoreboard_h.execute();
   join_none

   random_tester_h.execute();

   phase.drop_objection(this);
endtask : run_phase
```

- Es el mismo cuerpo que el `execute()` del testbench en objetos: tester, coverage y scoreboard, con los dos observadores en `fork ... join_none`
- Lo que cambia es **quién lo llama**: antes lo llamaba el módulo `top`, ahora lo llama UVM cuando le toca a la fase de run
- Las dos líneas nuevas son el `raise` y el `drop` de la objection, que son las que deciden cuándo termina todo

Tests

Objections: quién decide cuándo termina

```
task run_phase(uvm_phase phase);
   phase.raise_objection(this);    // "todavia tengo trabajo"
   ...                             // el estimulo
   phase.drop_objection(this);     // "listo, por mi que termine"
endtask
```

- Todos los `run_phase` corren **en paralelo**: ninguno puede decidir solo cuándo terminar. UVM cuenta, y la fase termina cuando cae la **última** objection
- Sin `raise_objection` la simulación se termina en el **tiempo 0**, y el test pasa con 0 errores sin haber mandado un estímulo
- Sin `drop_objection` no termina nunca: no hay error, la simulación sigue corriendo sola
- Los dos síntomas son **mudos**, y por eso existen dos plusargs: `+UVM_OBJECTION_TRACE` dice quién la levantó y quién la bajó, y `+UVM_TIMEOUT` le pone un techo a la corrida

Tests

El final que no es el final: `drain_time`

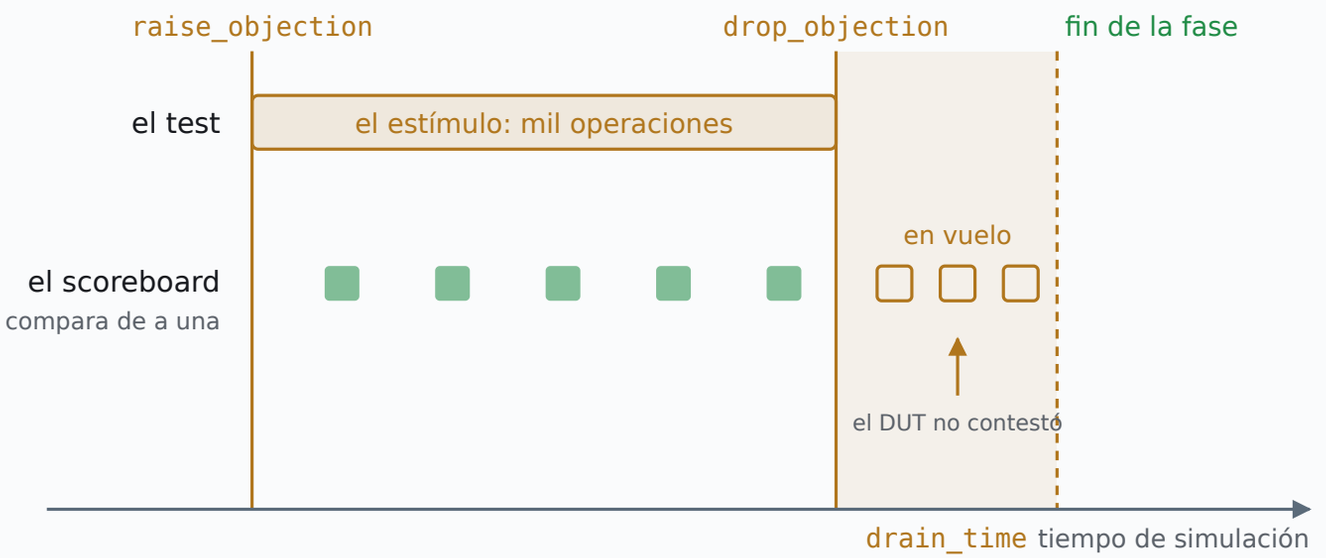
- La objection dice *"terminé de **mandar**"*. No dice *"terminé de **comparar**"*: entre las dos hay un scoreboard con una FIFO adentro y resultados que el DUT todavía no contestó
- Cuando cae la última objection la fase corta **en ese instante**: lo que quedó en vuelo no llega, nadie lo compara, y el resumen dice `UVM_ERROR : 0`

```
task run_phase(uvm_phase phase);
   phase.get_objection().set_drain_time(this, 200ns);  // un colchon tras el ultimo drop
   phase.raise_objection(this);
   ...                                                 // el estimulo
   phase.drop_objection(this);
endtask
```

- El `drain_time` es un número elegido a ojo. La versión con criterio es **`phase_ready_to_end()`**: UVM la llama cuando cayó la última objection, y el componente que todavía tiene trabajo levanta una más y la fase sigue
- Es **el** bug de la segunda semana, y es mudo: no hay error, no hay cuelgue, y las últimas transacciones nunca se compararon

Tests

La línea de tiempo, de punta a punta



La objection mide el estímulo, no el análisis: al caer la última, la fase corta en ese instante. Sin el colchón, esas tres no se comparan y el resumen dice `UVM_ERROR : 0`.

- `raise` y `drop` marcan **el estímulo**, y nada más que el estímulo
- Lo que quedó en vuelo no entra en esa cuenta: son las tres cajas vacías
- El `drain_time` es el tiempo que se regala para que esas respuestas lleguen
- Sin `raise` no hay nada entre las marcas; sin `drop`, la línea no termina nunca

Tests

El segundo test: lo mismo, con otro tester

```
task run_phase(uvm_phase phase);
   add_tester add_tester_h;
   coverage   coverage_h;
   scoreboard scoreboard_h;

   phase.raise_objection(this);

   add_tester_h = new(bfm);
   coverage_h = new(bfm);
   scoreboard_h = new(bfm);

   fork
      coverage_h.execute();
      scoreboard_h.execute();
   join_none

   add_tester_h.execute();
   phase.drop_objection(this);
endtask : run_phase
```

- `add_test` es `random_test` con **una línea distinta**: `add_tester` en lugar de `random_tester`. Todo lo demás se repite tal cual
- Esa repetición es real y es fea, y es la deuda que paga los components: cuando el testbench sea un árbol de components, el test va a construir un `env` y no va a tocar el estímulo
- Por ahora vale la pena que se vea copiada: es el problema que motiva lo que sigue

Tests

Cómo se corre, y qué imprime

```
set -e
. "$(dirname "${BASH_SOURCE[0]}")/../../verilator/common.sh"
vlt_uvm top --coverage-user -Wno-fatal -f dut.f -f tb.f
run_sim +UVM_TESTNAME=random_test
run_sim +UVM_TESTNAME=add_test
cov_report
```

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 46340: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 46ns
```

- Un solo `vlt_uvm`, dos `run_sim`: **una compilación, dos tests**. Eso es todo lo que promete la unidad, y acá está hecho
- `[RNTST] Running test random_test...` es UVM diciendo qué encontró en la factory con el nombre que le pasaste
- El *Report Summary* con `UVM_ERROR : 0` es el veredicto: reporting lo desarma y explica de dónde sale cada fila

Tests

Resumen de la unidad

- El testbench se compila **una vez** y el test se elige por línea de comandos: `+UVM_TESTNAME=random_test`. Ése es todo el negocio de la sección
- La cuenta que lo justifica: mil tests a cinco minutos de compilación cada uno son **tres días y medio** de esperar a que compile
- El `top` sigue siendo un módulo. Publica la BFM en el `uvm_config_db` **antes** de `run_test()`, porque antes de esa línea no hay un solo objeto de UVM vivo
- El **`uvm_config_db`** es una base de datos global y **tipada**: el `#(...)` es parte de la llave, y el ámbito es una **ruta jerárquica**
- Un test se registra con `` `uvm_component_utils ``. Sin esa línea, la factory no lo encuentra y UVM aborta
- La virtual interface se **declara** en la clase y se **llena** en el `build_phase`: son dos momentos distintos, y confundirlos deja un `null`
- La objection dice cuándo el test terminó de **mandar**, no cuándo el scoreboard terminó de **comparar**. Esa diferencia se paga con `drain_time`, o bien con `phase_ready_to_end()`

Components y fases

Un component es lo que está en el árbol, y al árbol lo recorre UVM

- Un testbench tiene tres cosas distintas: la **estructura** —qué partes hay y cómo se conectan—, las **secuencias** —qué comandos, en qué orden— y los **datos**. Esta unidad es la primera; los datos son el día 5 y las secuencias el día 6
- En los tests el test hacía `new()` de tres objetos sueltos y los llamaba él mismo. Eso no es una estructura: es una función larga
- Un `uvm_component` es un **nodo con nombre y con padre**. UVM arma el árbol completo y después lo recorre entero llamando **fases**, en un orden que garantiza
- Convertir una clase en component son cuatro pasos, siempre los mismos:

    1. extender `uvm_component` o una hija suya
    2. registrarla con `` `uvm_component_utils() ``
    3. el constructor mínimo: `(string name, uvm_component parent)`
    4. override de las fases que le hagan falta

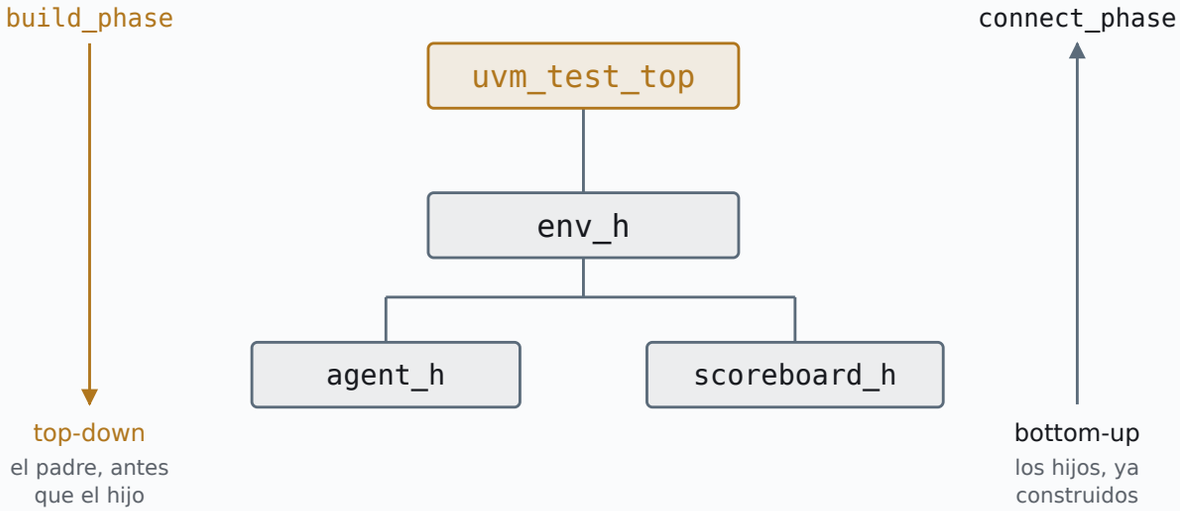
Components y fases

Intro UVM phases

- Todos los uvm components tienen estos phase methods por herencia
- UVM crea el TB y va llamando estos métodos en orden
- Las fases son *métodos*: se overridean como cualquier otro método virtual. Si además conviene llamar a `super`, es tema de la slide que sigue
- *function void build_phase(uvm_phase phase):* UVM crea el TB (top-down). Los componentes UVM se instancian en este método. Si tratás de instanciarlos en otro lado es error
- *function void connect_phase(uvm_phase phase):* Conexión de componentes
- *function void end_of_elaboration_phase(uvm_phase phase):* UVM lo llama una vez que creó y conectó todos los componentes
- *task run_phase(uvm_phase phase):* UVM ejecuta esta task en su propio thread. Todos los run_phase se ejecutan simultáneamente
- *function void report_phase(uvm_phase phase):* Se ejecuta cuando se termina la última objection del test. Muestra Resultados

Components y fases

El árbol, dos veces: una baja y la otra sube



Y adentro de esa bajada, el orden real:

`test.build_phase()` → `create("env_h")` → `env.new()` → `env.build_phase()`

El hijo se crea adentro del build del padre, y su propio build corre recién cuando el del padre terminó
Por eso instanciar un component afuera de `build_phase` llega tarde: el árbol ya se recorrió.

- `build_phase` baja porque el padre tiene que existir para crear a sus hijos
- `connect_phase` sube por lo contrario: el otro extremo ya tiene que estar
- El hijo se **crea** adentro del build del padre; su propio build corre después
- Por eso el `set()` del `config_db` va **antes** del `create()` que lo construye

Components y fases

¿Y el `super.build_phase()`?

```
// uvm_component.svh:2385-2393 — uvm-core 2020.3.1, con el return; elidido
function void uvm_component::build_phase(uvm_phase phase);
   build();            // -> apply_config_settings(): la config automatica
endfunction

function void uvm_component::connect_phase(uvm_phase phase);
   connect();          // -> return;  esta vacia, igual que las demas
endfunction
```

- `build_phase` es la **única** fase que hace algo en `uvm_component`: aplica los campos registrados con `` `uvm_field_* ``. El curso no usa esas macros
- Ojo, hay **dos `super` distintos**: ése, y el de una clase base **tuya**, que construye lo que vos escribiste. El segundo el curso **sí** lo llama
- Afuera: **ponelo siempre, salvo que puedas nombrar por qué en tu caso es un no-op**. De más no cuesta nada; de menos —un `` `uvm_field_* ``, o heredar de `uvm_agent`— el campo queda en su default y **nadie te avisa**
- Y el orden importa: lo que tenga que estar decidido *antes* de construir los hijos va antes del `super`. El `set_type_override` de `add_test` es eso

Components y fases

El cuadro completo: son nueve, no cinco

Fase	Para qué	Orden
`build_phase`	instanciar los componentes	**top-down**
`connect_phase`	conectar los ports	bottom-up
`end_of_elaboration_phase`	jerarquía lista, antes de simular	bottom-up
`start_of_simulation_phase`	último aviso antes del tiempo 0	bottom-up
`run_phase`	**task**: acá pasa la simulación	un thread c/u
`extract_phase`	juntar los datos de la corrida	bottom-up
`check_phase`	decidir si pasó o no	bottom-up
`report_phase`	imprimir el veredicto	bottom-up
`final_phase`	cerrar archivos y salir	**top-down**

- El curso usa cinco. Las otras cuatro existen, están vacías, y las vas a ver

Components y fases

Y adentro del `run_phase`, un cronograma



El nombre real lleva el sufijo: `pre_reset_phase`, `reset_phase`, `post_reset_phase`...
Cada una tiene su propio objection, y una fase runtime sin objection termina en cuanto arranca.

- Las doce son `task`, corren **en paralelo** con `run_phase`, y están para coordinar componentes de equipos distintos sin banderas a mano
- El curso usa `run_phase` y nada más — con un solo agent no hay nada que coordinar. Pero el `default_sequence` del día 6 se engancha a **`main_phase`**

Components y fases

El scoreboard, ahora como component

```
// The scoreboard of the TB in objects, now as a uvm_component. The four steps
// --extend, register, constructor, phases-- are in the slides of the components section.
class scoreboard extends uvm_component;
   `uvm_component_utils(scoreboard);

   virtual vtalu_bfm bfm;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   // Self-sufficient: the component asks the config_db for the BFM. Before, the
   // test passed it in through the constructor, and the test had to receive it
   // only to hand it out.
   function void build_phase(uvm_phase phase);
      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("SCOREBOARD", "Failed to get BFM")
   endfunction : build_phase
```

- Los cuatro pasos, en orden: extiende `uvm_component`, se registra, constructor `(name, parent)`, y override de `build_phase`
- El cambio de fondo no se ve en el diff: antes el **test** recibía la BFM y se la pasaba al scoreboard por el constructor. Ahora el scoreboard **se la pide solo** al config_db
- El `run_phase` es el `execute()` del testbench en objetos sin tocar una coma: el chequeo no cambió, cambió quién lo arranca

Components y fases

El test construye el árbol y se va

```
class random_test extends uvm_test;
   `uvm_component_utils(random_test);

   random_tester tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   // Components get instantiated in build_phase, never in the constructor.
   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   // No run_phase: the test's job ends once the tree is built.

endclass
```

- Los tres `new("nombre", this)` son lo que crea el árbol: `this` es el padre, y el string es el nombre que va a salir en `print_topology` y en cada mensaje
- Van en `build_phase` **y en ningún otro lado**. En el constructor todavía no hay jerarquía; después de `build_phase`, UVM ya siguió
- `random_test` **no tiene** `run_phase`: su trabajo terminó cuando el árbol quedó armado. Los tres hijos tienen el suyo, y corren en paralelo

Components y fases

El segundo test: hereda, y aun así copia

```
// add_test extends random_test, redeclares tester_h with another type and
// REWRITES the whole build_phase. That duplication is the problem the env section solves
// with the factory override.

class add_test extends random_test;
   `uvm_component_utils(add_test);

   add_tester tester_h;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction

endclass
```

- `add_test` extiende `random_test`, así que hereda `coverage_h` y `scoreboard_h`. Sólo redeclara `tester_h`, con otro tipo
- Pero el `build_phase` está **escrito de nuevo entero**, y las tres líneas son idénticas salvo una. Heredar la clase no alcanzó para heredar la estructura
- Y hay algo peor escondido: `add_tester tester_h;` **tapa** al `tester_h` de la clase base. Son dos variables distintas con el mismo nombre
- Es el mismo problema de los tests, disfrazado. El env lo resuelve de verdad: **una** estructura, y la factory sustituye la pieza que cambia

Components y fases

Resumen de la unidad

- Un `uvm_component` es un **nodo con nombre y con padre**. UVM arma el árbol con esos dos datos y después lo recorre solo
- Convertir una clase en component son **cuatro pasos, siempre los mismos**: extender, registrar con la macro, el constructor de dos argumentos, y las fases
- Las **fases** son métodos virtuales que UVM llama **en orden**: `build_phase` (top-down), `connect_phase`, `end_of_elaboration_phase`, `run_phase`, `report_phase`
- Los componentes se instancian **en el `build_phase` y en ningún otro lado**
- Todos los `run_phase` corren **en paralelo**, cada uno en su thread. Ninguno decide solo cuándo termina la simulación: eso son las **objections**
- Hay **dos `super.build_phase()`**: el de `uvm_component` y el de tu clase base. **Ponelo siempre, salvo que puedas nombrar por qué en tu caso es un no-op**

El env: estructura y estímulo

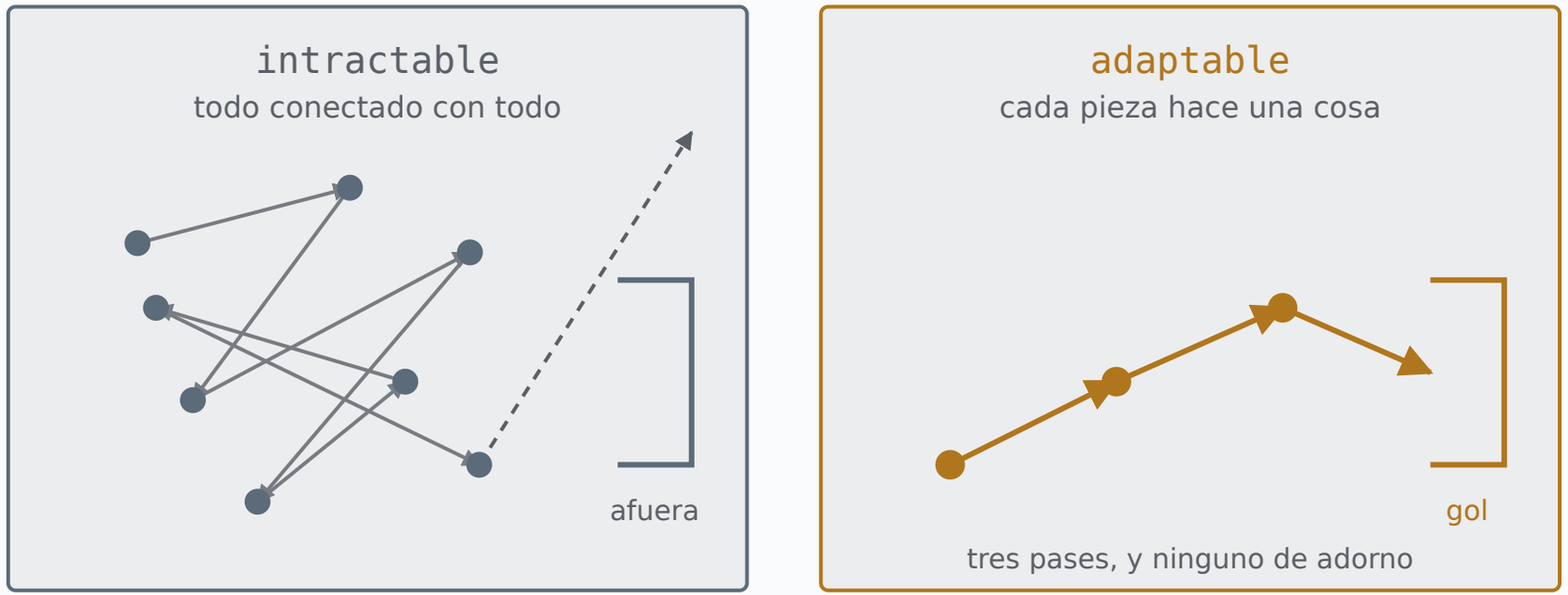
Dos cosas que cambian por motivos distintos

- Un testbench tiene **estructura** —qué componentes hay y cómo se conectan— y **estímulo** —qué se le manda al DUT—
- Cambian por motivos distintos y a ritmos distintos: la estructura cambia cuando cambia el diseño; el estímulo, cada vez que escribís un test
- Hasta los components están pegados: `add_test` es una copia de `random_test` con una línea cambiada. Diez tests son diez copias, y un cambio en la estructura son diez ediciones
- Esta sección los separa en dos clases: el **`uvm_env`** se queda con la estructura, el **`uvm_test`** con el estímulo
- Y la pieza que lo hace posible es la **factory**: el `env` pide un `base_tester` genérico, y cada test le dice a la factory cuál entregar

El env: estructura y estímulo

Intractable vs. adaptable

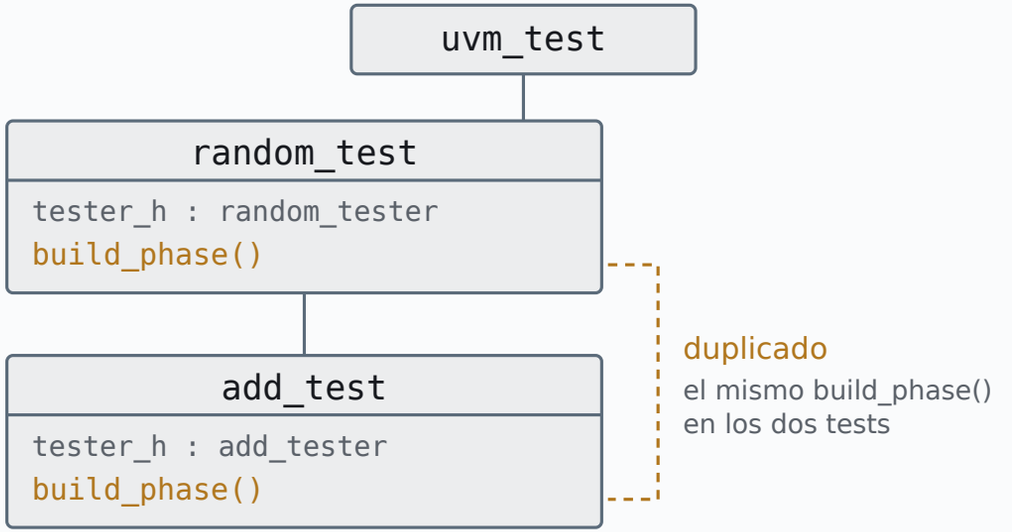
> *“Jugar al fútbol es muy simple, pero jugar un fútbol simple es lo más difícil que hay.” —* ***Johan Cruyff***



- *Intractable* es el código que sale rápido y después no se deja tocar. El criterio no es estético: **otra persona lo va a modificar**, y esa persona vas a ser vos dentro de un año, sin acordarte de nada
- Tres reglas para dar vuelta el TB del VTALU: **una clase, una cosa**; **nada hardcodeado**; **programar contra la interface**, no contra la implementación

El env: estructura y estímulo

El problema: dos tests, dos veces el mismo `build_phase()`



add_test hereda de random_test y le reescribe el build_phase entero: los tres new() son idénticos.

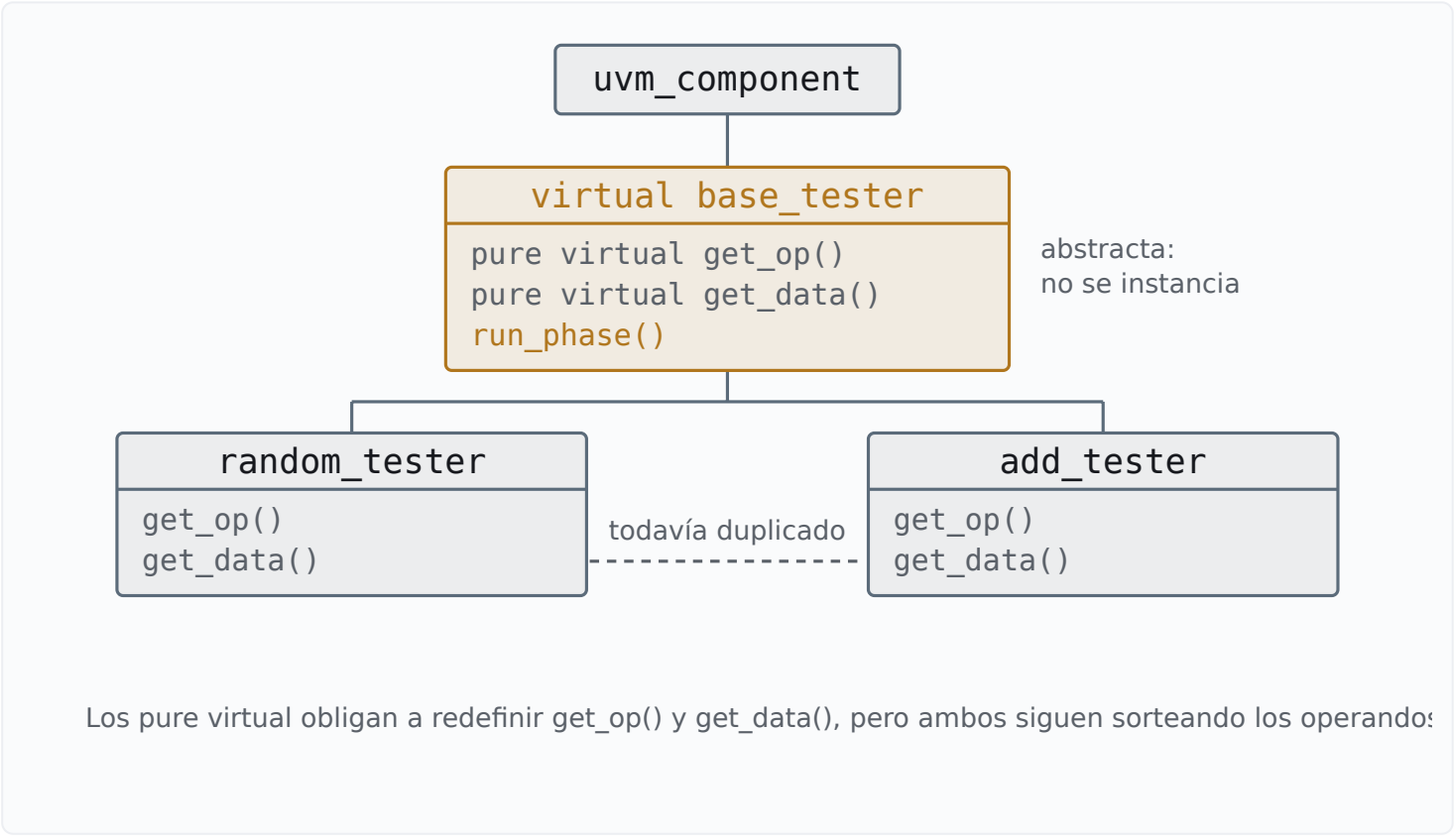
Los testers, en cambio, no duplican nada: add_tester extends random_tester y sólo redefine get_op()

El run_phase() está escrito una sola vez, en random_tester, y add_tester lo hereda.

- Hoy hay dos tests: `random_test` construye un `random_tester`, `add_test` un `add_tester`. Todo lo demás —coverage y scoreboard— lo construyen igual
- `add_test` **hereda** de `random_test` y le **reescribe el `build_phase()` entero**: los tres `new()` son idénticos y están escritos dos veces
- Los testers, en cambio, no duplican nada: `add_tester extends random_tester` y sólo redefine `get_op()`. El `run_phase()` está escrito **una sola vez**, en `random_tester`, y se hereda
- O sea: la escalera de herencia ya está bien resuelta abajo, en los testers, y mal resuelta arriba, en los tests. Las dos slides que siguen muestran cómo quedó abajo; el resto de la sección lo aplica arriba
- Duplicar el `build_phase` no es feo, es caro: diez tests son diez copias, y un cambio en la estructura son diez ediciones

El env: estructura y estímulo

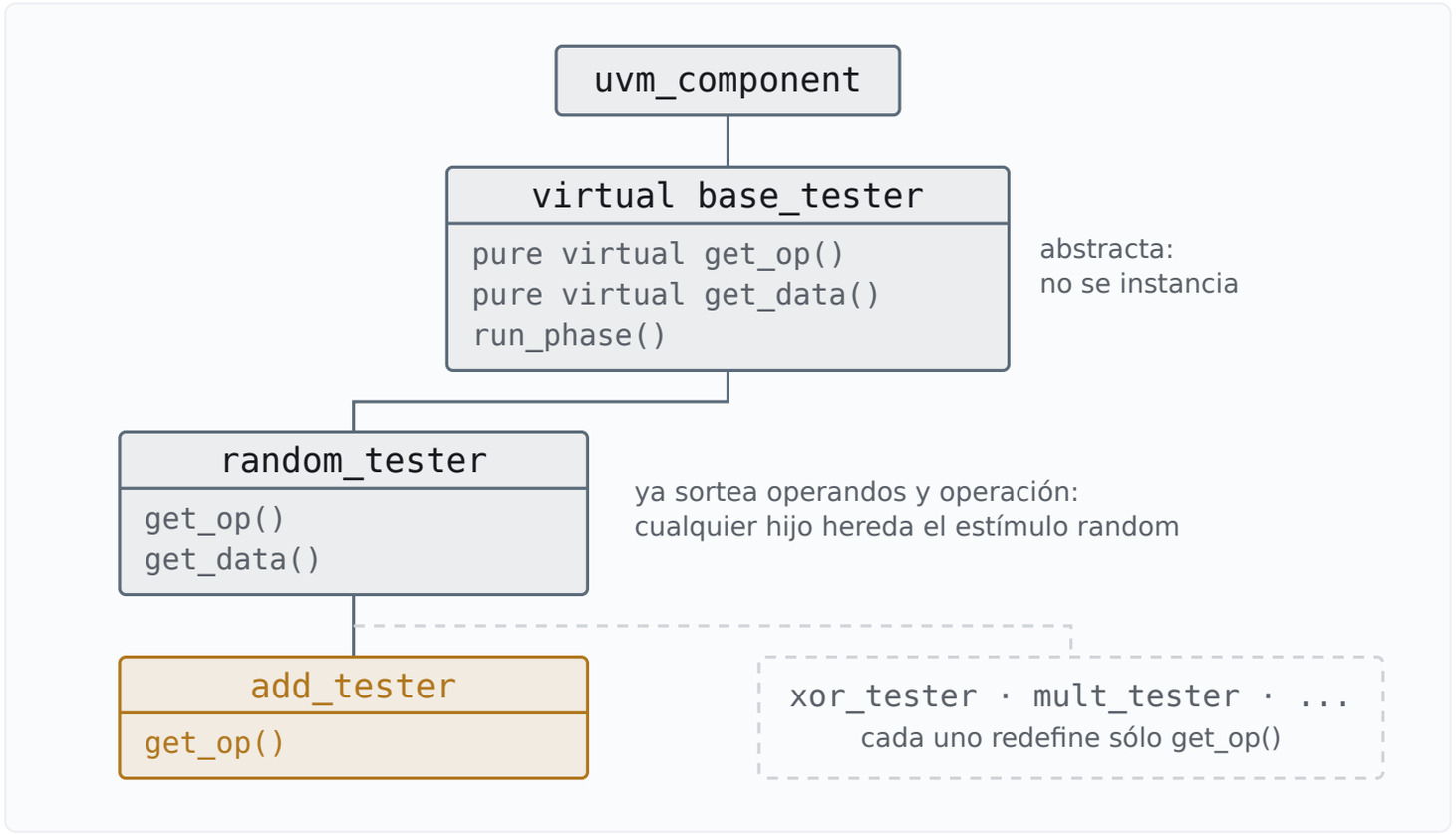
La salida: una clase abstracta, y el estímulo en las hijas



- `base_tester` es una clase **abstracta**: escribe el `run_phase()` una sola vez —mil operaciones— y lo arma llamando a `get_op()` y `get_data()`
- Esos dos métodos son `pure virtual`, así que la clase no se puede instanciar y toda hija está **obligada** a definirlos. La que se olvide, no compila
- Pero todavía queda código repetido: los dos testers randomizan los operandos igual, y eso es señal de que falta un escalón en la jerarquía

El env: estructura y estímulo

El resultado: la jerarquía de tres pisos



- `base_tester` pone el `run_phase()` y declara `get_op()` y `get_data()` como `pure virtual`. Nadie lo instancia
- `random_tester` baja un escalón y define las dos: operación al azar, operandos al azar
- `add_tester` **hereda de** **`random_tester`**, no de `base_tester`: se queda con los operandos random y sólo redefine `get_op()`
- Y el mismo escalón sirve para todos los que vengan: `xor_tester`, `mult_tester`, `maxmult_tester`. Cada uno, una línea útil

El env: estructura y estímulo

add_tester

- El resultado es que *add_tester* es muy simple: hereda de *random_tester* y sólo redefine *get_op()*

```
class add_tester extends random_tester;
   `uvm_component_utils(add_tester)

   function operation_t get_op();
      return add_op;
   endfunction : get_op

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass : add_tester
```

El env: estructura y estímulo

La otra mitad: quién instancia a quién

- Los testers ya están bien repartidos. Falta la otra mitad: **quién los instancia**
- La salida obvia es una familia de `uvm_test`, uno por tester. Y es la mala: mirá el `random_test` de abajo
- Instancia los tres componentes **y** elige el tester. Cada test nuevo copia esa estructura entera para cambiar una línea

```
class random_test extends uvm_test;
   `uvm_component_utils(random_test);

   random_tester tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   // Components get instantiated in build_phase, never in the constructor.
   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

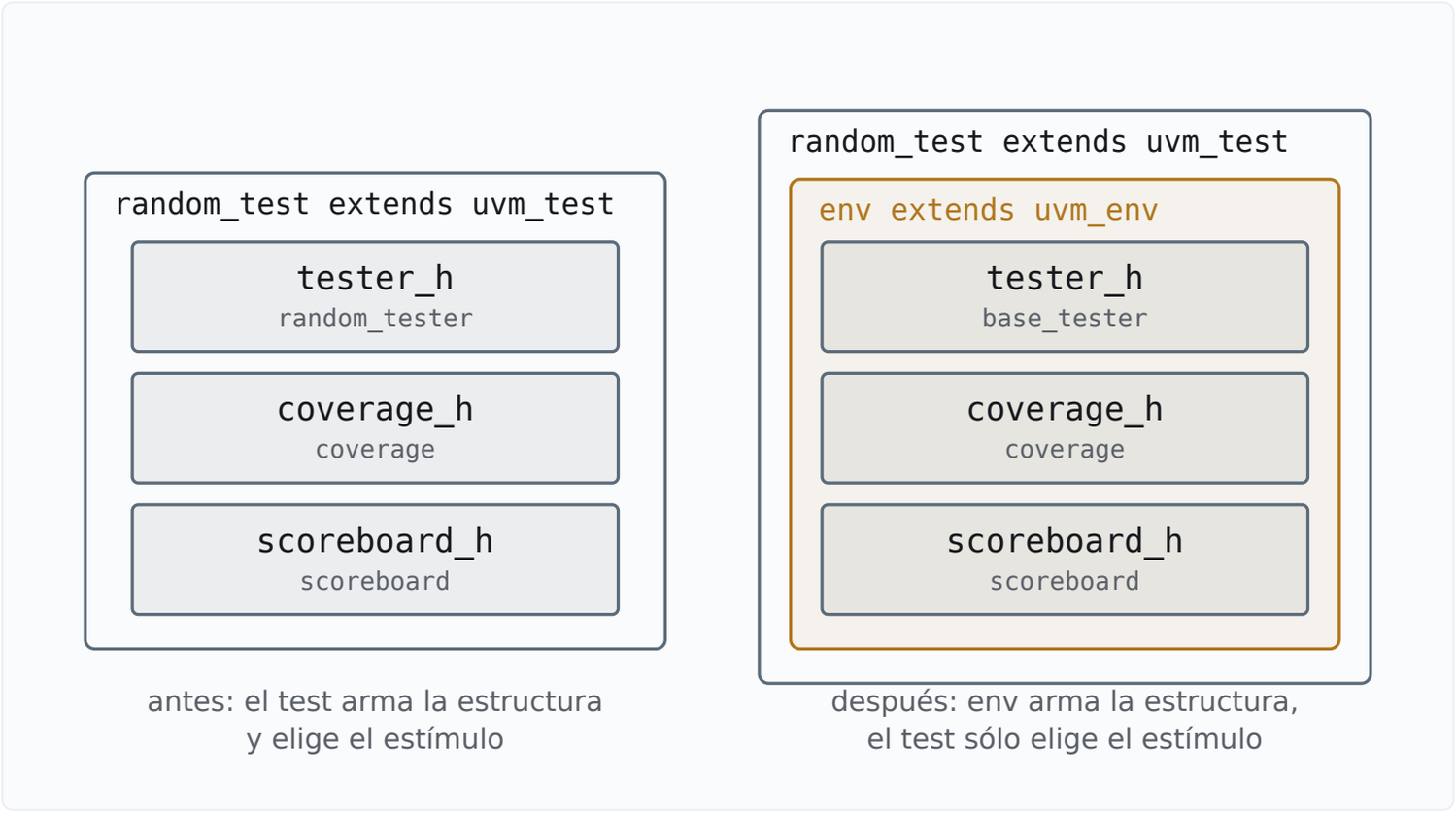
   // No run_phase: the test's job ends once the tree is built.

endclass
```

- El `random_test` hace **dos** cosas: arma la estructura del testbench *y* elige qué estímulo mandar. Son dos motivos de cambio en una sola clase
- UVM parte eso en dos: la estructura se muda a un `uvm_env`, y al test le queda sólo elegir el estímulo
- Un `env` típico define **dos** métodos y nada más: `build_phase` para instanciar y `connect_phase` para conectar

El env: estructura y estímulo

El mapa después de separar



- Dos jerarquías paralelas, y ésa es toda la sección: a la izquierda los tests —el estímulo—, a la derecha el `env` y sus componentes —la estructura—
- El `env` es **uno solo** y no se toca. Los tests son muchos y cambian siempre
- Lo único que los une es la factory: el test dice *"cuando pidan un `base_tester`, dales éste"*

El env: estructura y estímulo

La clase env: instanciar y conectar, nada más

- El `env` define la estructura del testbench: instancia los componentes y después los conecta. Es **uno solo**, y no cambia de un test a otro
- Los tests, en cambio, son muchos y cambian siempre. Ésa es toda la división
- El `build_phase()` no usa `new()`: cada componente se pide con `type_id::create()`, que es la fórmula de la factory de la slide que sigue

```
// The structure of the TB, and nothing else: it instantiates the components and
// connects them. Which tester arrives is the test's call, with a factory override.
class env extends uvm_env;
   `uvm_component_utils(env);

   base_tester tester_h;
   coverage    coverage_h;
   scoreboard  scoreboard_h;

   function void build_phase(uvm_phase phase);
      // base_tester is abstract and this works anyway: create() does not build
      // it, it asks the factory what to hand out when somebody asks for a base_tester.
      tester_h = base_tester::type_id::create("tester_h", this);
      coverage_h = coverage::type_id::create("coverage_h", this);
      scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new
```

```
endclass
```

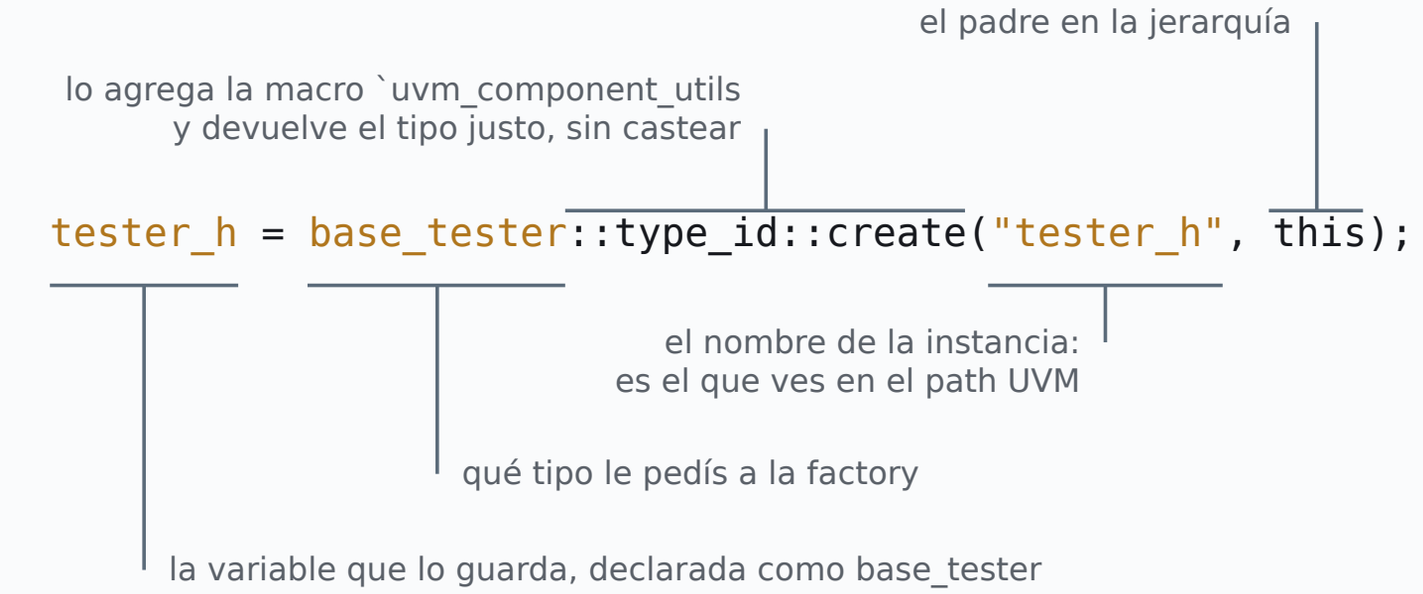
El env: estructura y estímulo

La factory de UVM: la clase se anota sola

- La factory tenía dos molestias: había que castear el resultado, y había que editar el `case` de la fábrica para agregar un tipo
- La de UVM no tiene ninguna de las dos. Una clase se registra **sola**, con una macro en su primera línea:
  - `` `uvm_component_utils() `` para lo que va en el árbol
  - `` `uvm_object_utils() `` para lo que no —las transactions—
- Y devuelve un objeto **del tipo correcto**, sin castear: `type_id` es una clase distinta por cada clase registrada, así que el tipo lo pone el compilador

El env: estructura y estímulo

La fórmula, leída de izquierda a derecha



En ámbar, lo único que cambia de una línea a otra; el resto es siempre igual.

- Es el patrón *static member/method* de las variables estáticas, tal cual: `type_id` es un miembro static de la clase, y `create()` un método static de ese miembro
- Los dos argumentos son los de siempre: el **nombre** que va a salir en la topología, y el **padre** que lo cuelga del árbol
- La clase mal escrita y la que no existe se ven al **compilar**. La que se olvidó el `` `uvm_component_utils ``, sólo si **ningún ancestro** está registrado: si hereda de una registrada, el `type_id` se **hereda** y no hay error

El env: estructura y estímulo

La objeción que salta sola: `base_tester` es abstracta

- Ahora la objeción que salta sola: `tester_h` es de tipo `base_tester`, que es una clase **abstracta**. Y esto compila, corre y anda. ¿Por qué?

```
function void build_phase(uvm_phase phase);
   // base_tester is abstract and this works anyway: create() does not build
   // it, it asks the factory what to hand out when somebody asks for a base_tester.
   tester_h = base_tester::type_id::create("tester_h", this);
   coverage_h = coverage::type_id::create("coverage_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
endfunction : build_phase
```

- Porque `base_tester` acá no es un molde: es un **formulario de pedido**. El `env` no construye un `base_tester`, le pide uno a la factory
- La factory devuelve un objeto de alguna clase **derivada**, y cuál es lo decide un `set_type_override` hecho **antes** de que corra el `build_phase`
- Ese override lo hace el **test**, que es el que instancia el `env`. La estructura no se entera de qué tester le tocó

El env: estructura y estímulo

El override: cambiar la pieza sin tocar el plano

- En los components, `add_test` era `random_test` copiado con una línea cambiada. La copia era el problema, no la solución
- Un `add_tester` **es** un `base_tester`, así que sirve en cualquier lugar donde el código pida un `base_tester` — eso es polimorfismo
- Entonces no hace falta tocar el `env`: alcanza con decirle a la factory *"cuando te pidan un `base_tester`, entregá un `add_tester`"*
- Eso es el **factory override**, y es lo único que un test tiene que hacer

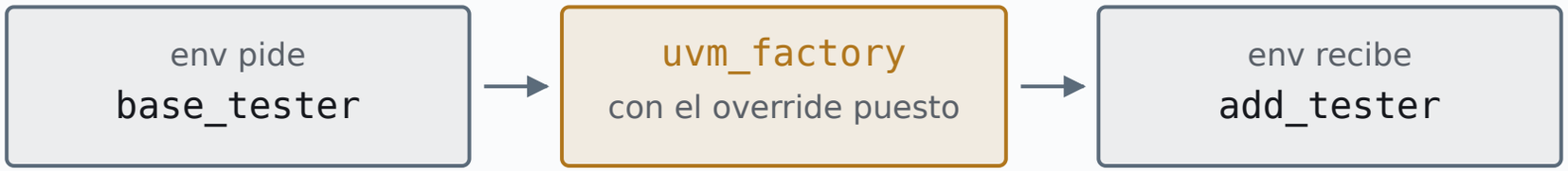
El env: estructura y estímulo

El override de la factory

```
base_tester::type_id::set_type_override(add_tester::get_type());
```

el tipo que el env le pide a la factory

el tipo que la factory devuelve



El override se pone en el build_phase del test, antes de que corra el build_phase del env.

- El método static *set_type_override()* le dice a la factory que cuando vea un pedido por una *base_class_name* retorne un objeto del tipo *new_class_name*
- Esta feature es la que nos permite separar la estructura del TB (*uvm_env*) del tipo de estímulo generado (*uvm_test*)

El env: estructura y estímulo

El override de la factory, en el test

```
class random_test extends uvm_test;
   `uvm_component_utils(random_test);

   env env_h;

   function void build_phase(uvm_phase phase);
      base_tester::type_id::set_type_override(random_tester::get_type());
      env_h = env::type_id::create("env_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass
```

- Vemos que esta versión es más adaptable
- Todos los tests hacen un override de base_tester en función del tester que necesitan para sus estímulos
- La clase env crea el objeto; ponerlo en ejecución es cosa de UVM
- Ahora la clase test hace una cosa bien (generar estímulos) y la clase env hace otra cosa muy bien (crear la estructura)

El env: estructura y estímulo

Por instancia, y la perilla para ver qué quedó puesto

```
base_tester::type_id::set_type_override(add_tester::get_type());          // TODOS
base_tester::type_id::set_inst_override(mult_tester::get_type(),
                                        "uvm_test_top.env_h.t2_h");      // uno solo

uvm_coreservice_t::get().get_factory().print(0);   // qué overrides hay puestos
```

```
Instance Overrides:
  Requested Type  Override Path            Override Type
  base_tester     uvm_test_top.env_h.t2_h  mult_tester
  base_tester     env_h.t3_h               mult_tester   <- registrado y NO aplicó
Type Overrides:
  base_tester     add_tester
```

- `set_inst_override` cambia **una ruta**, no un tipo. Y donde coinciden los dos, gana el de instancia
- La ruta es **absoluta y arranca en `uvm_test_top`**. Medido: `"env_h.t3_h"` se registra igual, no da error, y `t3_h` sale `add_tester`
- `factory.print(0)` es la contraparte de `uvm_top.print_topology()`: uno dice lo que **pediste**, el otro lo que **salió**. Cuando no coinciden, el bug está en el medio
- Con `*` anda: `"*.env_h.t4_h"` aplica. Es el rodeo cuando el test no sabe dónde lo van a instanciar

El env: estructura y estímulo

Resumen de la unidad

- Un testbench tiene **estructura** —qué componentes hay y cómo se conectan— y **estímulo** —qué se manda—. Cambian por motivos distintos y a ritmos distintos
- Hasta los components estaban pegados: `add_test` era una copia de `random_test` con una línea cambiada. Duplicar el bucle no es feo, es **caro**
- El **`uvm_env`** se queda con la estructura; el **test** sólo dice qué estímulo quiere. Un test nuevo deja de tocar el env
- La pieza que lo hace posible es la **factory**: el env pide un `base_tester` y el test decide cuál llega, con `set_type_override`
- La clase abstracta `base_tester` tiene el `run_phase` **escrito una sola vez**; lo único que cada tester redefine es `get_op()` y `get_data()`
- Las tres reglas para escribir código adaptable: **una clase, una cosa**; nada de copiar y pegar estructura; y programar contra la **clase base**

Reporting

El 47 % del tiempo se va acá

- El gráfico de tendencias del día 1 decía que casi la mitad del tiempo de un verificador se va en **debug**. Esta sección es sobre esa mitad
- Un scoreboard que sólo dice `FAIL` te deja justo ahí: sabés que algo está mal y no sabés qué componente, en qué momento, ni con qué datos
- La reacción natural es llenar el código de `$display`, y después borrarlos. Y volver a ponerlos la semana que viene
- UVM lo resuelve al revés: los mensajes **se dejan puestos** y se filtran. Cada uno trae solo el tiempo, la ruta jerárquica del componente y el archivo
- Y hay dos perillas distintas, que se confunden todo el tiempo: la **verbosidad** filtra `` `uvm_info ``; las **actions** controlan warnings, errores y fatales

Reporting

Lo que `$display` no puede hacer

- Mil operaciones por test, decenas de tests por regresión: un `$display` por transacción y el log deja de ser legible antes del primer café
- Y con `$display` la única forma de bajar el ruido es **borrar líneas**, o sea perder justo lo que vas a necesitar la próxima vez
- UVM reemplaza el `$display` por cuatro macros que traen dos cosas gratis: contexto —tiempo, componente, archivo— y un **filtro** que se maneja desde afuera

Reporting

Las macros de reporting

- Son cuatro y se diferencian por la **severidad**: `` `uvm_info ``, `` `uvm_warning ``, `` `uvm_error `` y `` `uvm_fatal ``. **Las cuatro** se cuentan en el *Report Summary*; lo que cambia es la **acción** por defecto — `` `uvm_error `` trae `UVM_DISPLAY|UVM_COUNT` y `` `uvm_fatal `` además mata la simulación
- El **ID** es el primer argumento: un string que dice quién habla. Con él se filtra después, así que conviene que sea el nombre del componente y siempre el mismo
- El **mensaje** es el segundo, y se arma con `$sformatf` cuando lleva datos
- La **verbosidad** es el tercero, y la tiene **sólo** `` `uvm_info ``: es la que decide si el mensaje sale o no

```
// ILLUSTRATION -- not compiled. The signature of each macro, not a call.
`uvm_info   (Message ID String, Message String, Verbosity)
`uvm_warning(Message ID String, Message String)
`uvm_error  (Message ID String, Message String)
`uvm_fatal  (Message ID String, Message String)
```

Reporting

Las macros de reporting: qué hace cada una

- Así se ve en el scoreboard: un `` `uvm_error `` cuando la comparación falla, con el mensaje armado con `$sformatf`
- Y así sale en el log. Fijate lo que trae la línea **sin que nadie lo pida**: el tiempo, el archivo y la línea, y la ruta jerárquica del componente que habló

```
    data_str = $sformatf(
        " %2h %0s %2h = %4h (%4h predicted)",
        cmd.A,
        cmd.op.name(),
        cmd.B,
        t,
        predicted_result
    );

    if (predicted_result != t) `uvm_error("SCOREBOARD", {"FAIL: ", data_str})
    else `uvm_info("SCOREBOARD", {"PASS: ", data_str}, UVM_HIGH)
```

```
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL:  5b add_op ff = 015a (015b predicted)
```

Reporting

Los seis niveles, y por qué el mensaje se queda

- El ciclo de siempre: llenás el código de `$display`, encontrás el bug, borrás todo — y la semana que viene lo escribís de nuevo
- Con verbosidad los mensajes **se quedan puestos** y no molestan. Son dos pasos, y el segundo no toca el código:
  - poner una verbosidad en cada `` `uvm_info ``, según cuánto ruido es
  - fijar el **techo** de la corrida: global, o por rama del árbol
- Un mensaje sale si su verbosidad es **menor o igual** al techo. Por eso `UVM_NONE` sale siempre

```
// ILLUSTRATION -- not compiled. uvm_object_globals.svh:382-390, uvm-core 2020.3.1,
// copied verbatim: code/.uvm/ is downloaded and gitignored, so the slide cannot
// pull the enum straight out of the library.
typedef enum {
   UVM_NONE   = 0,
   UVM_LOW    = 100,
   UVM_MEDIUM = 200,  // the default: with nothing set, UVM_MEDIUM is what you get
   UVM_HIGH   = 300,
   UVM_FULL   = 400,
   UVM_DEBUG  = 500
} uvm_verbosity;

// Any message above the verbosity that was set does not get printed
```

Reporting

El techo global: un plusarg, sin recompilar

- Se fija de dos formas, y la primera es la que se usa todos los días: un **plusarg** en la línea de comando
- No hay que tocar código ni recompilar. El mismo binario da el log de la regresión y el log del debug
- El problema aparece con el equipo: si son diez personas con sus propios mensajes de debug, subir el techo global te trae los de todos
- Por eso hace falta poder pedirlo **por componente**, que es la slide siguiente

```
./obj_dir/top/sim +UVM_TESTNAME=random_test +UVM_VERBOSITY=UVM_HIGH
```

Reporting

`uvm_cmdline_processor`: la línea de comandos, entera

- `+UVM_VERBOSITY` y `+UVM_TESTNAME` no los lee magia: los lee **una clase**, y está disponible para tus propios plusargs

```
uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();
string valor;

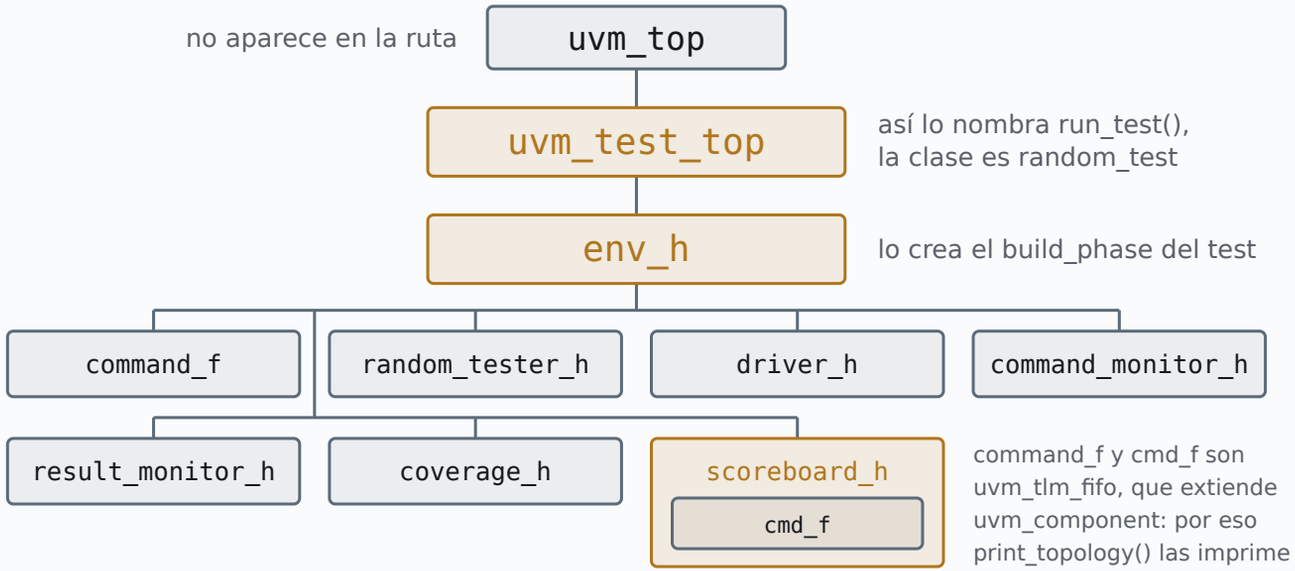
if (clp.get_arg_value("+COUNT=", valor)) count = valor.atoi();
```

- Es un **singleton**: `get_inst()` desde cualquier clase, sin construir nada y sin pasarlo por el `config_db`
- `get_arg_value` devuelve **cuántas veces apareció** el argumento, no el valor: el valor sale por el `ref`. Si apareció dos veces te enterás — `$value$plusargs` se queda callado con el primero
- Y están `get_args()`, `get_plusargs()` y `get_uvm_args()`, que devuelven la línea de comandos entera en un queue: es como se imprime en qué condiciones corrió una regresión de hace tres meses

Reporting

El techo por rama: dos métodos y un árbol

- Todo `uvm_component` trae métodos para fijar su propio techo, en dos sabores: sólo él, o él **y todo lo que cuelga debajo** (`_hier`)
- Ojo con qué jerarquía es: **no** es la de módulos del DUT. Es el árbol que UVM arma llamando `build_phase()` de arriba hacia abajo
- Es el mismo árbol del que sale el prefijo de cada mensaje, y el mismo que usa el `uvm_config_db` como ámbito




```
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290:
uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL: 5b add_op ff
```

Ese prefijo es la ruta del componente dentro del árbol.

Reporting

El techo por rama: dónde va la llamada

- El llamado va **después** de que la jerarquía exista —si no, el componente todavía no está— y **antes** de que arranque la simulación
- Esa ventana tiene nombre y es una de las nueve fases: `end_of_elaboration_phase`
- `set_report_verbosity_level_hier()` alcanza al componente **y a sus hijos**; `set_report_verbosity_level()`, sólo a él

```
// To use the UVM hierarchy to control verbosity, the reporting methods have to
// be called after the UVM hierarchy has been built, but BEFORE the simulation
// starts. That is what end_of_elaboration_phase is for
function void end_of_elaboration_phase(uvm_phase phase);
   //scoreboard_h.set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION);
   scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

Reporting

Apagar warnings, errores y mensajes fatales

- Situación real: el DUT está bien, el scoreboard predice mal la suma, y la clase la está arreglando otra persona. Necesitás seguir trabajando mientras tanto
- Bajar el techo de verbosidad **no sirve**: sólo afecta a `` `uvm_info ``, y lo que hay que callar es un `` `uvm_error ``
- Hace falta otra perilla

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
UVM_INFO tb_classes/scoreboard.svh(40) @ 150: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] PASS:  43 and_op ff = 0043 (0043 predicted)
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL:  5b add_op ff = 015a (015b predicted)
...
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    4
UVM_WARNING :    0
UVM_ERROR :    2
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[SCOREBOARD]     5
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 820ps
```

Reporting

Apagar mensajes: por dónde se hace

- Warnings, errores y fatales son **inmunes** al techo de verbosidad, y está bien que así sea: nadie quiere apagar un error sin querer
- La perilla que sí los alcanza se llama **actions**, y responde a otra pregunta: no *"¿se imprime?"* sino *"¿qué se hace con este mensaje?"*
- Imprimir es sólo una de las siete cosas posibles

```
// ILLUSTRATION -- not compiled. uvm_object_globals.svh:354-364, uvm-core 2020.3.1,
// copied verbatim: code/.uvm/ is downloaded and gitignored, so the slide cannot
// pull the enum straight out of the library.
typedef enum {
   UVM_NO_ACTION = 'b0000000,
   UVM_DISPLAY   = 'b0000001,
   UVM_LOG       = 'b0000010,
   UVM_COUNT     = 'b0000100,
   UVM_EXIT      = 'b0001000,
   UVM_CALL_HOOK = 'b0010000,
   UVM_STOP      = 'b0100000,
   UVM_RM_RECORD = 'b1000000
} uvm_action_type;
```

Reporting

Apagar mensajes: el ejemplo completo, y el log que queda

- `set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION)` sobre el scoreboard: cuando aparezca un error ahí adentro, que no haga nada
- Va en el `end_of_elaboration_phase` del `env`, por el mismo motivo que la verbosidad jerárquica
- Y el log de abajo es el resultado: mismo DUT, mismo scoreboard roto, cero errores en el *Report Summary*
- En `code/u4/reporting` la línea está escrita y **comentada**: se descomenta para reproducir este log, y se vuelve a comentar. Que viva comentada en el repo es parte de la lección

```
// To use the UVM hierarchy to control verbosity, the reporting methods have to
// be called after the UVM hierarchy has been built, but BEFORE the simulation
// starts. That is what end_of_elaboration_phase is for
function void end_of_elaboration_phase(uvm_phase phase);
   //scoreboard_h.set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION);
   scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
--- UVM Report Summary ---
```

```
** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 820ps
```

Reporting

`uvm_report_catcher`: demotar el error que esperabas

- `UVM_NO_ACTION` apaga **todos** los errores de una rama. Cuando el test verifica el camino de error a propósito —un `PSLVERR` que tiene que llegar— lo que hay que callar es **uno solo**, y el resto tiene que seguir gritando

```
class demotar_pslverr extends uvm_report_catcher;
   virtual function action_e catch();
      if (get_severity() == UVM_ERROR && get_id() == "PSLVERR")
         set_severity(UVM_INFO);      // deja de contar como error
      return THROW;                   // y sigue viaje, ya degradado
   endfunction
endclass

demotar_pslverr c = new();
uvm_report_cb::add(null, c);          // null = todo el testbench; o un componente
```

- El catcher se mete **antes** del reporte: ve cada mensaje y puede cambiarle la severidad, el ID, el texto o la acción
- `THROW` lo deja seguir ya modificado; `CAUGHT` lo hace desaparecer
- Y la diferencia que importa: el resumen imprime *Number of demoted UVM_ERROR reports : 1*. **El apagón deja rastro**, que es lo que `UVM_NO_ACTION` no hace

Reporting

Resumen de la unidad

- Dos perillas, y no se cruzan: **verbosidad** filtra `` `uvm_info ``; **actions** decide qué se hace con warnings, errores y fatales
- Las dos se pueden fijar por componente o por rama, en `end_of_elaboration_phase`
- Y hay una tercera para el error **esperado**: el `uvm_report_catcher` lo demota y lo deja contado en el resumen, en vez de apagarlo sin dejar rastro
- Con eso los mensajes de debug se escriben una vez y se quedan puestos para siempre — que es lo único que hace el debug barato
- Y llega acá, al final del día 3, a propósito: es la herramienta con la que se debuggean los ejercicios de los días 4 y 5 — el del día 5 se resuelve **subiendo la verbosidad**, y no hay otra forma de encontrarlo

Repaso · Día 3

1 de 7 · uvm_test

¿Qué te permite hacer `+UVM_TESTNAME=add_test` que antes no podías?

- **a** Elegir el test sobre un testbench ya compilado ✓
- **b** Correr más rápido la simulación
- **c** Cambiar el DUT sin recompilar: UVM vuelve a elaborar al arrancar
- **d** Bajar la verbosidad de los mensajes

Elegir el test sin recompilar — UVM lee ese plusarg y le pide el test a la **factory** por nombre. Es la diferencia entre 1000 tests × 5 minutos de compilación y una sola compilación.

Repaso · Día 3

2 de 7 · UVM Phases

¿En qué orden recorre UVM la jerarquía en `build_phase` y en `connect_phase`?

a. Las dos de arriba hacia abajo

b. Las dos de abajo hacia arriba

c. `build_phase` top-down, `connect_phase` bottom-up ✓

d. En el orden en que se declararon los componentes, de arriba hacia abajo

Build top-down, connect bottom-up — y tiene sentido: no podés conectar un componente que todavía no existe, así que primero se construye toda la jerarquía y recién después se conecta.

Repaso · Día 3

3 de 7 · Objections

¿Para qué sirve `raise_objection()` / `drop_objection()` en el `run_phase`?

a Para reportar errores del scoreboard

b Para mantener viva la simulación mientras hay trabajo ✓

c Para sincronizar dos threads

d Para que las `run_phase` de todos los componentes arranquen a la vez

Para que la fase no termine antes — todos los `run_phase` corren en paralelo, cada uno en su thread, y la fase termina cuando **cae la última objection**. Sin levantarla, la simulación se te termina en el tiempo 0.

Repaso · Día 3

4 de 7 · El env

¿Qué le toca a cada uno: `uvm_env` y `uvm_test`?

a `env` genera el estímulo; `test` arma la estructura

b Los dos hacen lo mismo, `env` es opcional

c `env` corre el DUT; `test` corre el scoreboard

d `env` arma la estructura; `test` elige el estímulo ✓

Estructura vs. estímulo — cada clase hace **una sola cosa bien**. Por eso el `env` casi siempre tiene sólo `build_phase` y `connect_phase`, y el test casi siempre tiene sólo un override de la factory.

Repaso · Día 3

5 de 7 · Factory override

`set_type_override()` reemplaza `base_tester` por `add_tester`. ¿Cuándo hay que llamarlo?

- **a** Antes de que corra el `build_phase` que crea el objeto ✓
- **b** En cualquier momento: la factory lo aplica retroactivamente a lo ya creado
- **c** Después del `connect_phase`
- **d** Dentro del `run_phase` del tester

Antes del `build_phase` — la factory decide qué construir **en el momento del `create()`**. Si el override llega tarde, el `env` ya instanció la clase base y no hace nada.

Repaso · Día 3

6 de 7 · Verbosidad

El techo de verbosidad (`+UVM_VERBOSITY=UVM_HIGH`), ¿sobre qué macros actúa?

a Sobre las cuatro: info, warning, error y fatal

b Sólo sobre `` `uvm_info `` ✓

c Sobre error y fatal solamente

d Sobre ninguna: sólo cambia el formato del mensaje

Sólo sobre `uvm_info` — warnings, errores y fatales son **inmunes** al techo de verbosidad, y está bien que así sea: nadie quiere apagar un error sin querer. Para esos hace falta el mecanismo de *actions*.

Repaso · Día 3

7 de 7 · Report actions

Querés silenciar los `uvm_error de un scoreboard que otra persona está arreglando. ¿Dónde va el `set_report_severity_action_hier()`?

a. En el `build_phase` del `env`

b. En el `run_phase` del test, antes de levantar el objection

c. En el constructor del scoreboard

d. En el `end_of_elaboration_phase` del `env` ✓

En `end_of_elaboration_phase` — tiene que ser **después** de que la jerarquía esté construida (si no, el componente todavía no existe) y **antes** de que arranque la simulación. Esa fase es exactamente esa ventana.

Ejercicio · Día 3 · 1 de 2

Un test nuevo sin tocar la estructura

```
$ cd code/ejercicios/d3 && bash run.sh
```

- Escribí `mult_tester` y `mult_test`, con `set_type_override`
- **Sin tocar `env.svh`**: la estructura no se entera de que cambió el estímulo
- El `env` trae un componente `chequeo` que corrige el ejercicio solo

Ejercicio · Día 3 · 2 de 2

El `uvm_error` que no dice nada

```
$ cd code/ejercicios/d3b && bash run.sh
```

- El scoreboard está **bien** —atrapa todos los errores— y cuando falla dice `FAILED` y nada más: menos todavía que la línea del ejercicio de las ondas del día 1
- Que el `uvm_error` diga **cuál** falló: `A`, `B`, la operación, el resultado del DUT y el que predijiste
- Y que el que **pasa** también se imprima, con `UVM_HIGH`, para que no aparezca en el log de todos los días
- El corrector mira el mismo bus que vos y sabe cuál fue la primera que falló

Ayer quedó...

Dónde dejamos el testbench

- Entró UVM: `run_test()`, el test, las fases, el `env` y el scoreboard, y el `+UVM_TESTNAME` que elige la clase desde la línea de comandos
- El **objection** es lo que mantiene viva la simulación, y el reporting ya distingue verbosidad de *action*
- Y quedó un cuello: el scoreboard es **uno solo** y está cableado al monitor. El día que haya dos oyentes hay que editar al que publica

Agenda

Día 4 · ≈ 4 h · unidad 5 · cómo hablan los componentes

- Un productor, muchos oyentes
- Un solo lugar que mira el cable
- Cuando alguien tiene que esperar
- Quién espera a quién

Al final del día podés:

- **Colgar** un subscriber nuevo de un analysis port **sin tocar al que publica**
- **Elegir** entre un analysis port y una FIFO de TLM, y decir cuál de los dos bloquea
- **Sacar** el `#500` del final de un test y reemplazarlo por el objection que corresponde

Un productor, muchos oyentes

Dos formas de hablar entre objetos

- El paradigma estructural pregunta *qué pasos seguir*. La OOP pregunta **qué objetos hay y cómo se hablan** — y ahí se acaba el testbench de un solo `initial`
- Y hablarse, entre objetos, es exactamente dos cosas. La diferencia está en si los dos corren en **el mismo thread** o no
- **Mismo thread:** uno llama al método del otro y vuelve enseguida. Es el `write()` de un `uvm_analysis_port` — las dos primeras secciones del día
- **Threads distintos:** los dos corren a la vez y hay que **coordinarlos**, o sea que alguien va a tener que esperar. Es `put` / `get` y las TLM FIFOs — las dos últimas
- El día 4 entero son esas dos columnas. Arrancamos por la de la izquierda

Un productor, muchos oyentes

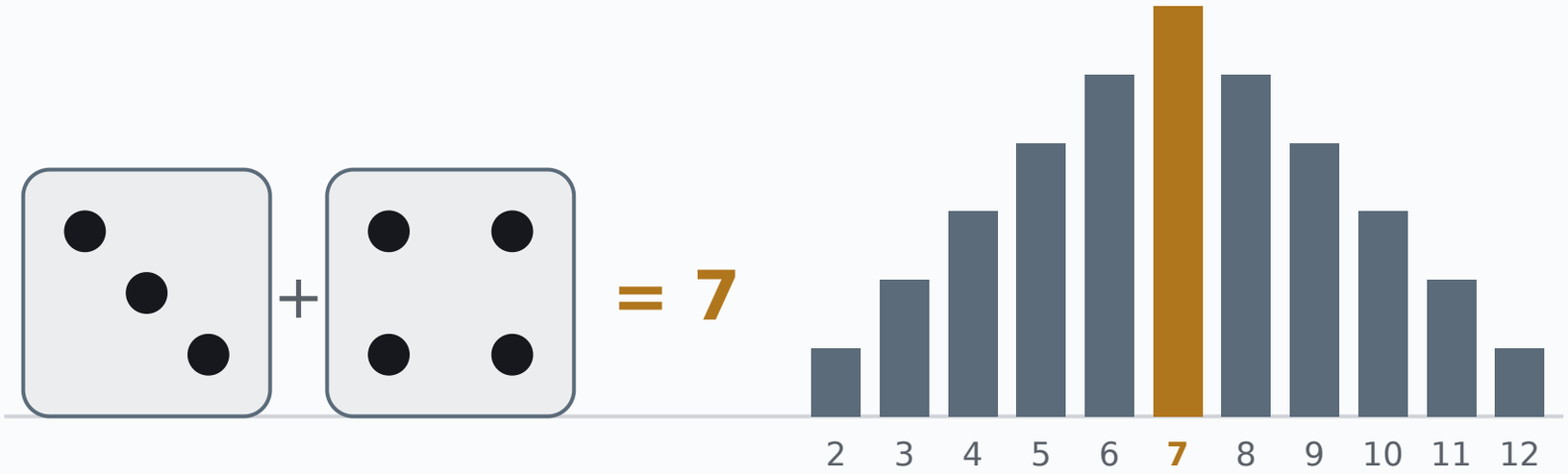
El que publica no sabe quién lo lee

- Un dato del testbench casi nunca tiene **un** destinatario: cada comando que ve el monitor lo quieren el scoreboard, la cobertura y, mañana, el log
- La solución obvia es que el productor guarde un handle a cada consumidor y les llame el método. Funciona, y agregar un consumidor obliga a **tocar el productor**
- Es el mismo problema que resolvió el BFM en interfaces y BFM, un nivel más arriba: ahí escondimos el cable, acá queremos esconder **la lista de destinatarios**
- La respuesta es un patrón viejo: el productor publica en un puerto y no sabe —ni le importa— quién está del otro lado
- La sección lo arma primero a mano con dados, y recién después lo reemplaza por `uvm_analysis_port`. El ejemplo es ajeno al hardware a propósito

Un productor, muchos oyentes

El problema: dos dados, tres cosas que hay que contar

- Escribir un programa que simule la ejecución de tirar dos dados (2d6) 20 veces e informe lo siguiente
  - El promedio de los valores que dieron los dados
  - Un histograma con la frecuencia de valores
  - Un reporte de cobertura que muestre si salieron todos los posibles valores de 2 a 12



20 tiradas de 2d6: promedio, histograma y cobertura de 2 a 12.

La suma nunca es 1 ni 13, y el 7 sale en 6 de las 36 combinaciones.

Un productor, muchos oyentes

Los tres observadores: la misma forma, tres cuerpos

```
class average extends uvm_component;
   `uvm_component_utils(average);

   protected real dice_total;
   protected real count;

   function new(string name, uvm_component parent = null);
      super.new(name, parent);
      dice_total = 0.0;
      count = 0.0;
   endfunction : new

   function void write(int t);
      dice_total = dice_total + t;
      count++;
   endfunction : write

   function void report_phase(uvm_phase phase);
      `uvm_info("AVERAGE", $sformatf("DICE AVERAGE: %2.1f", dice_total / count), UVM_NONE)
   endfunction : report_phase

endclass : average
```

- `average` es el más simple: un `write(int t)` que acumula, y un `report_phase()` que divide e informa al final
- Ésa es la **forma** que van a tener los tres: reciben una tirada de a una, y el resultado recién existe cuando terminó todo

Un productor, muchos oyentes

Los otros dos: cambia el cuerpo, no la forma

```
function void write(int t);
   rolls[t]++;
endfunction : write
```

```
function void write(int t);
   the_roll = t;
   dice_cg.sample();
endfunction : write
```

- `histogram` guarda las tiradas en un array asociativo y dibuja las barras en el `report_phase`
- `coverage` copia la tirada a `the_roll` y llama a `sample()`: es el covergroup del testbench convencional, con el muestreo atado al dato en vez de al reloj
- Los tres `write()` son distintos por dentro y **idénticos por fuera**: mismo nombre, mismo argumento. Eso es lo que va a permitir tratarlos igual

Un productor, muchos oyentes

El productor: tira los dados y devuelve un número

```
class dice_roller extends uvm_component;
   `uvm_component_utils(dice_roller);

   rand byte die1;
   rand byte die2;

   constraint d6 {
      die1 >= 1;
      die1 <= 6;
      die2 >= 1;
      die2 <= 6;
   }

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function byte two_dice();
      byte the_roll;
      void'(randomize());
      the_roll = die1 + die2;
      return the_roll;
   endfunction : two_dice

endclass : dice_roller
```

- `dice_roller` randomiza dos bytes con un `constraint` que los deja entre 1 y 6, y devuelve la suma
- Es todo lo que hace, y está bien que sea todo: **produce el dato y no sabe nada de quién lo usa**
- Fijate lo que **no** tiene: ni un handle a `coverage_h`, ni a `histogram_h`, ni a `average_h`. Esa lista está en otro lado, y ahí está el problema

Un productor, muchos oyentes

El que reparte: el test, y no debería

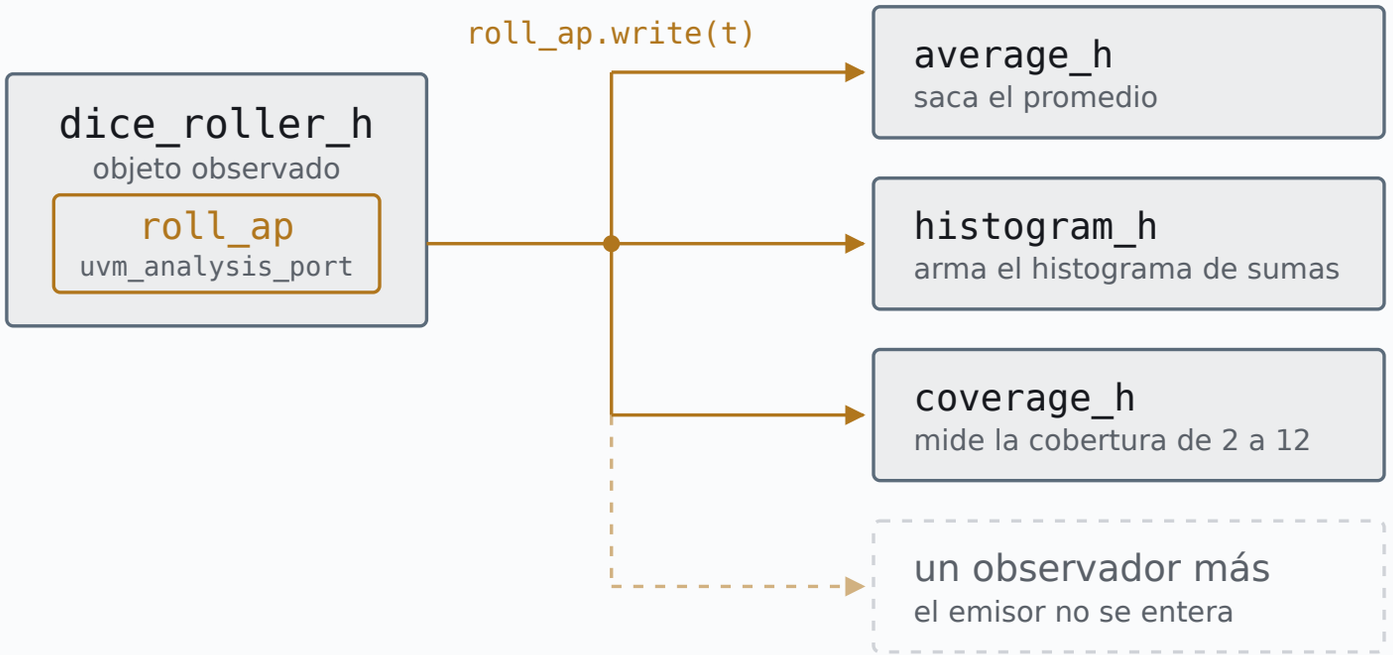
- El `dice_test` extiende `uvm_test`, instancia los cuatro componentes y arranca
- Funciona. Pero mirá el `run_phase`, y en particular qué hay adentro del `repeat (20)`

```
task run_phase(uvm_phase phase);
   int the_roll;
   phase.raise_objection(this);
   // This part needs attention.
   // What belongs here is:
   //  the Observer design pattern
   //  a UVM analysis port
   // Otherwise none of the OOP magic is being used:
   // these are plain functions passing data along
   repeat (20) begin
      the_roll = dice_roller_h.two_dice();
      coverage_h.write(the_roll);
      histogram_h.write(the_roll);
      average_h.write(the_roll);
   end
   phase.drop_objection(this);
endtask : run_phase
```

Un productor, muchos oyentes

Observer Design Pattern

- El que publica un tweet no sabe quién lo lee ni qué hace con él, y sin embargo llega a todos. Eso es el **Observer**
- Un objeto produce un dato y lo publica. Los que necesitan ese dato se suscriben. El productor **no lleva la lista**
- Acá el observado es `dice_roller_h`; los observadores —*subscribers*— son `coverage_h`, `histogram_h` y `average_h`
- La propiedad que compra todo: agregar un cuarto observador **no toca una línea** del que produce

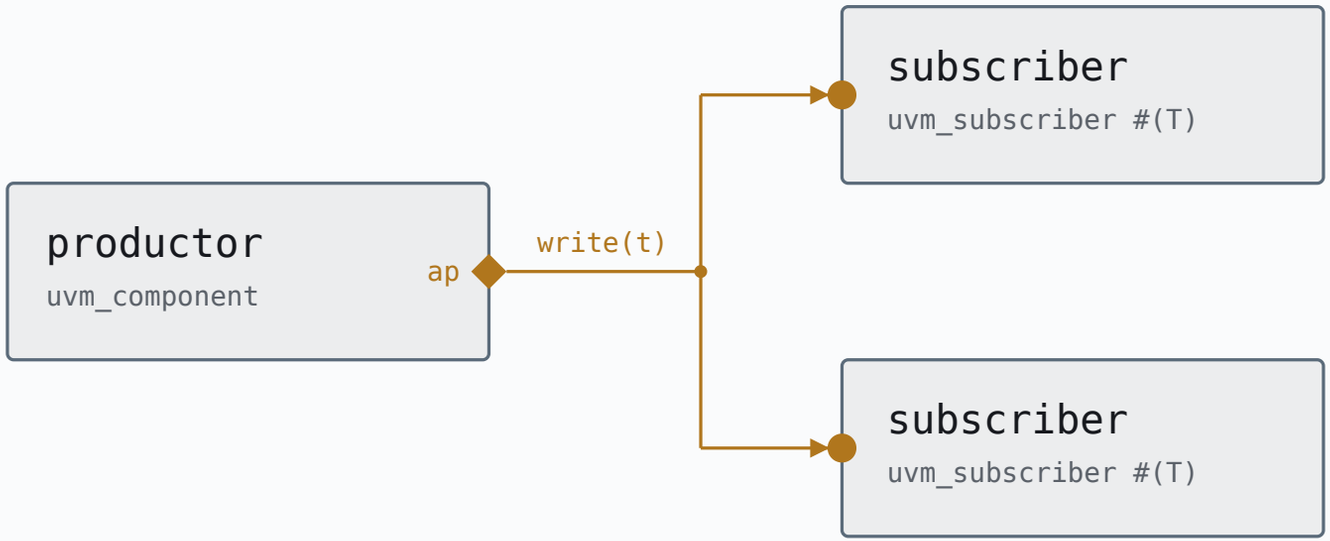


Las flechas van en un solo sentido: el emisor no sabe quiénes ni cuántos lo escuchan.

Un productor, muchos oyentes

El patrón, ya hecho: las dos puntas del cable

- UVM trae el patrón hecho, en dos clases que son las dos puntas del cable:
  - *uvm_analysis_port:* Envía data a un conjunto de subscribers (observadores)
  - *uvm_subscriber:* Extensión de uvm_component que permite al componente suscribirse a un uvm_analysis_port



◆ **uvm_analysis_port** lo declara el productor; publica con .write()

● **analysis_export** lo trae uvm_subscriber; se ata con .connect()

En el ejemplo: dice_roller publica el tiro; coverage, histogram y average lo reciben.

Un productor, muchos oyentes

uvm_analysis_port

- Declaramos una variable del tipo *uvm_analysis_port* que defina el tipo de datos que va a transportar
- Instanciamos el analysis port en *build_phase*
- Escribimos datos en el puerto mediante el método *.write()*
- Una vez que escribimos datos en el puerto, este va a todos sus subscribers
- Utilizamos el método *connect()* para conectar los subscribers al puerto. Este método tiene un único argumento, y es el **export** del subscriber (`analysis_export`), no otro port

```
// Same as version 01, but publishing: instead of returning the number it writes
// it to an analysis port, and stops knowing who reads it.
class dice_roller extends uvm_component;
   `uvm_component_utils(dice_roller);

   rand byte die1;
   rand byte die2;

   constraint d6 {
      die1 >= 1;
      die1 <= 6;
      die2 >= 1;
      die2 <= 6;
   }
```

```
   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   // 1) declare the port, with the data type it will carry
   uvm_analysis_port #(int) roll_ap;

   // 2) instantiate it in build_phase, with new(): ports do not go through the factory
   function void build_phase(uvm_phase phase);
      roll_ap = new("roll_ap", this);
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      int the_roll;
      phase.raise_objection(this);
      repeat (20) begin
         void'(randomize());
         the_roll = die1 + die2;
         // 3) write: the port calls write() on every subscriber
         roll_ap.write(the_roll);
      end
      phase.drop_objection(this);
   endtask : run_phase

endclass : dice_roller
```

Un productor, muchos oyentes

uvm_subscriber

- Extendemos la clase *uvm_subscriber* paramétricamente, para saber qué tipo de datos va a leer
- La clase nos da un objeto llamado *analysis_export*
- La clase requiere que creemos un método llamado *write()* que tiene un único argumento *t*, que es del mismo tipo que la extensión de la clase
- En nuestro ejemplo, tenemos 3 subscribers (average, coverage y histogram)

```
// uvm_subscriber:
//    extends the uvm_component class and lets you hook onto an analysis port
//    It is a parameterizable class!
//    The class gives you something and asks for something in return
//       * It gives you an object called analysis_export
//       * It asks you to write a write() method that handles the data

class coverage extends uvm_subscriber #(int);
   `uvm_component_utils(coverage);
   int the_roll;

   covergroup dice_cg;
      coverpoint the_roll {bins twod6[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};}
   endgroup

   function new(string name, uvm_component parent = null);
      super.new(name, parent);
      dice_cg = new();
   endfunction : new
```

```
   function void write(int t);
      the_roll = t;
      dice_cg.sample();
   endfunction : write

   function void report_phase(uvm_phase phase);
      `uvm_info("COVERAGE", $sformatf("COVERAGE: %2.0f%%", dice_cg.get_inst_coverage()),
                UVM_NONE)
   endfunction : report_phase

endclass : coverage
```

Un productor, muchos oyentes

El productor, ahora publicando

```
// 1) declare the port, with the data type it will carry
uvm_analysis_port #(int) roll_ap;

// 2) instantiate it in build_phase, with new(): ports do not go through the factory
function void build_phase(uvm_phase phase);
   roll_ap = new("roll_ap", this);
endfunction : build_phase

task run_phase(uvm_phase phase);
   int the_roll;
   phase.raise_objection(this);
   repeat (20) begin
      void'(randomize());
      the_roll = die1 + die2;
      // 3) write: the port calls write() on every subscriber
      roll_ap.write(the_roll);
   end
   phase.drop_objection(this);
endtask : run_phase
```

- Los tres pasos numerados son todo lo que cambió respecto de la versión anterior: declarar el port, instanciarlo en `build_phase`, escribir con `write()`
- `roll_ap = new(...)`: los ports **no** se crean con la factory. Son el cableado del testbench, no piezas intercambiables
- El `two_dice()` que devolvía un número desapareció: ahora la tirada se publica y el productor sigue de largo

Un productor, muchos oyentes

connect_phase(): dónde se arma el cableado

- Necesitamos vincular al productor con el consumidor
- UVM provee un phase method (*connect_phase()*) para conectar los objetos
- UVM llama al método build_phase de manera TOPDOWN, una vez terminado con todos los objetos en la jerarquía, UVM llama al método connect_phase de manera BottomUP
- El proceso de conexión tiene 2 partes fundamentales
  - Los uvm_subscribers contienen un objeto llamado analysis_export, que no debemos instanciarlo ya que viene cuando extendemos la clase uvm_subscriber
  - La clase uvm_analysis_port provee un método llamado connect()

Un productor, muchos oyentes

connect_phase(): dónde se arma el cableado

- Una sola línea de `connect_phase()` engancha al subscriber con el analysis port: `ap.connect(sub_h.analysis_export)`

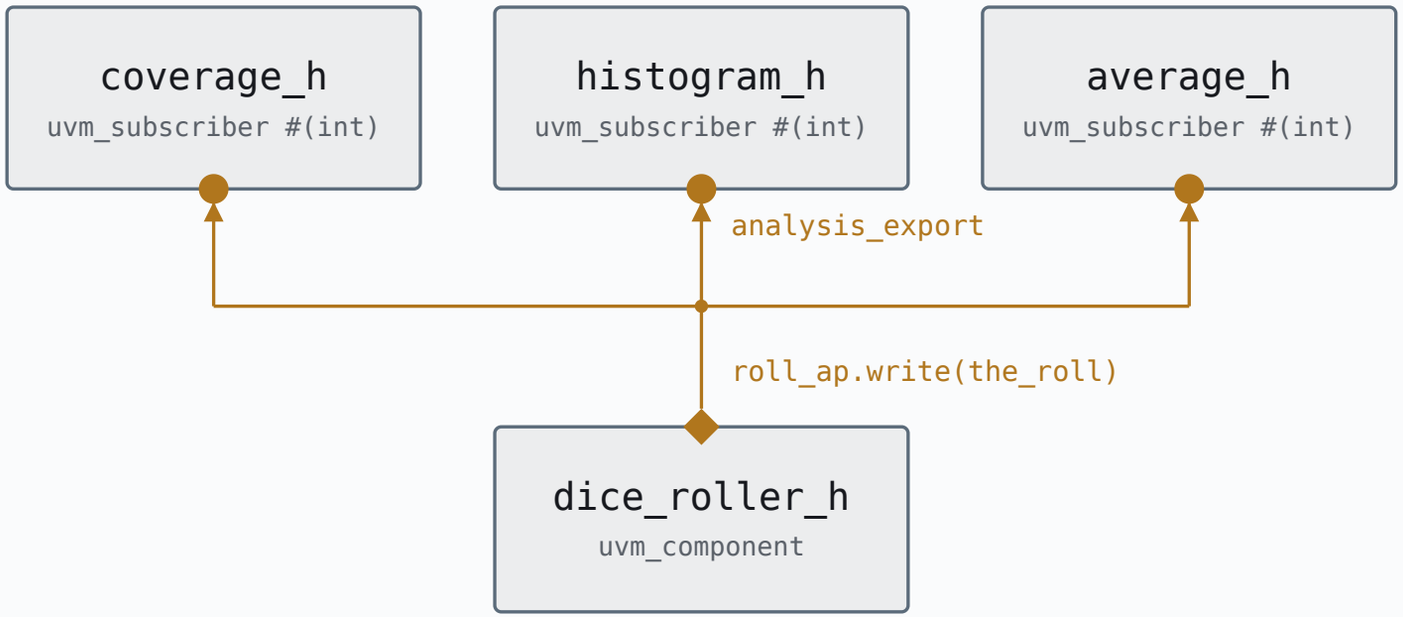
```
// build_phase: UVM calls it TOP-DOWN, the parent first.
function void build_phase(uvm_phase phase);
   dice_roller_h = new("dice_roller_h", this);
   coverage_h = new("coverage_h", this);
   histogram_h = new("histogram_h", this);
   average_h = new("average_h", this);
endfunction : build_phase

// connect_phase: BOTTOM-UP, and always port.connect(export).
function void connect_phase(uvm_phase phase);
   dice_roller_h.roll_ap.connect(coverage_h.analysis_export);
   dice_roller_h.roll_ap.connect(histogram_h.analysis_export);
   dice_roller_h.roll_ap.connect(average_h.analysis_export);
endfunction : connect_phase
```

Un productor, muchos oyentes

La solución, cableada

- En el diagrama de conexionado podemos ver que la clase dice_roller tiene un objeto uvm_analysis_port llamado roll_ap
- Cada subscriber tiene un objeto analysis_export
- La conexión entre ambos se realiza mediante el método connect()



`connect_phase: roll_ap.connect(coverage_h.analysis_export)`

Un solo write() en roll_ap llega a los tres: el productor no sabe quiénes lo escuchan.

Un productor, muchos oyentes

Resumen de la unidad

- Dos objetos se hablan de **exactamente dos formas**, y la diferencia es si corren en el mismo thread o no
- **Mismo thread**: uno llama al método del otro y vuelve enseguida. Es una `function`, **no puede bloquear**. Es el `write()` de un analysis port
- **Threads distintos**: hay que coordinarlos y alguien va a esperar. Es una `task`, **puede bloquear**. Es `put` / `get`, y son las dos últimas secciones
- La pregunta frente a cualquier port de UVM es siempre la misma: **¿esto puede bloquear?** La firma te lo dice antes de leer la implementación
- El **analysis port** resuelve el problema de fondo: el que publica **no sabe quién lo lee**, y agregar un lector no le toca una línea
- Es el patrón *observer* de siempre, y es el mismo movimiento que hizo la BFM en interfaces y BFM: un nivel más arriba

Un solo lugar que mira el cable

Un solo lugar que mira el cable

- Hoy el scoreboard y la cobertura miran **las señales** cada uno por su cuenta: los dos esperan `done`, los dos leen `op_set`, los dos saben el protocolo
- Eso es el mismo código escrito dos veces, y el día que el protocolo cambie hay que acordarse de los dos lugares
- La pieza que falta tiene nombre en la industria: el **monitor**. Uno solo mira el bus, arma la transacción y la publica
- El scoreboard y la cobertura pasan a ser **subscribers** de los dados de la unidad anterior: no vuelven a tocar un cable
- En la jerga UVM esos dos forman el *analysis layer*, y de ahí sale el nombre `uvm_analysis_port`

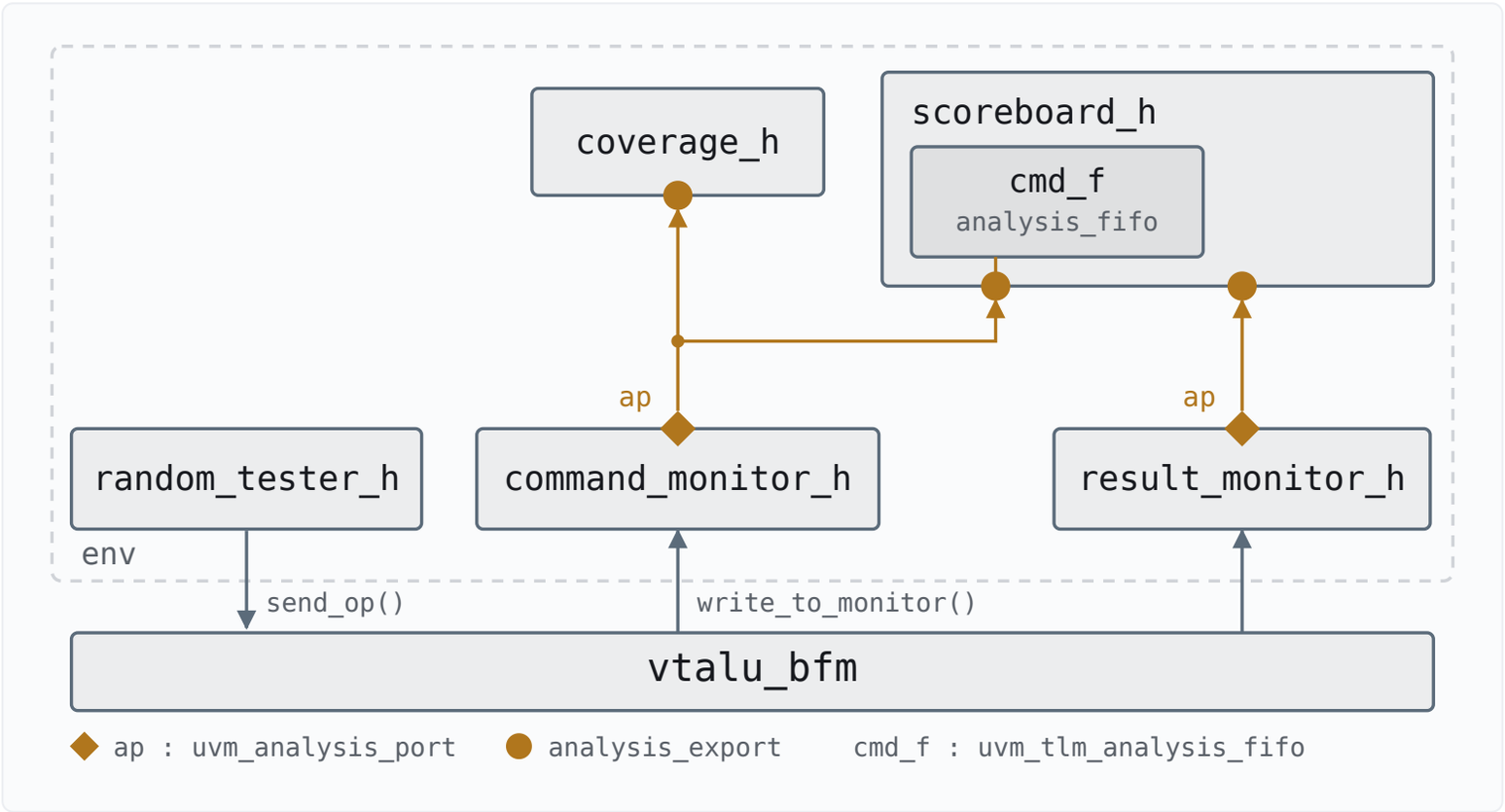
Un solo lugar que mira el cable

Qué piezas aparecen

- `command_monitor` — mira el arranque de cada operación y publica qué se pidió
- `result_monitor` — mira el `done` y publica qué contestó el DUT
- El **BFM** deja de ser sólo señales: gana tres `always` y un handle a cada monitor
- El **scoreboard** y la **cobertura** dejan de esperar flancos y pasan a tener un `write()`
- El `tester` no se entera de nada: sigue mandando estímulo igual que ayer

Un solo lugar que mira el cable

Diagrama del Testbench



Un solo lugar que mira el cable

Los handles que la BFM guarda

- Hasta ahora la clase leía la interface. Acá se da vuelta: **la interface guarda un handle a la clase** y es ella la que llama al método

```
interface vtalu_bfm;
   import vtalu_pkg::*;

   // These handles get assigned in the build_phases of the classes
   command_monitor        command_monitor_h;
   result_monitor         result_monitor_h;

   byte unsigned          A;
   byte unsigned          B;
   bit                    clk;
   bit                    reset_n;
   bit                    start;
   wire                   done;
   wire            [15:0] result;
   wire                   ovf;
   operation_t            op_set;
```

- Estos handles los setea cada monitor en su `build_phase`, justo después de sacar la BFM del config_db

```
   function void build_phase(uvm_phase phase);
      virtual vtalu_bfm bfm;

      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("COMMAND MONITOR", "Failed to get BFM")
```

```
    bfm.command_monitor_h = this;

    ap = new("ap", this);

endfunction : build_phase
```

Un solo lugar que mira el cable

Monitoreando los comandos del VTALU

- Tres `always` en el BFM: el del comando dispara cuando arranca una operación, el del reset en el flanco de bajada de `reset_n`, y el del resultado cuando el DUT levanta `done`

```
// Here is the first monitor (commands)
// A very simple FSM. If start is up, check whether this is a new command
// If it is, send it to the TB with commands_monitor.write_to_monitor()
always @(posedge clk) begin : cmd_monitor
   bit new_command;
   if (!start) new_command = 1;
   else if (new_command) begin
      command_monitor_h.write_to_monitor(A, B, op_set);
      new_command = (op_set == 3'b000);  // handle no_op
   end
end : cmd_monitor

always @(negedge reset_n) begin : rst_monitor
   if (command_monitor_h != null)  //guard against VCS time 0 negedge
      command_monitor_h.write_to_monitor(A, B, rst_op);
end : rst_monitor

always @(posedge clk) begin : rslt_monitor
   if (done) result_monitor_h.write_to_monitor(result);
end : rslt_monitor
```

- Y del otro lado, el método al que le avisan: empaqueta la `command_s` y la publica en el analysis port

```
function void write_to_monitor(byte A, byte B, bit [2:0] op);
   command_s cmd;
   cmd.A = A;
   cmd.B = B;
   cmd.op = op2enum(op);
   `uvm_info("COMMAND MONITOR", $sformatf("A:0x%2h B:0x%2h op: %s", A, B, cmd.op.name()
             ), UVM_HIGH)
   ap.write(cmd);
endfunction : write_to_monitor
```

Un solo lugar que mira el cable

VTALU Coverage Class como Subscriber

- Con el `command_monitor` hay **un solo lugar** en todo el testbench que sabe cómo se ve un comando en el cable
- El que necesite saber qué operación se ejecutó se suscribe a su analysis port e implementa `write()`. No vuelve a mirar una señal

```
// With the analysis port this is far simpler,
// because nothing here has to know about the ALU signals:
// what arrives is a command_s struct that wraps the command
//
function void write(command_s t);
      A = t.A;
      B = t.B;
      op_set = t.op;
      op_cov.sample();
      zeros_or_ones_on_ops.sample();
endfunction : write
```

Un solo lugar que mira el cable

Suscribiéndose a múltiples analysis ports

- El mecanismo básico de analysis port de UVM permite que un uvm_subscriber agarre un único analysis_port
- Sin embargo, en casos como un scoreboard, necesitamos que un mismo componente reciba datos de 2 analysis port
- El motivo del límite: `uvm_subscriber` te da **un solo** `write()`, y ése es el método que el puerto llama. Dos puertos no tienen dónde entrar
- UVM lo resuelve sin partir el componente, con la clase *uvm_tlm_analysis_fifo*, que es **ilimitada por construcción**: su constructor le pasa `size = 0` (`tlm1/uvm_tlm_fifos.svh:281`, con el comentario «analysis fifo must be unbounded»)
- Y el `write()` es una `function` que hace `void'(this.try_put(t))`: descarta el retorno. Si la FIFO estuviera acotada tiraría **sin avisar** — por eso ésta se construye sin tope
- Es paramétrica y tiene dos caras: un *analysis_export* de un lado —que se conecta como cualquier subscriber— y un `try_get()` del otro
- `try_get()` saca un elemento y devuelve 0 si la FIFO está vacía, sin bloquear
- O sea: convierte *"me avisan cuando llega"* en *"lo busco cuando lo necesito"*

Un solo lugar que mira el cable

Suscribiéndose a múltiples analysis ports

- Clase Scoreboard:

```
uvm_tlm_analysis_fifo #(command_s) cmd_f;

function void build_phase(uvm_phase phase);
   cmd_f = new("cmd_f", this);
endfunction : build_phase

// Every time the DUT hands a result to the analysis port, this method
// gets triggered
function void write(shortint t);
   shortint  predicted_result;
   command_s cmd;
   cmd.op = no_op;

   // Here is where the prediction happens: look in our FIFO for the command
   // that was sent, skipping the resets and the no_ops with the
   // do....while
   // since whenever there is a result there must be an operation,
   // a result with no operation behind it is an error
   do
   if (!cmd_f.try_get(cmd)) `uvm_fatal("SCOREBOARD", "No command in self checker")
   while ((cmd.op == no_op) || (cmd.op == rst_op));

   case (cmd.op)
      add_op: predicted_result = cmd.A + cmd.B;
      sub_op: predicted_result = cmd.A - cmd.B;
      and_op: predicted_result = cmd.A & cmd.B;
      xor_op: predicted_result = cmd.A ^ cmd.B;
      mul_op: predicted_result = cmd.A * cmd.B;
   endcase  // case (op_set)

   if (predicted_result != t)
      `uvm_error("SCOREBOARD", $sformatf(
```

```
                    "FAILED: A: %2h  B: %2h  op: %s actual result: %4h   expected: %4h",
                    cmd.A,
                    cmd.B,
                    cmd.op.name(),
                    t,
                    predicted_result
                    ))
    endfunction : write
```

Un solo lugar que mira el cable

La otra forma: `uvm_analysis_imp_decl

```
`uvm_analysis_imp_decl(_cmd)        // fabrica la clase uvm_analysis_imp_cmd
`uvm_analysis_imp_decl(_result)

class scoreboard extends uvm_component;
   uvm_analysis_imp_cmd    #(command_s, scoreboard) cmd_imp;
   uvm_analysis_imp_result #(shortint,  scoreboard) result_imp;

   function void write_cmd(command_s t);    ... endfunction   // un write por puerto
   function void write_result(shortint t);  ... endfunction
endclass
```

- La macro **genera una clase** por cada sufijo, y cada una llama a un `write_` distinto: así un componente recibe de dos puertos sin FIFO
- Es lo que vas a ver en la mayoría del código de producción, así que **hay que saber leerla**
- El curso usa la FIFO igual, por una razón de fondo: `write_cmd()` se ejecuta **cuando llega el comando**, y el scoreboard lo necesita **cuando llega el resultado**. La FIFO le da ese control; la macro lo obliga a guardarse el dato a mano

Un solo lugar que mira el cable

Suscribiendo los monitores

- Conectamos los analysis port a los monitores usando el método connect_phase() en la clase env:

```
class env extends uvm_env;
   `uvm_component_utils(env);

   random_tester   random_tester_h;
   coverage        coverage_h;
   scoreboard      scoreboard_h;
   command_monitor command_monitor_h;
   result_monitor  result_monitor_h;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      random_tester_h = random_tester::type_id::create("random_tester_h", this);
      coverage_h = coverage::type_id::create("coverage_h", this);
      scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
      command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
      result_monitor_h = result_monitor::type_id::create("result_monitor_h", this);

   endfunction : build_phase

   function void connect_phase(uvm_phase phase);

      result_monitor_h.ap.connect(scoreboard_h.analysis_export);
      command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
      command_monitor_h.ap.connect(coverage_h.analysis_export);

   endfunction : connect_phase
```

```
endclass
```

Un solo lugar que mira el cable

Cuando el DUT no contesta en orden

- El scoreboard de esta sección hace `try_get()` de una FIFO: el primer comando que entró es el que corresponde al primer resultado que salió. Eso vale porque **la VTALU contesta en orden**, una operación por vez
- Un bus real no. En AXI cada transferencia lleva un **ID** y las respuestas pueden volver cruzadas: la FIFO empieza a comparar la respuesta de una con el pedido de otra, y grita en todas

```
transaccion esperado [int];              // asociativo, indexado por lo que aparea

function void write_cmd(transaccion c);  esperado[c.id] = c;  endfunction

function void write_result(respuesta r);
   if (!esperado.exists(r.id)) `uvm_error("SB", "una respuesta que nadie pidio")
   else begin comparar(esperado[r.id], r); esperado.delete(r.id); end
endfunction
```

- El patrón es siempre el mismo: **una tabla indexada por lo que aparea**, un `delete` cuando se comparó, y al final de la corrida la tabla **tiene que quedar vacía**
- Ese chequeo del final es la mitad del valor: lo que quedó adentro son los pedidos que el DUT nunca contestó

Un solo lugar que mira el cable

De un lado del monitor se habla en señales, del otro en transactions

- El **monitor** es la única clase que sabe que existe un `clk`. Coverage y scoreboard reciben datos y no tienen idea de dónde salieron
- El Observer de hablar con varios objetos, aplicado al DUT: el monitor publica y no lleva la lista. Agregar un observador es una línea en el `connect_phase`
- Y el límite, que es la sección que sigue: todo esto pasa en **un solo thread**. El `always` del BFM llama a `write_to_monitor()`, que llama a `ap.write()`, que llama a los `write()` de los subscribers — una cadena de funciones, sin que avance el tiempo

Un solo lugar que mira el cable

Resumen de la unidad

- Hasta ayer el scoreboard y la cobertura miraban **las señales** cada uno por su cuenta: el mismo código de protocolo escrito dos veces
- La pieza que faltaba tiene nombre en la industria: el **monitor**. Uno solo mira el cable y **publica** lo que vio
- Son dos: `command_monitor` publica qué se pidió, `result_monitor` publica qué contestó el DUT
- El scoreboard y la cobertura pasan a ser **subscribers**: dejan de esperar flancos y sólo implementan `write()`
- Ese par —monitores que publican, subscribers que analizan— es el **analysis layer**, y de ahí viene el nombre del puerto
- El `tester` **no se entera de nada**. Ésa es la prueba de que el reparto quedó bien: cambiar el análisis no toca el estímulo
- Y el scoreboard con FIFO vale **mientras el DUT conteste en orden**. Cuando no, el patrón es una tabla por ID que tiene que quedar vacía al final

Cuando alguien tiene que esperar

Esto ya lo hacías, con módulos

- El nombre asusta y el mecanismo lo venís usando desde siempre: dos módulos con puertos, cada uno con su `always`, pasándose datos

```
// Sends the data through the shared variable and signals it by toggling get_it
// Then the producer blocks on the put_it signal
module producer (
    output byte shared,
    input  bit  put_it,
    output bit  get_it
);
   initial
      repeat (3) begin
         $display("Sent %0d", ++shared);
         get_it = ~get_it;
         @(put_it);
      end
endmodule : producer

// The consumer blocks on the get_it signal, waiting for data from the producer
// when the producer toggles that signal, it processes the data and toggles put_it
// when done, to unblock the producer
module consumer (
    input  byte shared,
    output bit  put_it,
    input  bit  get_it
);
   initial
      forever begin
         @(get_it);
         $display("Received: %0d", shared);
         put_it = ~put_it;
```

```
    end
endmodule : consumer
```

- Eso **es** comunicación entre threads. Lo único que falta es hacerlo entre objetos, que no tienen puertos

Cuando alguien tiene que esperar

Un objeto no tiene port list

- Lo que sí hay en SystemVerilog son handles compartidos, semáforos y mailboxes
- Con eso alcanza para resolverlo, y ése es el problema: **cada uno lo resuelve distinto**, y después nadie puede leer el testbench del de al lado
- UVM estandariza una sola forma, en dos piezas:
  - *Ports*: se instancian en un `uvm_component` para que su `run_phase()` hable con otro thread. *Put port* para mandar, *get port* para recibir
  - *TLM FIFO*: el objeto que une un put port con un get port. `new()` la crea con **tamaño 1** por defecto —`new(name, parent, size = 1)`—, y con `0` sería ilimitada. La del ejemplo usa el default
- Con tamaño 1 la FIFO no es un buffer, es un **punto de encuentro**: el que pone espera a que el otro saque

Cuando alguien tiene que esperar

Productor como un objeto

- El productor declara un `uvm_put_port` y llama a `put()`. Eso es todo: ni handshake de señales, ni bandera compartida, ni semáforo

```
// UVM solution to interthread communication
//  It has 2 parts:
//    * Ports-Objects: instantiated in our uvm_components so that
//        run_phase can talk to other threads
//    * TLM Fifos (they hold only 1 element)

class producer extends uvm_component;
   `uvm_component_utils(producer);

   int shared;
   // uvm_put_port is a parameterized class
   // this class takes care of the whole synchronization mess around
   // get_it and put_it
   uvm_put_port #(int) put_port_h;

   function void build_phase(uvm_phase phase);
      put_port_h = new("put_port_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      repeat (3) begin
         put_port_h.put(++shared);
         `uvm_info("PRODUCER", $sformatf("Sent %0d", shared), UVM_MEDIUM)
      end
```

```
      phase.drop_objection(this);
   endtask : run_phase
endclass : producer
```

Cuando alguien tiene que esperar

Consumidor como un objeto

- Del otro lado, `uvm_get_port` y un `get()`. Con la FIFO vacía, `get()` **bloquea** y el thread se queda ahí
- Cuando el productor pone un dato, el `get()` se desbloquea y sigue con el dato ya leído. La sincronización no la escribió nadie: la da la FIFO

```
class consumer extends uvm_component;
   `uvm_component_utils(consumer);

   // uvm_get_port handles all the interthread coordination
   uvm_get_port #(int) get_port_h;
   int shared;

   function void build_phase(uvm_phase phase);
      get_port_h = new("get_port_h", this);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   task run_phase(uvm_phase phase);
      forever begin
         get_port_h.get(shared);
         `uvm_info("CONSUMER", $sformatf("Received: %0d", shared), UVM_MEDIUM)
      end
   endtask : run_phase
endclass : consumer
```

Cuando alguien tiene que esperar

Instanciar los tres: dos componentes y una FIFO

- La `uvm_tlm_fifo` lleva **el mismo parámetro** que los dos ports: si no coinciden, no compila
- Los tres se instancian en el `build_phase` del test, como cualquier component. La FIFO también es un `uvm_component`
- Y el orden es el de siempre: UVM llama todos los `build_phase` de arriba hacia abajo, y recién cuando terminó arranca los `connect_phase` de abajo hacia arriba

```
class communication_test extends uvm_test;
   `uvm_component_utils(communication_test)

   producer producer_h;
   consumer consumer_h;
   uvm_tlm_fifo #(int) fifo_h;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      producer_h = new("producer_h", this);
      consumer_h = new("consumer_h", this);
      fifo_h = new("fifo_h", this);
   endfunction : build_phase
```

```
    // Connect phase: the two ports and the fifo already exist -- build_phase
    // runs first, top-down -- so this is where they get wired together.
    function void connect_phase(uvm_phase phase);
       producer_h.put_port_h.connect(fifo_h.put_export);
       consumer_h.get_port_h.connect(fifo_h.get_export);
    endfunction : connect_phase

endclass : communication_test
```

Cuando alguien tiene que esperar

Conectarlos: `put_export` y `get_export`

- Es el mismo `connect()` de los analysis ports: de un lado el **port**, del otro el **export** que provee la FIFO
- `uvm_tlm_fifo` expone doce handles; los dos que se usan son `put_export` para el que pone y `get_export` para el que saca —que es un alias de `get_peek_export`, que es el nombre que vas a ver en `print_topology()`—
- La regla que vale para todo TLM: **el port se conecta al export**, nunca dos ports entre sí

```
// Connect phase: the two ports and the fifo already exist -- build_phase
// runs first, top-down -- so this is where they get wired together.
function void connect_phase(uvm_phase phase);
   producer_h.put_port_h.connect(fifo_h.put_export);
   consumer_h.get_port_h.connect(fifo_h.get_export);
endfunction : connect_phase
```

```
UVM_INFO @ 0: reporter [RNTST] Running test communication_test...
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 1
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 1
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 2
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 2
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 3
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 3
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 0: reporter [UVM/REPORT/SERVER]
```

```
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    7
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[CONSUMER]     3
[PRODUCER]     3
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 0s
```

Cuando alguien tiene que esperar

Comunicación NO bloqueante

- Bloquear está bien mientras el que espera no tenga nada mejor que hacer. Con un reloj de por medio, quedarse esperando es perder un flanco
- Para eso están las versiones no bloqueantes: `try_put()` y `try_get()`
- Devuelven 1 si pudieron y **0 si no**, y siguen. Por eso son `function` y no `task`: no consumen tiempo

```
class consumer extends uvm_component;
   `uvm_component_utils(consumer);

   uvm_get_port #(int) get_port_h;
   virtual clk_bfm clk_bfm_i;
   int shared;

   function void build_phase(uvm_phase phase);
      get_port_h = new("get_port_h", this);
      clk_bfm_i = example_pkg::clk_bfm_i;
   endfunction : build_phase

   // Consumer and producer run on different clocks
   // The consumer has its first positive edge at 7ns
   //    and then checks the data every 14ns, because it uses bfm.clk (= 14ns)
   // The producer produces one datum every 17ns
   task run_phase(uvm_phase phase);
      forever begin
         @(posedge clk_bfm_i.clk);
         if (get_port_h.try_get(shared))
            `uvm_info("CONSUMER", $sformatf("Received: %0d", shared), UVM_MEDIUM)
      end
```

```
   endtask : run_phase

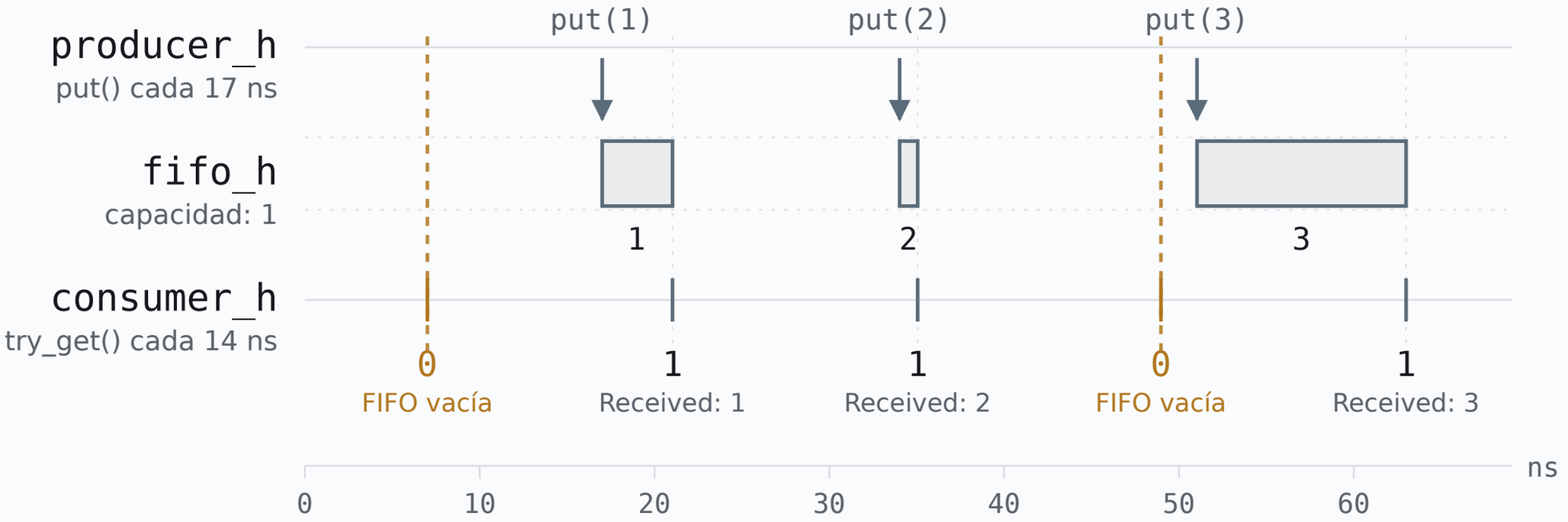
   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass : consumer
```

```
UVM_INFO @ 0: reporter [RNTST] Running test communication_test...
UVM_INFO producer.svh(18) @ 17: uvm_test_top.producer_h [PRODUCER] Sent 1
UVM_INFO consumer.svh(21) @ 21: uvm_test_top.consumer_h [CONSUMER] Received: 1
UVM_INFO producer.svh(18) @ 34: uvm_test_top.producer_h [PRODUCER] Sent 2
UVM_INFO consumer.svh(21) @ 35: uvm_test_top.consumer_h [CONSUMER] Received: 2
UVM_INFO producer.svh(18) @ 51: uvm_test_top.producer_h [PRODUCER] Sent 3
UVM_INFO consumer.svh(21) @ 63: uvm_test_top.consumer_h [CONSUMER] Received: 3
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 68: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    7
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[CONSUMER]     3
[PRODUCER]     3
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 68ns
```

Cuando alguien tiene que esperar

Comunicación NO bloqueante: la línea de tiempo

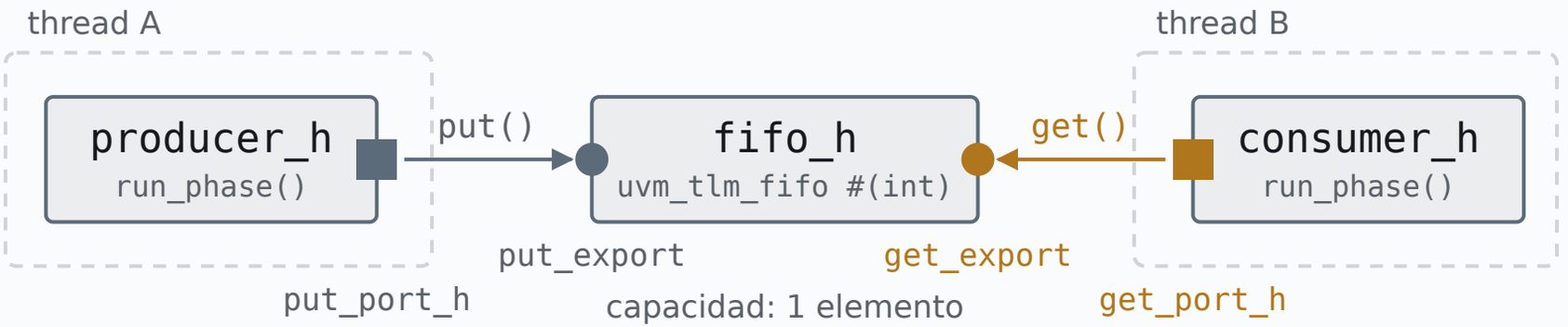


Con `get()` bloqueante, el consumer se quedaba esperando en los dos flancos en ámbar.

- El producer pone un dato cada 17 ns y el consumer mira cada 14 ns: los relojes corren distinto, y en dos flancos la FIFO está vacía
- Ahí `try_get()` devuelve 0 y el consumer sigue. Con `get()` se quedaba esperando

Cuando alguien tiene que esperar

Cómo se leen los diagramas TLM



- `get()` bloquea al `consumer_h` hasta que la FIFO tenga un dato
- `put()` bloquea al `producer_h` si la FIFO ya tiene su único elemento
- `try_get()` y `try_put()` no bloquean: devuelven 0 si no pudieron

- La convención vale para todo el material de UVM que vayas a leer después:
  - *Cuadrado:* put port / get port — el que **inicia** la llamada
  - *Círculo:* export — el que la **recibe** e implementa
  - *Rombo:* analysis port — el de las dos secciones anteriores

Cuando alguien tiene que esperar

fork: las tres formas de arrancar en paralelo

```
fork  wait_done();  count_cycles();  join       // sigue cuando terminaron LAS DOS
fork  wait_done();  count_cycles();  join_any   // sigue con LA PRIMERA; la otra sigue viva
fork  wait_done();  count_cycles();  join_none  // sigue YA; las dos quedan corriendo
```

Variante	El padre sigue…	Para qué se usa
`join`	cuando terminan **todas**	dos chequeos que tienen que cerrar los dos
`join_any`	cuando termina **la primera**	respuesta contra timeout
`join_none`	**enseguida**	lanzar threads que viven todo el test

- La diferencia entre las tres no es cómo arrancan —las tres arrancan todo a la vez— sino **cuándo sigue el que las arrancó**
- `join_none` ya lo usaron: es el `fork` de la clase `testbench` del día 2, y es lo que UVM hace sola con un `run_phase()` por componente
- La rama que quedó viva después de un `join_any` **no se muere sola**. Sigue hasta que termine — o hasta que alguien la mate, que es la slide siguiente

Cuando alguien tiene que esperar

`disable fork` y `wait fork`: apagar lo que quedó prendido

```
// El idiom de la respuesta contra el timeout. El fork de afuera AÍSLA
fork begin
   fork
      begin  wait_done();          `uvm_info("BFM", "llegó", UVM_LOW)  end
      begin  repeat (100) @(posedge clk);  `uvm_error("BFM", "timeout")   end
   join_any
   disable fork;      // mata a la hermana que perdió, y a nadie más
end join

wait fork;            // no mata a nadie: espera a TODOS los hijos de este thread
```

- `disable fork` mata **todos los procesos hijos del thread que lo ejecuta**. Por eso el `fork begin ... end join` de afuera: sin él, se lleva puesto también lo que ya estaba corriendo
- `wait fork` es lo contrario: no mata nada, **espera** a que terminen los hijos. Es lo que usa un test para no cerrar con transacciones en vuelo
- El par `join_any` + `disable fork` es el timeout de todo BFM de producción. Se escribe una vez y se copia siempre
- Gana la que termine primero: si `done` llega antes de los 100 flancos, el `uvm_error` del timeout no se imprime; si no, el `uvm_info` del `"llegó"` no se imprime. El `disable fork` mata a la que perdió

Cuando alguien tiene que esperar

⚠ *La trampa: el índice del `for` adentro del `fork`*

```
// A -- a single variable, shared by the three threads. A join_none runs
// nothing: it schedules. By the time the three run, the for is over.
for (i = 0; i < 3; i++)
  fork got = {got, $sformatf("%0d", i)}; join_none
wait fork;
check("int i outside the for, no copy", "333");

// B -- declared inside the for: one per pass, and nothing to copy.
for (int j = 0; j < 3; j++)
  fork got = {got, $sformatf("%0d", j)}; join_none
wait fork;
check("int j declared in the for", "012");

// C -- the slide's one: the automatic as the first line of a begin/end.
// Here the copy is NOT made when the fork is scheduled, so k reads the same
// i that A does and gives the same answer as A. This is the finding.
for (i = 0; i < 3; i++)
  fork begin
    automatic int k = i;
    got = {got, $sformatf("%0d", k)};
  end join_none
wait fork;
check("fork begin automatic k = i; end  (the slide's)", "333");

// D -- the LRM 9.3.2 one: the automatic declared in the fork, no begin/end.
// Same workaround, written where the LRM puts it, and this one does copy.
for (i = 0; i < 3; i++)
  fork
    automatic int k = i;
    got = {got, $sformatf("%0d", k)};
  join_none
wait fork;
check("fork automatic k = i;  no begin/end (LRM 9.3.2)", "012");
```

- Un `join_none` **no ejecuta nada todavía**: deja el thread listo y sigue. Para cuando el thread corre, el `for` ya terminó y la variable vale lo último
- Si la variable se declara **adentro** del `for`, cada vuelta tiene la suya y no hay problema. Es lo que dice el LRM y lo que Verilator hace
- Si viene de afuera —un `int` del `run_phase`, un campo de la clase— hay que **copiarla** con un `automatic`, y **dónde** se escribe la copia cambia el resultado: adentro de un `fork begin ... end` **no copia**; declarada en el `fork` mismo, sin `begin`/`end` —que es la forma del LRM 1800-2017 §9.3.2—, sí
- Medido en Verilator 5.052, y el archivo termina en `$fatal` si algún día cambia (`make repros`): las cuatro variantes imprimen `3 3 3`, `0 1 2`, **`3 3 3`** y `0 1 2`

Cuando alguien tiene que esperar

Resumen de la unidad

- La comunicación **inter**-thread entre objetos es el equivalente de lo que entre módulos hacen los puertos: pasar data de un thread a otro
- UVM la provee con *uvm_put_port*, *uvm_get_port* y *uvm_tlm_fifo*
- Cualquier objeto que quiera comunicarse con otro thread debe instanciar un port y conectarlo a una FIFO
- Ojo con el par de nombres: los *analysis port* de las dos secciones anteriores son **intra**-thread —`write()` es una `function` y corre en el thread del que publica—; esto es **inter**-thread, y por eso `put()` y `get()` son `task`
- Y abajo de todo está `fork`: `join` espera a todos, `join_any` a la primera, `join_none` a ninguno. `disable fork` mata a los hijos y `wait fork` los espera
- Ahora tenemos que usar esto para conectar nuestro TB: vamos a separar la generación de estímulo del driver de la DUT

Quién espera a quién

Qué mandar y cómo mandarlo

- Los analysis ports partieron el análisis: el monitor mira, los subscribers interpretan. Falta hacer lo mismo del lado del estímulo
- Hoy el `base_tester` hace **dos** cosas: elige la operación *y* la aplica sobre la BFM ciclo por ciclo. Son dos trabajos que cambian por motivos distintos
- El **qué** cambia con cada test. El **cómo** cambia sólo si cambia el protocolo del DUT — o sea, casi nunca
- Separarlos deja el protocolo escrito en un solo lugar: el **driver**. Cada test nuevo escribe qué mandar y no vuelve a mirar un `@(negedge clk)`
- Y como los dos corren en threads distintos, la unión es la de la comunicación entre threads: `uvm_put_port`, `uvm_get_port` y una `uvm_tlm_fifo` en el medio

Quién espera a quién

El tester hace dos trabajos, y uno no es suyo

- Así quedó el `base_tester` de los analysis ports, y hace **dos** trabajos:
  - elige qué operación mandar
  - y la aplica sobre la BFM, ciclo por ciclo
- El segundo es el que no debería estar ahí: es protocolo, y el protocolo no cambia cuando cambia el test
- La clase que se lleva ese trabajo se llama **driver**: toma un dato del testbench y lo convierte en señales

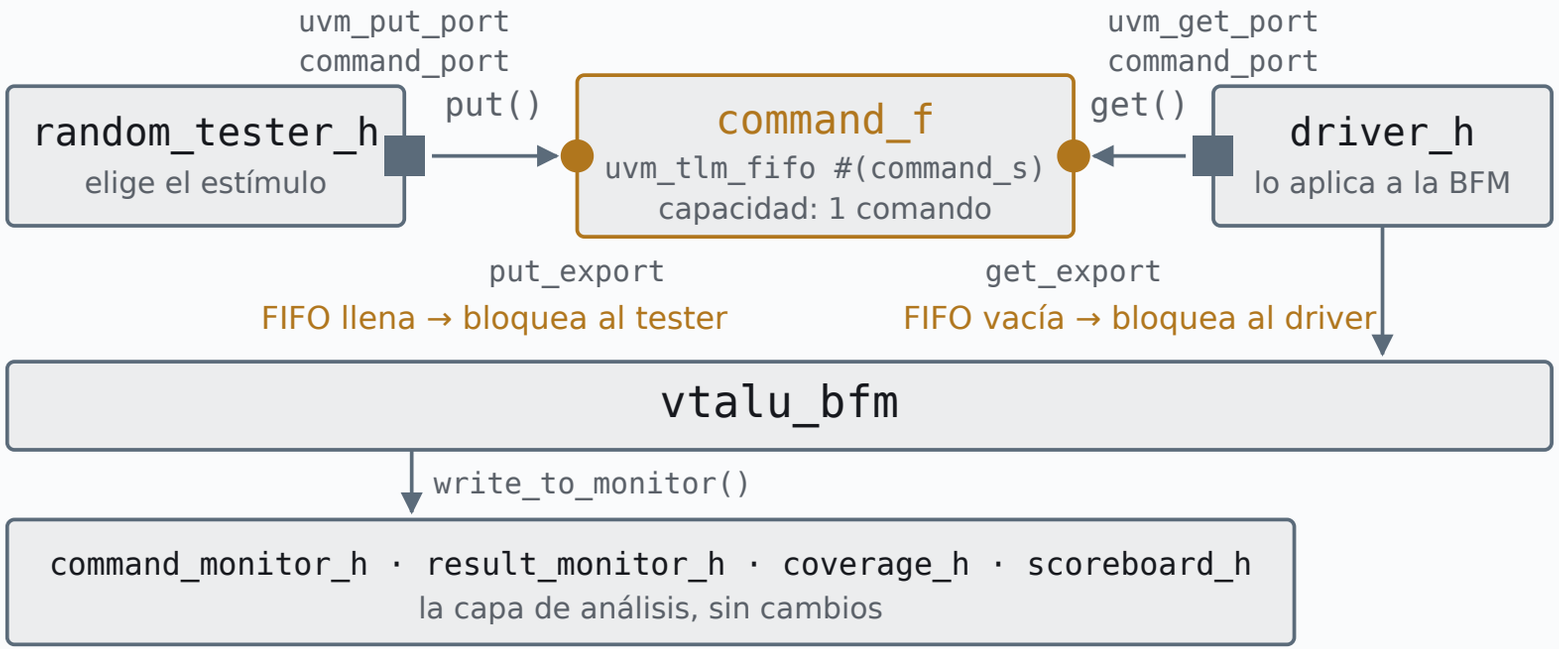
```
task run_phase(uvm_phase phase);
   byte unsigned iA;
   byte unsigned iB;
   operation_t   op_set;

   phase.raise_objection(this);
   bfm.reset_alu();
   repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set);
   end : random_loop
   #500;
   phase.drop_objection(this);
endtask : run_phase
```

Quién espera a quién

Cómo queda partido

- El resultado: una clase decide **qué** mandar y otra sabe **cómo** mandarlo, y entre las dos hay una FIFO en vez de una llamada a método


uvm_put_port
command_port
uvm_get_port
command_port
random_tester_h
elige el estímulo
put()
command_f
uvm_tlm_fifo #(command_s)
capacidad: 1 comando
get()
driver_h
lo aplica a la BFM
put_export
FIFO llena → bloquea al tester
get_export
FIFO vacía → bloquea al driver
vtalu_bfm
write_to_monitor()
command_monitor_h · result_monitor_h · coverage_h · scoreboard_h
la capa de análisis, sin cambios

Quién espera a quién

El env: siete objetos y cinco `connect()`

- Mantra: "Ports connect to exports"

```
function void build_phase(uvm_phase phase);
   command_f = new("command_f", this);
   random_tester_h = random_tester::type_id::create("random_tester_h", this);
   driver_h = driver::type_id::create("driver_h", this);

   coverage_h = coverage::type_id::create("coverage_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
   result_monitor_h = result_monitor::type_id::create("result_monitor_h", this);
endfunction : build_phase

function void connect_phase(uvm_phase phase);
   // PORTS CONNECT TO EXPORTS!
   driver_h.command_port.connect(command_f.get_export);
   random_tester_h.command_port.connect(command_f.put_export);

   result_monitor_h.ap.connect(scoreboard_h.analysis_export);

   command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
   command_monitor_h.ap.connect(coverage_h.analysis_export);
endfunction : connect_phase
```

Quién espera a quién

El `base_tester`, sin una sola señal

- Cambia una sola cosa: donde había un handle a la BFM ahora hay un `uvm_put_port #(command_s)` llamado `command_port`
- `random_tester` y `add_tester` **no se tocan**. Sólo definen `get_op()` y `get_data()`, y ninguna de las dos sabía cómo se aplicaba el estímulo
- Ése es el retorno de haber separado bien en el env: el cambio se detuvo en la clase base

```
virtual class base_tester extends uvm_component;

   `uvm_component_utils(base_tester)

   // the handle to the BFM is replaced by a uvm_put_port
   uvm_put_port #(command_s) command_port;

   function void build_phase(uvm_phase phase);
      command_port = new("command_port", this);
   endfunction : build_phase

   pure virtual function operation_t get_op();

   pure virtual function byte get_data();

   task run_phase(uvm_phase phase);
      byte unsigned iA;
      byte unsigned iB;
      operation_t   op_set;
```

```
    command_s     command;

    phase.raise_objection(this);
    command.op = rst_op;
    command_port.put(command);
    repeat (1000) begin : random_loop
       command.op = get_op();
       command.A = get_data();
       command.B = get_data();
       command_port.put(command);
    end : random_loop
    #500;
    phase.drop_objection(this);
  endtask : run_phase
```

Quién espera a quién

El driver: el único que toca el cable

- Un `forever` con dos líneas: `get()` un comando —que bloquea hasta que haya— y aplicarlo con `send_op()`. Todo el protocolo vive acá

```
class driver extends uvm_component;
   `uvm_component_utils(driver)

   virtual vtalu_bfm bfm;

   uvm_get_port #(command_s) command_port;

   function void build_phase(uvm_phase phase);
      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("DRIVER", "Failed to get BFM")
      command_port = new("command_port", this);
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      command_s command;

      forever begin : command_loop
         // Blocking: if there are no commands to send, it waits
         command_port.get(command);
         bfm.send_op(command.A, command.B, command.op);
      end : command_loop
   endtask : run_phase
```

Quién espera a quién

Resumen de la unidad

- El testbench quedó partido en tres capas que cambian por motivos distintos: el **estímulo** (tester), el **protocolo** (driver y monitores) y el **análisis** (scoreboard y cobertura)
- Ninguna conoce el interior de las otras: entre ellas sólo viajan comandos y resultados
- Falta que el testbench sepa **contar lo que pasa**, que es la unidad que sigue

Repaso · Día 4

1 de 5 · Observer Pattern

En el patrón Observer, ¿qué sabe el objeto observado sobre sus observadores?

a Cuántos son y de qué tipo

b Sólo el primero que se suscribió

c Los conoce porque se los pasan en el constructor, uno por uno

d Nada: ni cuántos son, ni quiénes son ✓

No sabe nada — y esa ignorancia es la gracia. Agregar un cuarto subscriber no obliga a tocar una línea del que publica.

Repaso · Día 4

2 de 5 · Analysis Ports

Un `uvm_subscriber` necesita datos de dos analysis ports distintos. ¿Cómo se resuelve?

a. Implementando dos veces el método `write()`

b. Con una `uvm_tlm_analysis_fifo` para el segundo puerto ✓

c. Registrando el componente dos veces en la factory, una por puerto

d. Conectando los dos puertos al mismo `analysis_export`

Con una `uvm_tlm_analysis_fifo` — un `uvm_subscriber` tiene un solo `write()`, así que sólo puede escuchar un puerto. La FIFO da un `analysis_export` de un lado y un `try_get()` del otro. Es lo que hace el scoreboard del VTALU. La otra forma es `` `uvm_analysis_imp_decl ``, que fabrica un `write_` por sufijo: no son dos `write()` en la misma clase, que SystemVerilog no permite.

Repaso · Día 4

3 de 5 · Intra vs. inter thread

Cuando el monitor publica y se ejecutan los `write()` de los subscribers, ¿cuántos threads intervienen?

a. Uno por subscriber

b. Dos: el del publicador y el del suscriptor

c. Uno solo: `write()` es una llamada a función ✓

d. Uno por subscriber más el del monitor, y UVM los sincroniza al final del delta

Uno solo, y es comunicación intra-thread — `write()` es una `function`, no una `task`: no consume tiempo y corre en el thread del que publica. Justamente por eso hace falta **otro** mecanismo (put/get + FIFO) para hablar entre threads.

Repaso · Día 4

4 de 5 · Put y get ports

El consumidor llama a `get()` y la `uvm_tlm_fifo` está vacía. ¿Qué pasa?

- **a** Se bloquea hasta que el productor ponga un dato ✓
- **b** Devuelve 0 y sigue
- **c** Error fatal de UVM
- **d** Devuelve el último dato leído, que sigue en la FIFO hasta que lo pisen

Se bloquea — `get()` es bloqueante, y por eso se declara `task` y no `function`. Esa es toda la sincronización: no hace falta un handshake de señales a mano.

Repaso · Día 4

5 de 5 · try_get()

¿Y `try_get()` con la FIFO vacía?

- a — Se bloquea igual que `get()`
- b — Devuelve 1 con un dato basura
- c — Espera un ciclo de reloj y reintenta, hasta el timeout de la fase
- **d — Devuelve 0 inmediatamente, sin bloquear** ✓

Devuelve 0 y sigue — es la versión no bloqueante, y por eso puede ser una `function`. Fijate que el scoreboard la usa en un `do ... while` para saltear los `no_op` y los `rst_op`.

Ejercicio · Día 4 · 1 de 2

Un observador más, sin tocar a los que ya miran

```
$ cd code/ejercicios/d4 && bash run.sh
```

- Un `uvm_subscriber #(command_s)` que cuente los comandos
- Instancialo y conectalo en el `env`, sin tocar las conexiones que ya están
- Se corrige cruzado: tu cuenta tiene que dar igual que la del `command_monitor`

Ejercicio · Día 4 · 2 de 2

El #500 es un parche

```
$ cd code/ejercicios/d4b && bash run.sh
```

- El testbench de *put y get* con **una línea cambiada**: la FIFO del `env` viene sin tope
- El tester vacía sus mil comandos en `t = 0`, espera el `#500` de siempre, baja la objection — y la simulación termina **en verde con 13 comandos en el bus**
- Se arregla en el driver: una objection **mientras hay un comando en vuelo**
- `env.svh` no se toca: la FIFO sin tope es el DUT de este ejercicio

Ayer quedó…

Dónde dejamos el testbench

- El monitor **publica** y no conoce a nadie: cuelgan de él el scoreboard y la cobertura, y agregar un tercero no toca al que publica
- Las FIFO de TLM resolvieron quién espera a quién, y `fork`/`join` quién corre con quién
- Pero el dato sigue siendo **tres campos sueltos**: `A`, `B` y `op` viajan separados, se copian a mano, y compararlos es escribir la comparación entera

Agenda

Día 5 · ≈ 4 h 30 · unidad 6 · el dato

- Copiar un objeto que contiene otro
- Transactions
- Constrained random

Al final del día podés:

- **Escribir** una constraint que llegue a un caso de borde, y **medir** si el `dist` hace lo que dice
- **Cerrar** un bin que quedó abierto, con un caso dirigido pedido por `randomize() with {}`
- **Explicar** la diferencia entre copiar el handle y copiar el objeto, y qué pasa cuando el objeto contiene otro

Copiar un objeto que contiene otro

El handle no es el objeto

- Desde las transactions los datos del testbench dejan de ser una `struct` y pasan a ser **objetos que viajan**: el monitor los crea, el scoreboard los compara
- Y ahí aparece un problema que una `struct` no tenía. Una `struct` se copia al asignarla; un objeto **no**: `obj1_h = obj2_h` copia el handle, no los datos
- O sea que dos partes del testbench pueden estar mirando el mismo objeto sin saberlo. Si una lo modifica, la otra ve el cambio a mitad de camino
- Copiarlo de verdad hay que escribirlo, y escribirlo **en cada nivel de la jerarquía**, o los campos de arriba se pierden en silencio
- Esta sección es la OOP que falta para que las transactions se puedan copiar, comparar e imprimir sin mentir

Copiar un objeto que contiene otro

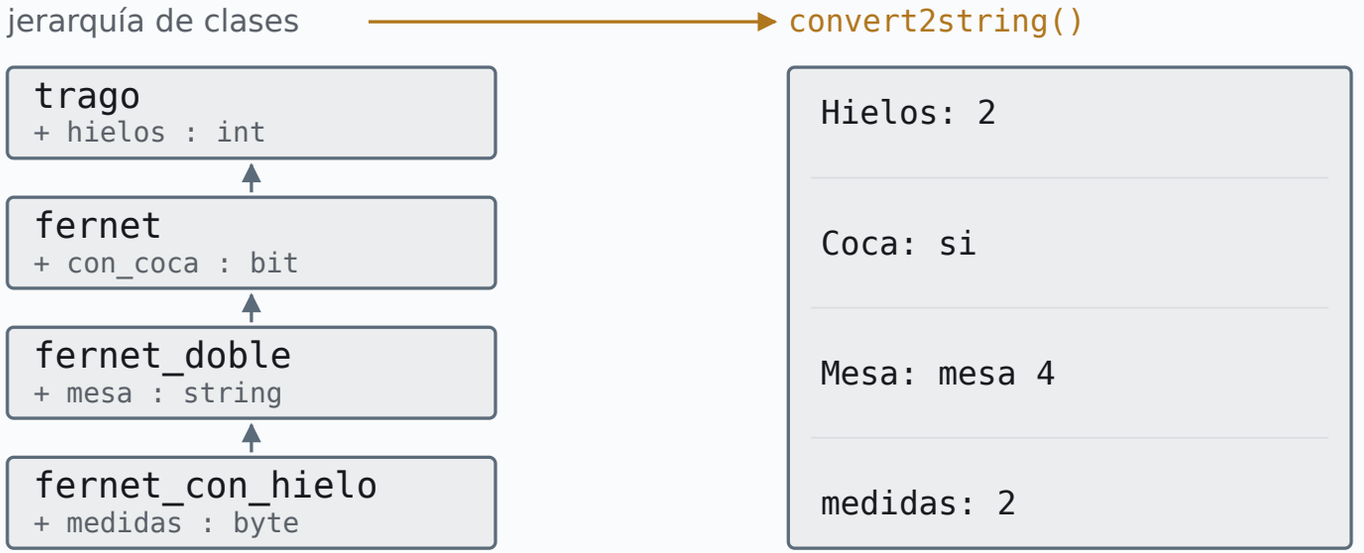
Nunca dupliques código

> *“Si nos encontramos copiando código y haciéndole pequeñas modificaciones dentro de nuestro programa, estamos haciendo las cosas mal. Nunca hay que duplicar código.”*

En OOP en vez de copiar código...

Copiar un objeto que contiene otro

convert2string(): el problema



una línea por clase —
mientras nadie toque la jerarquía

Cada clase agrega un campo y convert2string() los imprime todos, de lo más general a lo más partic

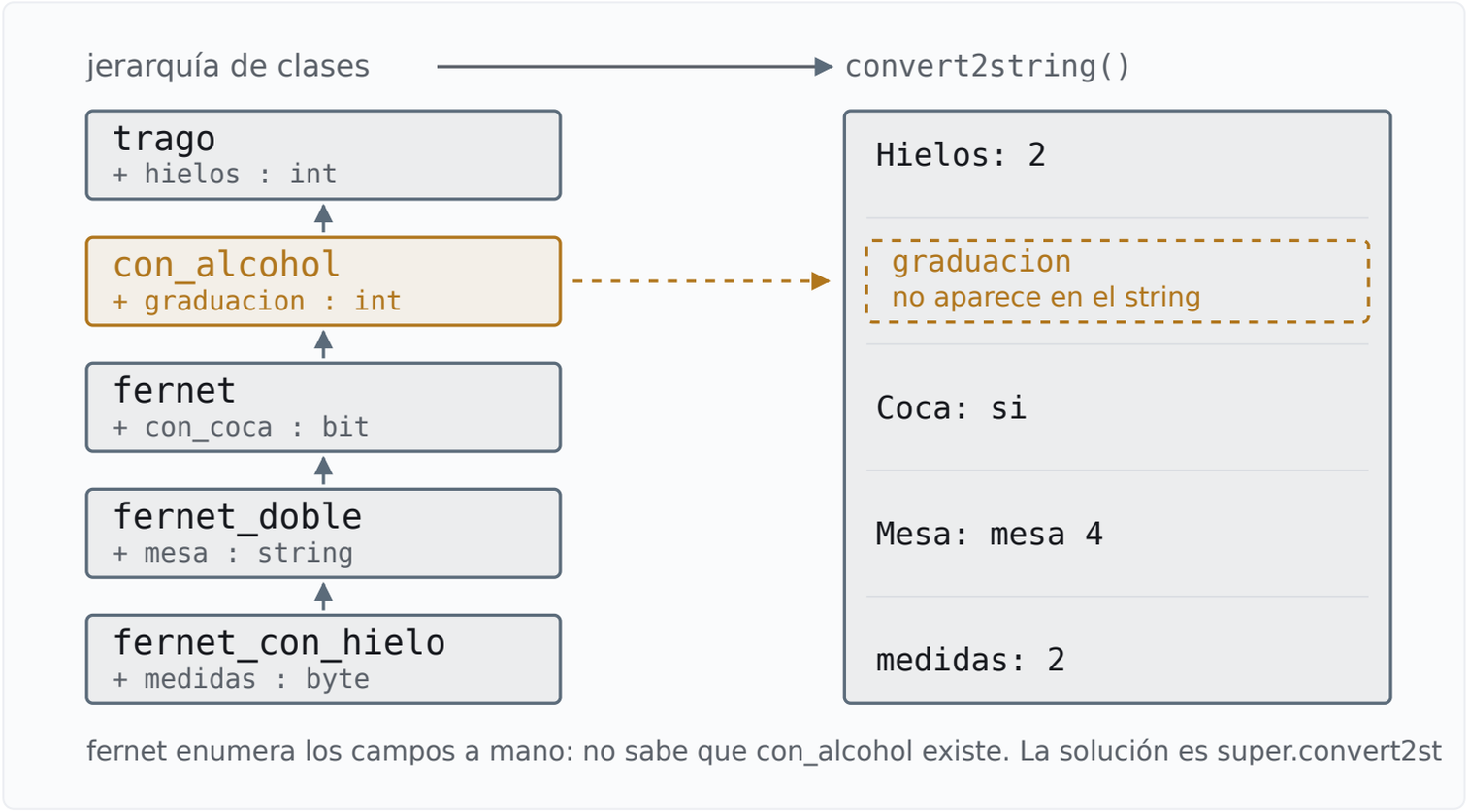
- La jerarquía va de lo general a lo particular, y cada clase agrega campos propios
- Queremos dos cosas de ella: poder **copiar** un objeto entero e **imprimirlo** entero, sin que se pierda nada de los niveles de arriba
- Esta primera versión funciona. Contá cuántas veces aparece el nombre de un campo de la clase base

```
// ILLUSTRATION -- not compiled. The WRONG way to write convert2string!
// The classes are drawn as comments: only the methods are on screen.
//class trago;
   virtual function string convert2string();
      return $sformatf("Hielos: %0d", hielos);
   endfunction : convert2string
//class fernet extends trago;
   function string convert2string();
      string coca_s;
      coca_s = (con_coca) ? "si" : "no";
      return $sformatf("Hielos: %0d  Coca: %s", hielos, coca_s);
   endfunction : convert2string

//class fernet_doble extends fernet;
   function string convert2string();
      return {$sformatf("hielos: %0d  coca: %0b  mesa: %s",
                  hielos, con_coca, mesa) };
   endfunction : convert2string
```

Copiar un objeto que contiene otro

convert2string(): la versión deep



fernet enumera los campos a mano: no sabe que con_alcohol existe. La solución es super.convert2st

- El día que alguien mete una clase **en el medio**, los de abajo siguen imprimiendo lo que sabían y la del medio no aparece. Y no hay error: el método sigue devolviendo un string
- La regla que lo arregla: cada clase imprime **sólo lo suyo** y le pide el resto a `super`
- Nadie sabe qué hay más arriba, así que insertar un nivel no rompe nada

```
// ILLUSTRATION -- not compiled. The same three methods written deep, with the
// classes drawn as comments. The version that does compile is deep.sv.
//virtual class trago;
   virtual function string convert2string();
      return $sformatf("Hielos: %0d", hielos);
   endfunction : convert2string

//class con_alcohol extends trago;

   function string convert2string();
      return {super.convert2string(),
              $sformatf("\ngraduacion: %0d",graduacion)};
   endfunction : convert2string

//class fernet extends con_alcohol;

   function string convert2string();
      string coca_s;
      coca_s = (con_coca) ? "si" : "no";
      return {super.convert2string(), "\nCoca: ",coca_s};
   endfunction : convert2string
```

Copiar un objeto que contiene otro

El signo igual no copia nada

- `obj1_h = obj2_h` **no copia nada**: deja dos handles apuntando al mismo objeto
- Si lo modificás por uno, el otro lo ve. No hay aviso, no hay warning
- Con una `struct` eso no pasaba: se copia al asignarla. Es la diferencia que hay que tener presente desde acá hasta el final del curso

Copiar un objeto que contiene otro

Copiar de verdad son dos pasos

Copiar de verdad son dos pasos, y hay que escribirlos: **instanciar** un objeto nuevo y **pasarle los datos** campo por campo

```
// ILLUSTRATION -- not compiled. The initial block of deep.sv:110-118, on its own.
initial begin
   fernet_hielo1_h =
       new(.hielos(2), .con_coca(1), .graduacion(2), .mesa("mesa 4"), .medidas(2));
   $display("\n--- Fernet 1 ---\n", fernet_hielo1_h.convert2string());
   fernet_hielo2_h = new();
   $display("\n--- Fernet 2 before the copy ---\n", fernet_hielo2_h.convert2string());
   // do_copy is a method that copies member by member
   // it suffers from the same problem already seen with convert2string()
   // Each class can copy the variables defined in that class
   // super.do_copy() has to be used to copy the inherited ones
   fernet_hielo2_h.do_copy(fernet_hielo1_h);
   $display("\n--- Fernet 2 after the copy ---\n", fernet_hielo2_h.convert2string());
end
```

- El `new()` es el primer paso y no es opcional: sin un objeto nuevo del otro lado, no hay adónde copiar
- El segundo paso es a mano, campo por campo. SystemVerilog no tiene una copia profunda incorporada
- Y campo por campo quiere decir **también los de las clases de arriba**, que es justo lo que se olvida

Copiar un objeto que contiene otro

do_copy(): cada clase toca lo suyo

do_copy(): Cada clase en la jerarquía, debe llamar a su clase superior, por lo tanto, todos los métodos do_copy() necesitan el mismo tipo de argumento. Pero tenemos clases distintas... Usamos *polimorfismo*

```
class fernet_con_hielo extends fernet_doble;
   byte medidas;

   function new(int hielos = 0, bit con_coca = 0, int graduacion = 0,
                string mesa = "", byte medidas = 0);
      super.new(hielos, graduacion, con_coca, mesa);
      this.medidas = medidas;
   endfunction : new

   function string convert2string();
      return {super.convert2string(), "\n", $sformatf("medidas: %0d", medidas)};
   endfunction : convert2string

   //do_copy passes the argument up to its parent class
   function void do_copy(trago copia);
      fernet_con_hielo copia_fernet_con_hielo;
      super.do_copy(copia);
      // Then the copy gets cast to a variable of its own class type
      $cast(copia_fernet_con_hielo, copia);
      this.medidas = copia_fernet_con_hielo.medidas;
   endfunction : do_copy

endclass : fernet_con_hielo
```

Copiar un objeto que contiene otro

Resumen: la solución pésima

```
function string convert2string();
   return {
      $sformatf(
          "hielos: %0d  coca: %0b  mesa: %s  medidas: %0d",
          hielos,
          con_coca,
          mesa,
          medidas
      )
   };
endfunction : convert2string
```

- Es el `convert2string()` de `fernet_con_hielo`, la clase de más abajo: imprime **los cuatro campos a mano**, incluidos los tres que heredó
- Funciona, y por eso es peligroso. El día que `fernet` gane un campo, este método sigue compilando y sigue imprimiendo — de menos
- El archivo entero está en `code/u6/jerarquias/wrong.sv`. Los tres `do_copy()` de ahí sí llaman a `super`; la que tiene el vicio es `bad_copy()`, abajo de todo

Copiar un objeto que contiene otro

Resumen: la solución deep

```
function string convert2string();
   string coca_s;
   coca_s = (con_coca) ? "si" : "no";
   return {super.convert2string(), "\nCoca: ", coca_s};
endfunction : convert2string

function void do_copy(trago copia);
   fernet copia_fernet;
   super.do_copy(copia);
   $cast(copia_fernet, copia);
   this.con_coca = copia_fernet.con_coca;
endfunction : do_copy
```

- Los mismos dos métodos, escritos como corresponde: cada uno llama a `super` y después toca **un solo campo**, el suyo
- `convert2string()` **concatena** lo que devolvió el de arriba; `do_copy()` **llama primero** al de arriba y después copia lo propio
- Con esa forma, meter una clase nueva en el medio de la jerarquía funciona sola: la cadena de `super` la incluye sin que nadie edite nada

Copiar un objeto que contiene otro

Resumen de la unidad

- `obj1_h = obj2_h` **no copia nada**: deja dos handles apuntando al mismo objeto. Si lo tocás por uno, el otro lo ve, y no hay ni un warning
- Con una `struct` eso no pasaba —se copia al asignarla—, y ésa es la diferencia que hay que tener presente hasta el final del curso
- **Copiar de verdad son dos pasos**: instanciar un objeto nuevo y pasarle los datos. De ahí sale el `clone()` de las transactions
- En una jerarquía, cada clase debe tocar **un solo campo, el suyo**, y llamar a `super`. Escribir los cuatro campos a mano es la solución pésima
- Para que eso funcione, los `do_copy()` / `do_compare()` toman todos **el mismo tipo de argumento**, la clase base. Es polimorfismo, otra vez
- El día que alguien mete una clase **en el medio** de la jerarquía, la versión deep sigue andando sola. La otra hay que salir a corregirla

Transactions

El testbench está bien repartido, y el dato no

- El testbench ya está bien repartido, pero **los datos siguen siendo una `struct`**: `command_s` para los comandos y una `shortint` pelada para los resultados
- Una `struct` no hace nada por sí sola. Imprimir un comando es un `$sformatf` escrito a mano, y está escrito **tres veces**: los dos monitores y el scoreboard
- Lo mismo con randomizar y con comparar: cada componente se arma el suyo, y el día que la `struct` gana un campo hay que acordarse de los tres
- Y el resultado ni siquiera es una `struct`: es un `shortint` **pelado**, así que el `ovf` del VTALU —la segunda salida— no tiene dónde viajar
- Una clase, en cambio, tiene métodos: los datos y lo que se hace con ellos viven juntos, y el resto del testbench se achica

Transactions

Una transaction es un dato con los métodos que operan sobre ese dato

- Es la palabra que usa la industria, y no hay ninguna clase mágica atrás: una **transaction** es eso y nada más
- Los métodos son cuatro, y son siempre los mismos:
  - `randomize()` — se llena sola, respetando sus `constraint`
  - `convert2string()` — se imprime sola
  - `do_copy()` — se copia sola
  - `do_compare()` — se compara sola
- Lo que cambia no es el dato: es **quién sabe cosas sobre él**. El `tester` dejaba de saber qué valores son legales; ahora sólo dice *"randomizate"*
- Y el resto del testbench se achica: los tres `$sformatf` escritos a mano se vuelven una llamada a `convert2string()`

Transactions

Definir una transaction: de qué clase se extiende

- Las transactions se definen extendiendo la clase base *uvm_sequence_item* y escribiendo los siguientes métodos:
  - do_copy()
  - do_compare()
  - convert2string()
- Lo que sigue es transformar la vieja estructura `command_s` en una clase, `command_transaction`, método por método

```
// ILLUSTRATION -- not compiled. The struct as it is in code/u5/put-get/vtalu_pkg.sv,
// the one this section turns into a class.
typedef struct {
   byte unsigned A;
   byte unsigned B;
   operation_t   op;
} command_s;
```

Transactions

Los campos randomizados: el "antes"

- Hoy el estímulo se arma con dos funciones escritas a mano, `get_op()` y `get_data()`, que sortean los valores con `$random` — el de Verilog-95, con semilla global
- El rango legal de `A` y `B` y el sesgo a los bordes son propiedades **del dato**, no del test — y sin embargo viven en el tester
- SystemVerilog ya trae esto hecho, y no hace falta escribir ni un `$urandom`

```
// ILLUSTRATION -- not compiled. The two functions as they are in
// code/u5/put-get/tb_classes/random_tester.svh, the section this one replaces.
//
// This function handles the fact that
// the operation bus has 8 possible values,
// but only 6 of them are valid operations
virtual function operation_t get_op();
  bit [2:0] op_choice;
  op_choice = $random;
  case (op_choice)
          3'b000 : return no_op;
          3'b001 : return add_op;
          3'b010 : return sub_op;
          3'b011 : return and_op;
          3'b100 : return xor_op;
          3'b101 : return mul_op;
          3'b110 : return rst_op;
          3'b111 : return rst_op;
  endcase // case (op_choice)
```

```
endfunction : get_op

// This function was meant to spread out
// the probability of getting all zeros (prob = 1/3),
// biasing the randomness a little so the prob is not 1/256,
// since that would make our coverage goal hard to reach
virtual    function byte get_data();
  bit [1:0]   zero_ones;
  zero_ones = $random;
  if (zero_ones == 2'b00)
    return 8'h00;
  else if (zero_ones == 2'b11)
    return 8'hFF;
  else
    return $random;
endfunction : get_data
```

Transactions

Los campos randomizados: el "después"

- Toda clase de SystemVerilog trae un `randomize()` implícito, que elige valores para los campos marcados `rand`
- `get_op()` desaparece: un `enum` se randomiza solo sobre sus valores legales
- `get_data()` también, y lo reemplaza una `constraint` con `dist`, que es donde se declara el sesgo a los bordes

```
class command_transaction extends uvm_sequence_item;
   `uvm_object_utils(command_transaction)
   rand byte unsigned A;
   rand byte unsigned B;
   rand operation_t   op;

   // :/ and NOT :=  ---  this is the line that reproduces the bias towards the
   // corner cases that get_data() of the Functional coverage section did by hand (1/4 at 00, 1/4 at FF,
   // 1/2 in the middle). With ":=" the weight applies to EVERY value of the range, so the
   // middle takes 254 out of 256 and 00 comes up once every 256: the distribution
   // ends up almost uniform and the corner bins never fill.
   // The Constrained Random unit has the experiment with the numbers.
   constraint data {
      A dist {
         8'h00 :/ 1,
         [8'h01 : 8'hFE] :/ 2,
         8'hFF :/ 1
      };
      B dist {
         8'h00 :/ 1,
         [8'h01 : 8'hFE] :/ 2,
         8'hFF :/ 1
```

```
    };
}
```

Transactions

El constructor: un `uvm_object` no tiene padre

- *uvm_sequence_item* extiende *uvm_transaction*, y ésta *uvm_object* — o sea que una transaction **no** es un *uvm_component*. Por lo tanto, tiene un constructor más simple
- Una transaction **no vive en el árbol del testbench**: ese árbol lo forman los *uvm_components*, y es el que UVM recorre en `build_phase`. Sin lugar en el árbol no hay padre, y por eso el constructor sólo pide un `name`
- Los objetos se crean y se destruyen todo el tiempo mientras la simulación corre; los componentes se construyen una vez, antes de arrancar, y se quedan
- Igual conviene darles nombre: es el que aparece en los mensajes de UVM

```
// Constructor of a uvm_object: dead simple, it only takes a name. A
// transaction has no parent because it does not live in the component tree.
function new(string name = "");
   super.new(name);
endfunction : new
```

Transactions

do_copy(): copiar el objeto, no el handle

- `uvm_object` trae dos métodos hechos: `copy()`, que vuelca los datos sobre otro objeto **que ya existe**, y `clone()`, que además crea el objeto nuevo
- Los dos funcionan sólo si la clase implementa `do_copy()`: la librería sabe copiar, pero no sabe **qué campos** tiene tu transaction
- Es el mismo patrón que el resto de la sección: `copy()` es el que se llama, `do_copy()` es el que se escribe
- La única diferencia es el argumento: tiene que ser un `uvm_object` —de ahí el `$cast`—. UVM lo llama *rhs* (*right hand side*) por convención, no por requisito

```
function void do_copy(uvm_object rhs);
   command_transaction copied_transaction_h;

   // Checks that make DEBUG easier
   if (rhs == null)
      `uvm_fatal("COMMAND TRANSACTION", "Tried to copy from a null pointer")

   if (!$cast(copied_transaction_h, rhs))
      `uvm_fatal("COMMAND TRANSACTION", "Tried to copy wrong type.")

   super.do_copy(rhs);  // copy all parent class data

   A = copied_transaction_h.A;
   B = copied_transaction_h.B;
```

```
    op = copied_transaction_h.op;

  endfunction : do_copy
```

Transactions

La regla MOOCOW: compartir sí, modificar no

- Varios componentes pueden ver el mismo dato si tienen handles al mismo objeto. Es barato y funciona — mientras nadie lo modifique
- La regla que lo sostiene se llama **MOOCOW** —*Manual Obligatory Object Copy On Write*—, y así la bautiza *The UVM Primer*, de Ray Salemi
- Dice esto: compartir un handle está bien **siempre que no toques el objeto**. El que quiere modificar, copia primero
- Es una disciplina del equipo, no algo que el lenguaje imponga. UVM sólo pone la herramienta: `clone()`, que devuelve un `uvm_object` y hay que castear

Transactions

`clone_me()`: el `$cast` escrito una sola vez

- Como `clone()` devuelve un `uvm_object`, cada componente que lo use termina escribiendo su propio `$cast` — el mismo, repetido
- La convención del *UVM Primer* es agregarle a la transaction un `clone_me()` que haga las dos cosas y devuelva ya el tipo correcto
- No lo trae UVM: es cuatro líneas escritas una vez en el dato, para que ninguno de los que lo usan tenga que acordarse del cast

```
function command_transaction clone_me();
   command_transaction clone;
   uvm_object tmp;

   tmp = this.clone();
   $cast(clone, tmp);
   return clone;
endfunction : clone_me
```

Transactions

`do_compare()`: comparar dos objetos sin escribir un `if` por campo

- `compare()` lo trae `uvm_object` y devuelve 1 si los dos objetos son iguales. Es el que se llama; el que se **escribe** es `do_compare()`
- Toma dos argumentos: el `rhs` —el otro objeto— y un `uvm_comparer`, que es la política de comparación: cuántas diferencias reportar y con qué verbosidad
- Casi nadie toca el comparer, pero hay que declararlo porque la firma lo pide

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
   command_transaction compared_transaction_h;
   bit same;

   if (rhs == null)
      `uvm_fatal("RANDOM TRANSACTION", "Tried to do comparison to a null pointer");

   // First check whether we are comparing the same type
   if (!$cast(compared_transaction_h, rhs)) same = 0;
   else
      // Deep comparison: what the parent compares, plus the three fields of
      // this class
      same = super.do_compare(
          rhs, comparer
      ) && (compared_transaction_h.A == A) && (compared_transaction_h.B == B) &&
          (compared_transaction_h.op == op);

   return same;
endfunction : do_compare
```

Transactions

`convert2string()`: la transaction se imprime sola

- Es el método que devuelve el objeto como texto. Se escribe **una vez**, y lo usan el scoreboard, el monitor y cualquiera que tenga que reportar
- `$sformatf()` arma el string con los mismos especificadores de formato de siempre
- Y el detalle que hace legible un log: un `enum` tiene `name()`, que devuelve `add_op` en vez de `3'b001`. Sin eso el error dice un número y nadie lo lee

```
function string convert2string();
   string s;
   // op.name() can be add_op, no_op ...
   s = $sformatf("A: %2h  B: %2h op: %s", A, B, op.name());
   return s;
endfunction : convert2string
```

Transactions

Las field macros: leerlas aunque no las escribas

```
class command_transaction extends uvm_sequence_item;
   `uvm_object_utils_begin(command_transaction)
      `uvm_field_int (A,  UVM_DEFAULT | UVM_HEX)   // un campo integral
      `uvm_field_enum(operation_t, op, UVM_DEFAULT)
      `uvm_field_object(payload, UVM_DEFAULT | UVM_REFERENCE)  // handle, no copia
   `uvm_object_utils_end
endclass
```

- Es lo que vas a encontrar en **todo VIP** que abras. Generan `do_copy`, `do_compare`, `do_print`, `do_pack` y la configuración automática, de una declaración por campo
- El segundo argumento es una máscara: `UVM_DEFAULT` prende todas las operaciones, y `UVM_NOCOMPARE` / `UVM_NOPRINT` apagan una. El radix (`UVM_HEX`, `UVM_DEC`) es sólo para imprimir
- `UVM_REFERENCE` en un `uvm_field_object` copia **el handle**; sin él, la copia es profunda. Es la misma decisión de `do_copy()`, tomada en una constante
- El curso los escribe a mano por lo que ya dijimos: eso se debuggea, esto no. Pero leerlas hay que saber

Transactions

Los siete pasos, y por qué son siete

Lo que cuesta cambiarle el tipo de dato a un TB que ya existe. Vemos **2, 3, 6 y 7**; **1, 4 y 5** son traducción directa y están en el repo:

#	Qué	Dónde
1	una `result_transaction` para la vuelta	`result_transaction.svh`
2	una `add_transaction`, sólo sumas	`add_transaction.svh`
3	los dos testers se funden en uno	`tester.svh`
4	el `command_monitor` publica objetos	`command_monitor.svh`
5	el `result_monitor`, lo mismo	`result_monitor.svh`
6	el `scoreboard` usa `compare()`	`scoreboard.svh`
7	`add_test` overridea el **dato**	`add_test.svh`

Transactions

Paso 2 · una `add_transaction` que sólo suma

- `add_transaction` extiende `command_transaction` y le agrega **una sola** constraint: `op == add_op`
- Con eso alcanza para cambiar el estímulo, y el `tester_h` no se entera: usa la clase hija exactamente igual que a la madre
- Las constraints **se heredan y se acumulan**: el sesgo a los bordes de `A` y `B` sigue estando, porque la de la clase base no desaparece

```
class add_transaction extends command_transaction;
   `uvm_object_utils(add_transaction)

   constraint add_only {op == add_op;}

   function new(string name = "");
      super.new(name);
   endfunction
endclass : add_transaction
```

Transactions

Paso 3 · dos testers se vuelven uno

```
task run_phase(uvm_phase phase);
   command_transaction command;

   phase.raise_objection(this);

   // ALL transactions come out of the factory, the directed ones too.
   // A new() here would compile and work just the same, but it would skip
   // add_test's set_type_override: the factory never hears about what you
   // build by hand. See the slide "the new() that eats the override".
   command = command_transaction::type_id::create("command");
   command.op = rst_op;
   command_port.put(command);

   repeat (1000) begin : random_loop
      command = command_transaction::type_id::create("command");
      // randomize() returns 0 if the constraints have no solution, and aborts
      // nothing. A bare assert() lets an unrandomized transaction through in
      // silence: the TB carries on, sending garbage.
      if (!command.randomize()) `uvm_fatal("TESTER", "randomize() failed")
      command_port.put(command);
   end : random_loop

   // Directed: the multiplier overflow case, which 1000 random operations
   // might never touch. It comes out of the factory like the others, but
   // since it is not randomized, add_transaction's constraint does not run:
   // the factory picks the TYPE, constraints only act inside randomize().
   command = command_transaction::type_id::create("command");
   command.op = mul_op;
   command.A = 8'hFF;
   command.B = 8'hFF;
   command_port.put(command);

   #500;
```

```
    phase.drop_objection(this);
  endtask : run_phase
```

- El testbench del env tenía `base_tester` y `add_tester`: dos clases para dos tipos de estímulo
- Ahora hay **una sola**. El `repeat` crea una transaction y le dice `randomize()`; qué sale de ahí lo decide el **dato**, no el tester
- Ésa es la línea que borra una clase: la decisión se mudó de la estructura al tipo de la transaction

Transactions

Paso 6 · el scoreboard compara objetos, no números

- Los dos lados de la comparación son ahora `result_transaction`: el que llega del `result_monitor` y el que arma el predictor
- El `write()` toma el resultado `t`, busca el comando que le corresponde en la cola del `command_monitor`, y le pide a `predict_result()` la predicción
- La comparación queda en una línea — `predicted.compare(t)` — y el mensaje de error se arma solo con los `convert2string()` de las tres transactions

```
function void write(result_transaction t);
   string data_str;
   command_transaction cmd;
   result_transaction predicted;

   do
   if (!cmd_f.try_get(cmd))
      `uvm_fatal("SELF CHECKER", "Missing command in self checker")
   while ((cmd.op == no_op) || (cmd.op == rst_op));

   predicted = predict_result(cmd);

   data_str = {
      cmd.convert2string(),
      " ==>  Actual ",
      t.convert2string(),
      "/Predicted ",
      predicted.convert2string()
   };
```

```
    if (!predicted.compare(t)) `uvm_error("SELF CHECKER", {"FAIL: ", data_str})
    else `uvm_info("SELF CHECKER", {"PASS: ", data_str}, UVM_HIGH)

  endfunction : write
```

Transactions

Paso 7 · el override, ahora sobre el dato

```
class add_test extends random_test;
   `uvm_component_utils(add_test);

   function void build_phase(uvm_phase phase);
      command_transaction::type_id::set_type_override(add_transaction::get_type());
      super.build_phase(phase);
   endfunction : build_phase

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass
```

- El test entero son **dos líneas**: el override y el `super.build_phase()`
- Y el override ya no cambia un **component** —el tester— sino un **dato** —la transaction—. La estructura del testbench dejó de tener opinión sobre el test
- `super.build_phase(phase)` va **después** del override, y ese orden no es negociable: el env se construye ahí adentro

Transactions

El new() que se come el override

```
command = new("command");                                    // NO
command.op = rst_op;

command = command_transaction::type_id::create("command");  // SI
command.op = rst_op;
```

- `set_type_override()` le habla **a la factory**, y la factory sólo se entera de lo que pasa por `type_id::create()`
- Un `new()` construye el tipo que dice esa línea y nada más: el override **no lo alcanza**. Compila, corre, y no falla — ese objeto simplemente se quedó afuera
- Es la peor clase de bug: no rompe, **miente**. Con un test y una transaction no se nota nunca; el día que el override sí importaba, ya es tarde
- La regla, corta: **todo `uvm_object` que viaje por el testbench sale de `type_id::create()`** — las transactions del tester, y también las que arman los monitores y el `predicted` del scoreboard
- Los *ports* y los *exports* son la excepción: no están en la factory y se instancian con `new()`, como vimos en un productor, muchos oyentes. La *TLM FIFO* sí está registrada, pero el curso la crea con `new()` igual porque el `size` va por constructor y `create()` no lo puede pasar

Transactions

Y `randomize()` nunca va solo

```
assert (command.randomize());                                  // NO
if (!command.randomize()) `uvm_fatal("TESTER", "randomize() fallo")  // SI
```

- `randomize()` devuelve **0** cuando las constraints no tienen solución. No aborta, no imprime nada por su cuenta: devuelve 0 y sigue
- Un `assert()` pelado reporta por afuera de UVM: no entra en el *Report Summary* —el mismo que aprendimos a leer en reporting— y en muchos simuladores la ejecución continúa con la transaction **sin randomizar**
- Y si el proyecto compila con las aserciones apagadas, hay simuladores que ni evalúan la expresión: `randomize()` directamente **no se llama**
- Con `` `uvm_fatal `` el error trae componente, tiempo y archivo, y frena la corrida ahí mismo

Transactions

Resumen de la unidad

- Vimos cómo utilizar *transactions* para mover datos por el TB
- Esto nos permite simplificar los componentes del TB y hasta remover una clase de nuestro TB inicial (add_tester)
- La unidad que sigue agrupa las clases que hablan con una misma interface en un solo bloque reutilizable: el `uvm_agent`

Constrained random

La otra mitad de la pinza

- La cobertura funcional del testbench convencional te dice **qué falta**. No te dice cómo llegar: eso es el estímulo
- Escribir un test dirigido por cada bin no escala — son 76 en el VTALU, y un chip de verdad tiene miles
- La receta de la industria es al revés: **random para el grueso, dirigido para los agujeros**. Y el random tiene que ser *legal*, o el DUT se queja de cosas que nunca van a pasar en silicio
- *Constrained random* es escribir las reglas del estímulo **una vez, en la transaction**, y dejar que el solver arme los casos

Constrained random

`rand` y `randomize()`: lo que ya viste

```
class command_transaction extends uvm_sequence_item;
   rand byte unsigned A;      // rand  -> entra en el sorteo
   rand byte unsigned B;
   rand operation_t   op;     // un enum se randomiza sobre sus valores

   constraint data { ... }    // las reglas, en un bloque aparte
endclass
```

- `randomize()` viene de SystemVerilog, no de UVM: **toda** clase lo tiene
- Elige valores para los campos `rand` que **cumplan todas las constraints activas a la vez**
- Un `constraint` **no es código secuencial**: no se ejecuta de arriba abajo, no tiene orden. Es un conjunto de relaciones que el solver satisface junto
- Por eso dos constraints que se contradicen no dan error de compilación: dan un `randomize()` que devuelve 0

Constrained random

Las otras tres palabras: `randc` y los dos ganchos

```
class command_transaction extends uvm_sequence_item;
   rand  byte unsigned A;       // puede repetir: 3 3 6 1 3 4 6 2 ...
   randc bit [2:0]     indice;  // recorre los 8 y recién ahí repite

   function void pre_randomize();   // corre ANTES del sorteo
      if (!primera) A = A_anterior; // …con los valores todavía viejos
   endfunction

   function void post_randomize();  // corre DESPUÉS, con los valores nuevos
      checksum = A ^ B;             // lo que no se sortea, se calcula acá
   endfunction
endclass
```

- `randc` es **cíclico**: agota todos los valores antes de repetir uno. Medido en Verilator 5.052: `4 2 1 7 5 6 3 0` y recién entonces vuelve a empezar
- Sirve para recorrer un espacio chico —un opcode, un índice de registro— sin esperar a que el azar lo cubra. Sólo en tipos integrales **chicos**: el LRM garantiza 8 bits y los simuladores suelen cortar en 16
- `pre_randomize()` / `post_randomize()` son los dos ganchos que el `randomize()` llama solo. El campo **derivado** —un checksum, una paridad— se calcula en el segundo, no se sortea
- Y una advertencia: lo que se calcule en `post_randomize()` **no** participa del solver. Si tiene que cumplir una constraint, es un campo `rand`

Constrained random

`dist :=` no es lo mismo que `:/`

```
A dist {8'h00 := 1, [8'h01 : 8'hFE] := 1, 8'hFF := 1};   // el peso va a CADA valor
A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};   // el peso se REPARTE
```

- `dist` pone **pesos**: no cambia qué valores son legales, cambia con qué frecuencia salen
- Con `:=` el peso se aplica **a cada valor del rango**: el rango del medio pesa 254 y los bordes 1 cada uno. Sale `A=00` **1 vez cada 256**
- Con `:/` el peso es **del rango entero**: 1 – 2 – 1 da 25 % en `00`, 50 % en el medio, 25 % en `FF`. Que es exactamente el sesgo que el `get_data()` del testbench convencional hacía a mano
- Las dos líneas compilan, las dos corren, y **una de las dos no llena los bins de borde nunca**

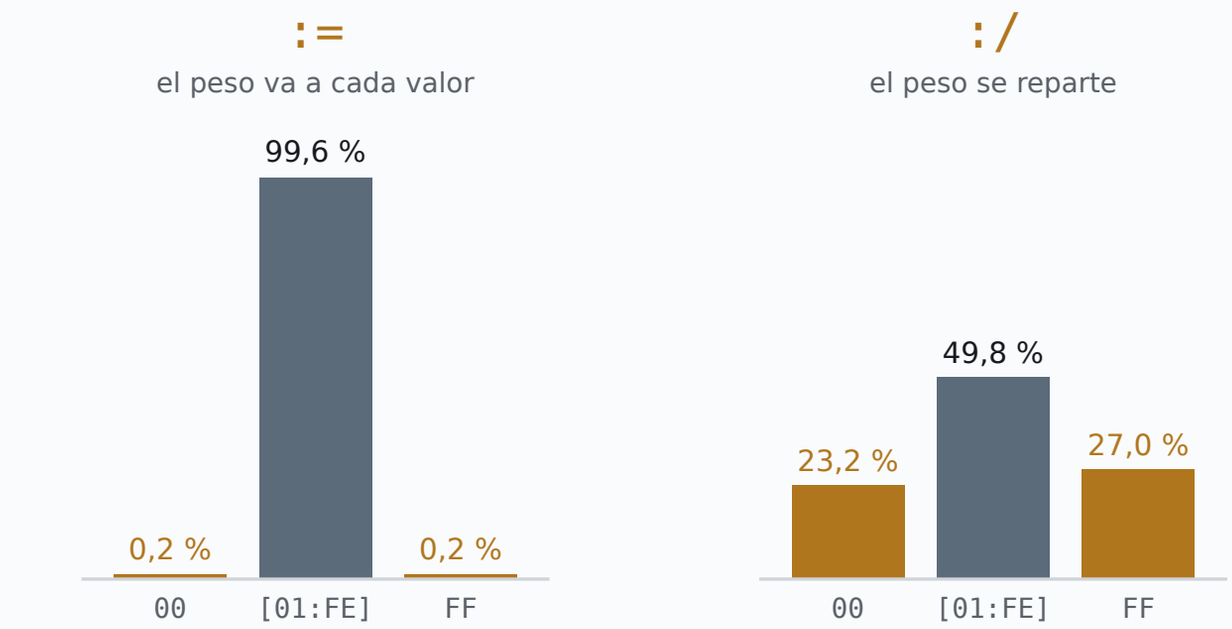
Constrained random

No lo supongas: medilo

```
// The weight goes to EVERY value of the range: the middle takes 254 of 256.
class por_valor;
   rand byte unsigned A;
   constraint data {A dist {8'h00 := 1, [8'h01 : 8'hFE] := 1, 8'hFF := 1};}
endclass

// The weight is SPLIT inside the range: 1/4 at 00, 1/2 in the middle, 1/4 at FF.
// It is the same bias get_data() of the Functional coverage section did by hand.
class por_rango;
   rand byte unsigned A;
   constraint data {A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};}
endclass
```

```
$ cd code/u6/transactions/constraints && bash run.sh
400 randomizations of each version
  :=   A=00   0.2%   A=FF   0.2%   the weight goes to each value
  :/   A=00  23.2%   A=FF  27.0%   the weight gets split
```



400 randomizaciones de cada versión, medidas. Las dos compilan, las dos corren, ninguna avisa. Los bins de borde son los mismos, y con `:=` tardan 64 veces más en llenarse.

Constrained random

inside y randomize() with {}

```
class comando;
   rand byte unsigned A;
   rand byte unsigned B;
   rand operation_t   op;

   constraint data {
      A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};
      B dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};
   }

   // inside is a set of legal values. Without it the random also asks for
   // no_op and rst_op, which compute nothing.
   constraint utiles {op inside {add_op, and_op, xor_op, mul_op};}
endclass
```

```
   // The portable workaround meanwhile: turn off the split, which for a
   // directed case makes no sense anyway.
   c.data.constraint_mode(0);
   if (!c.randomize() with {
         op == mul_op;
         A  == 8'hFF;
         B  == 8'hFF;
       })
      $fatal(1, "randomize() with failed even with the constraint turned off");
   $display("with with: 1 try,      A=%2h %s B=%2h", c.A, c.op.name(), c.B);
```

- `inside {a, b, c}` es el conjunto de valores legales: sin él, el random también pide `no_op` y `rst_op`, que no calculan nada. Acá además deja afuera a `sub_op`, para que el sorteo quede uniforme entre cuatro
- `randomize() with { ... }` agrega constraints **sólo para esa llamada**: el caso dirigido se pide en el punto de uso, sin tocar la clase ni escribir un tester nuevo. Es la herramienta del *coverage closure*

Constrained random

`soft`: el default que un `with {}` puede pisar

```
class command_transaction extends uvm_sequence_item;
   rand byte unsigned A;
   constraint por_defecto { soft A == 8'h00; }   // "cero, salvo que me pidan otra cosa"
endclass

t.randomize();                          // A = 00
t.randomize() with { A inside {[1:10]}; };   // A entre 1 y 10: el soft se cae solo
```

- Una constraint `soft` es un **deseo**, no una regla: si choca con otra, el solver la **descarta** en vez de devolver 0
- Es la forma moderna de escribir un valor por defecto en la transaction sin obligar a nadie a llamar `constraint_mode(0)` para pisarlo
- Gana la constraint **dura**, y entre dos `soft` que chocan gana la declarada **después** — el orden sí importa acá, y sólo acá
- Medido en Verilator 5.052: con el `with {}` sale entre 1 y 10; sin él, `00`

Constrained random

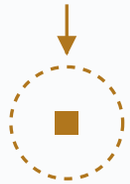
El orden de resolución sesga sin avisar

```
rand bit           es_reset;
rand byte unsigned A;
constraint c {es_reset -> A == 8'h00;}   // "si pido reset, los operandos en cero"
```

- Parece inofensiva. **No lo es**: el solver elige entre las *soluciones*, no entre los valores de cada campo
- Con `es_reset = 1` hay **una**; con `es_reset = 0`, **256**. Pediste reset la mitad de las veces y lo vas a ver **1 de cada 257**
- El bin *"cualquier operación después de un reset"* del plan no se llena, y el reporte no te dice por qué
- La respuesta del lenguaje es `solve es_reset before A`: elegí el campo de control primero

Una celda, una solución. El sorteo es uniforme sobre **todas**:

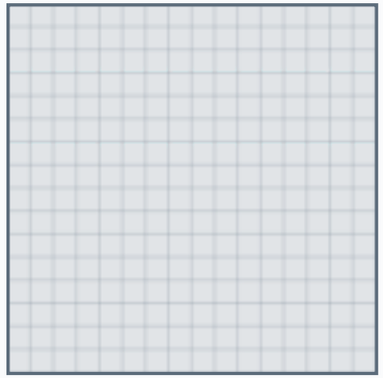
`es_reset = 1`



1 solución

`A == 8'h00`

`es_reset = 0`



256 soluciones

A queda libre

El solver elige entre las **257 soluciones**, no entre los dos valores de `es_reset`.
Pediste reset la mitad de las veces y lo vas a ver 1 de cada 257. Medido: 0,3 %.

Constrained random

Medido, con el rodeo que Verilator sí respeta

```
class sesgada;
   rand bit           es_reset;
   rand byte unsigned A;
   constraint c {es_reset -> A == 8'h00;}
endclass

// The language's answer is "solve es_reset before A": pick the control field
// FIRST and then solve the rest. Verilator 5.052 accepts the syntax and does
// NOT honour it -- it leaves es_reset stuck at 0, which is worse than
// ignoring it. Repro in code/verilator/repro-solve-before.sv.
//
// The workaround that IS portable: ask explicitly for the split of the
// control field. It says the same and does not rely on the solver ordering well.
class pareja extends sesgada;
   constraint reparto {es_reset dist {1'b0 :/ 1, 1'b1 :/ 1};}
endclass
```

```
2000 randomizations of each version
  as written             es_reset=1 in   0.3%   (1 in 257)
  with dist on es_reset  es_reset=1 in  51.7%
```

- Verilator 5.052 **acepta** `solve ... before` **y no lo respeta**: deja el campo clavado en 0, que es peor que ignorarlo. Repro en `code/verilator/repro-solve-before.sv`
- El rodeo portable: pedir el reparto del campo de control con un `dist`. Dice lo mismo y no depende de que el solver ordene bien

Constrained random

Cuando no hay solución

```
class comando;
   rand byte unsigned A;
   constraint chico {A < 8'h10;}
   constraint grande {A > 8'hF0;}  // contradicts the previous one: no solution
endclass

initial begin
   comando c;
   c = new();

   // 1. Both constraints active: there is no A that satisfies the two.
   if (!c.randomize()) $display("1. randomize() returned 0: the constraints do not close");
   else $display("1. randomize() gave %2h  (should not get here)", c.A);

   // 2. constraint_mode(0) turns a constraint off at run time.
   c.grande.constraint_mode(0);
   if (!c.randomize()) $display("2. randomize() returned 0  (should not get here)");
   else $display("2. with 'grande' turned off: A=%2h, and it honours 'chico'", c.A);

   // 3. rand_mode(0) takes the field out of the draw: it keeps its current value.
   c.A.rand_mode(0);
   if (!c.randomize()) $display("3. randomize() returned 0  (should not get here)");
   else $display("3. with A out of the draw: A=%2h, the same as before", c.A);

   $finish;
```

```
1. randomize() returned 0: the constraints do not close
2. with 'grande' turned off: A=06, and it honours 'chico'
3. with A out of the draw: A=06, the same as before
```

- `constraint_mode(0)` apaga **una constraint** en tiempo de ejecución; `rand_mode(0)` saca **un campo** del sorteo y le deja el valor que tenía — y si ese valor no cumple las constraints que lo nombran, `randomize()` devuelve 0
- Sirven para el test que necesita romper una regla a propósito — inyectar un opcode ilegal, por ejemplo — sin tocar la clase que todos los demás usan

Constrained random

Cómo se debuggea una constraint que no cierra

```
// 1. Que el testbench lo diga. Sin esto no hay bug: hay silencio
if (!t.randomize()) `uvm_fatal("RAND", "la constraint no cerró")

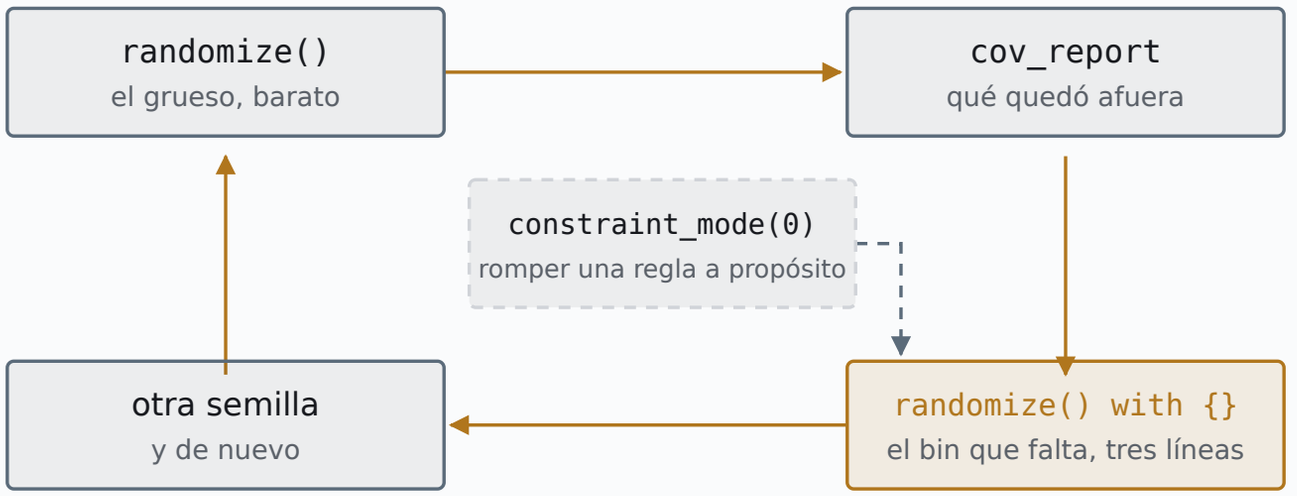
// 2. Bisección: apagar de a una hasta que cierre. La última que apagaste es
foreach (nombres[i]) begin
   t.chico.constraint_mode(0);
   if (t.randomize()) `uvm_info("RAND", "cierra sin 'chico'", UVM_LOW)
   t.chico.constraint_mode(1);
end

// 3. Sacar el campo del sorteo y ver qué valor lo estaba trabando
t.A.rand_mode(0);
```

- El síntoma nunca es *"no cierra"*: es un `randomize()` que devuelve 0 y una transaction con **los valores de la anterior**. Primero hay que hacerlo ruidoso
- El método es **bisección**, igual que con `git bisect`: `constraint_mode(0)` de a una hasta que cierre. La que faltaba apagar es la culpable
- El otro corte es por campo: `rand_mode(0)` deja el valor que tenía y muestra si el conflicto era ése
- La causa es casi siempre la misma: una constraint de la clase **base** que el `with {}` del test contradice. Ahí la respuesta de diseño es `soft`

Constrained random

Así se cierra la cobertura



Cerrar cobertura es este ciclo, y es a lo que un verificador le dedica el día.
Lo que no es: escribir un test por bin, ni mirar el porcentaje total.

- Ningún paso pide una clase nueva: `dist` e `inside` en la transaction, tres líneas de `with {}`, y `rand_mode(0)` o `constraint_mode(0)` para romper una regla
- Eso es *coverage closure*, y es a lo que un verificador le dedica el día
- Lo que **no** es: escribir un test por bin. Ni mirar el porcentaje total
- El estímulo todavía sale de un `tester` que escribiste vos. En UVM moderno eso es una `uvm_sequence` — y es el día 6

Constrained random

«Otra semilla, y de nuevo»: qué mueve y qué no

```
$ SEED=7 bash run.sh    # u6/transactions/constraints, 400 randomizaciones
    seed: 7
  :/   A=00  25.0%   A=FF  22.0%   # con SEED=8: 20.8% y 27.0%
$ SEED=7 bash run.sh    # u2/convencional, y de nuevo con SEED=8
  covergroup : 86.8% (66/76)       # las dos veces. Y el merge, igual
```

- `SEED=N` pasa `+verilator+seed+N`, y `run_sim` **imprime la que usó**: un fallo que sale una vez cada diez corridas no se debuggea, se repite
- **Sí** mueve los porcentajes de un `dist` medido con pocas muestras: 400 randomizaciones dan 25 % o 21 % según el sorteo. Es una muestra, no la distribución
- **No** mueve los 10 bins que faltan. Con 1000 operaciones el random ya llegó hasta donde puede, y otra semilla es tirar la moneda esperando otro resultado
- Por eso *"otra semilla"* es el **último** paso del ciclo anterior y no el primero: primero el `with {}`

Constrained random

Resumen de la unidad

- La cobertura te dice **qué falta**; constrained random es **cómo llegar**. Son las dos mitades de la misma pinza
- Escribir un test dirigido por bin no escala: son **76 bins** en el VTALU. La receta es al revés — **random para el grueso, dirigido para lo que queda**
- Un `constraint` **no es código secuencial**: no se ejecuta de arriba abajo. Es un sistema de ecuaciones que el solver resuelve todo junto
- Por eso dos constraints que se contradicen **no dan error de compilación**: dan un `randomize()` que devuelve 0
- `dist` pone **pesos**, no legalidad. Y `:=` no es `:/`: el primero pesa **cada valor** del rango, el segundo **el rango entero**
- `randomize()` devuelve 0 y sigue. Por eso va **siempre** adentro de un `if` con `` `uvm_fatal `` — es la trampa muda más cara del curso

Repaso · Día 5

1 de 8 · Copia profunda

`obj1_h = obj2_h`. ¿Qué copié?

- **a** Nada: los dos handles apuntan al mismo objeto ✓
- **b** Todos los campos de `obj2_h` a `obj1_h`
- **c** Sólo los campos `rand`
- **d** Una copia superficial: el primer nivel se copia y los handles de adentro se comparten

Nada — no hay copia, hay un segundo nombre para el mismo objeto. Si lo modificás por un handle, el otro lo ve. De ahí sale la regla del *MOOCOW*: si vas a modificar, copiá primero.

Repaso · Día 5

2 de 8 · super.do_copy()

¿Por qué cada `do_copy()` de la jerarquía tiene que llamar a `super.do_copy()`?

a. Porque UVM lo exige para registrar la clase

b. Para que el objeto quede registrado en la factory

c. Porque si no, los campos de las clases de arriba no se copian ✓

d. Porque `do_copy()` es `pure virtual` y la clase base no tiene implementación

Si no, se pierden los campos de arriba — es el mismo problema que `convert2string()` : el día que alguien mete un nivel nuevo en el medio, el método deja de ver la mitad de los datos y no te avisa nadie.

Repaso · Día 5

3 de 8 · clone()

`clone()` devuelve un `uvm_object`. ¿Por qué se recomienda escribir además un `clone_me()`?

- **a** Para encapsular el `$cast` en un solo lugar ✓
- **b** Porque `clone()` no copia los datos
- **c** Porque `clone()` está deprecado en IEEE 1800.2 y lo reemplaza `copy()`
- **d** Para poder clonar componentes además de transacciones

Para no repetir el `$cast` — sin él, cada lugar que clona termina escribiendo su propio casteo. Es la misma idea de siempre: si lo vas a repetir, encapsulalo.

Repaso · Día 5

4 de 8 · dist: := contra :/

`A dist {8'h00 := 1, [8'h01:8'hFE] := 1, 8'hFF := 1};` — **¿cada cuánto sale `A = 8'h00`?**

a. Un tercio de las veces: son tres entradas con el mismo peso

b. La mitad: los bordes se reparten entre los dos

c. Nunca: `:=` sólo acepta valores sueltos, no rangos, y el rango se descarta

d. Una vez cada 256: con `:=` el peso va a **cada valor** ✓

1 de cada 256 — `:=` le da peso 1 a *cada uno* de los 254 valores del medio, así que el rango pesa 254 contra 1 y 1 de los bordes. Para repartir el peso *dentro* del rango va `:/`. Las dos formas compilan y corren: la diferencia sólo aparece en la cobertura que no sube.

Repaso · Día 5

5 de 8 · randomize() with {}

`cmd.randomize() with { A == 8'hFF; }` — ¿qué pasa con las constraints que la clase ya tenía?

a. Se reemplazan: para esa llamada vale sólo lo que está entre las llaves

b. Se suman: hay que satisfacer las dos ✓

c. Quedan apagadas hasta el próximo `randomize()` sin `with`

d. Depende del orden de declaración: gana la que está más abajo en el archivo

Se suman — el `with {}` agrega constraints **sólo para esa llamada** y no borra nada. Por eso es la herramienta para cerrar un bin sin escribir una clase nueva: tres líneas en el punto de uso, y el resto del estímulo sigue siendo el de siempre. Y por eso también choca con un `dist` que ya sesgue el mismo campo: las dos tienen que dar a la vez.

Repaso · Día 5

6 de 8 · constraint_mode()

`randomize() with { A == 8'hFF; }` sobre un campo que la clase reparte con `dist` devuelve 0 tres de cada cuatro veces. ¿Cuál es el rodeo?

a. Reintentar en un `do ... while` hasta que devuelva 1

b. `constraint_mode(0)` sobre la constraint del `dist`, para ese objeto ✓

c. Subir el peso del bin `8'hFF` en el `dist` de la clase

d. `rand_mode(0)` sobre el campo: saca el `dist` de la ecuación y deja mandar al `with`

Apagar la constraint que estorba — Verilator resuelve el `dist` **eligiendo un valor primero** y recién después chequea el resto, así que la probabilidad de éxito es la del bin: medido, `with {A == 8'hFF}` resuelve el 25 % de las veces. `constraint_mode(0)` la apaga sólo para ese objeto y no toca a nadie más. Es exactamente la línea que pide el ejercicio `d5c`.

Repaso · Día 5

7 de 8 · rand_mode()

¿Qué hace `cmd.A.rand_mode(0)`?

a Apaga todas las constraints que mencionan ese campo

b Lo randomiza una sola vez y después lo congela

c Hace que el solver lo resuelva último, después de todos los demás campos

d Saca el campo del sorteo y le deja el valor que tenía ✓

Deja de ser `rand` — son las dos perillas de tiempo de ejecución y se confunden seguido: `rand_mode(0)` saca **un campo** del sorteo, `constraint_mode(0)` apaga **una constraint**. Una elige *qué se sortea*, la otra *qué reglas valen*. Sirve para fijar un operando a mano y seguir randomizando el resto.

Repaso · Día 5

8 de 8 · El orden de resolución

`rand bit es_reset; rand byte unsigned A;` con `constraint c { es_reset -> A == 8'h00; }`. ¿Cada cuánto sale `es_reset = 1`?

a. La mitad de las veces: es un bit `rand` y nada lo prohíbe

b. Nunca: la implicación lo fuerza a 0

c. 1 de cada 257: el solver sortea entre las **soluciones** ✓

d. Depende de la semilla, y con suficientes corridas promedia la mitad

1 de cada 257 — el solver elige uniformemente entre las *soluciones*, no entre los valores de cada campo: `es_reset=1` deja una sola combinación (`A = 00`) y `es_reset=0` deja 256. El bin *"cualquier operación después de un reset"* del plan no se llena, y el reporte no dice por qué. La respuesta del lenguaje es `solve es_reset before A`; Verilator la acepta y **no la respeta**, así que el rodeo portable es pedir el reparto del campo de control con un `dist`.

Ejercicio · Día 5 · 1 de 3

El scoreboard grita y el DUT está sano

```
$ cd code/ejercicios/d5 && bash run.sh
```

- El TB de las transactions reporta `FAIL` en casi todas las operaciones
- El DUT es el mismo que viene pasando desde la spec. El bug es del TB
- Los mensajes que lo delatan son `UVM_HIGH`: subí la verbosidad
- Si te trabás, el apéndice **caja de debug** tiene la fila de este síntoma; el de **trampas mudas**, la causa

Ejercicio · Día 5 · 2 de 3

Medí tu `dist`

```
$ cd code/ejercicios/d5b && bash run.sh
```

- Una clase, un `dist` y un histograma. Sin UVM, sin DUT, sin testbench
- Los pesos **ya dicen 10, 80 y 10**. Corré primero, y mirá lo que sale
- Se pide que los tres casilleros den 10 / 80 / 10 con ±2 puntos de tolerancia
- La diferencia entre lo que hay y la solución es **un carácter**

Ejercicio · Día 5 · 3 de 3

Cerrar un bin

```
$ cd code/ejercicios/d5c && bash run.sh
```

- El test manda reset y **60 operaciones al azar**, y el reporte muestra el bin *"las dos patas en `FF`, multiplicando"* vacío
- Escribí el caso dirigido que lo cierre. Pero **pedilo con `randomize() with {}`**, no asignando los campos, que es lo que hace el `tester` de esta sección tres líneas más arriba
- Se corrige comparando la cobertura de antes y la de después: tiene que subir
- Es la **fila 3** del plan de verificación del día 1: la única de las doce cuya columna de estímulo dice *caso dirigido*

Ayer quedó...

Dónde dejamos el testbench

- El dato es un **objeto**: `command_transaction` con `do_copy`, `do_compare` y `convert2string`, y el scoreboard compara objetos, no campos
- El estímulo se pide en vez de escribirse: `randomize()`, `constraint`, `dist`, y el histograma que dice si la distribución hace lo que promete
- Y quedan **dos** cosas pegadas que no van juntas: el `tester` es estructura y estímulo a la vez, y el `env` nombra al monitor y al driver de a uno

Agenda

Día 6 · ≈ 4 h 45 · unidad 7 · el testbench reutilizable

- Agents
- Callbacks
- Sequences

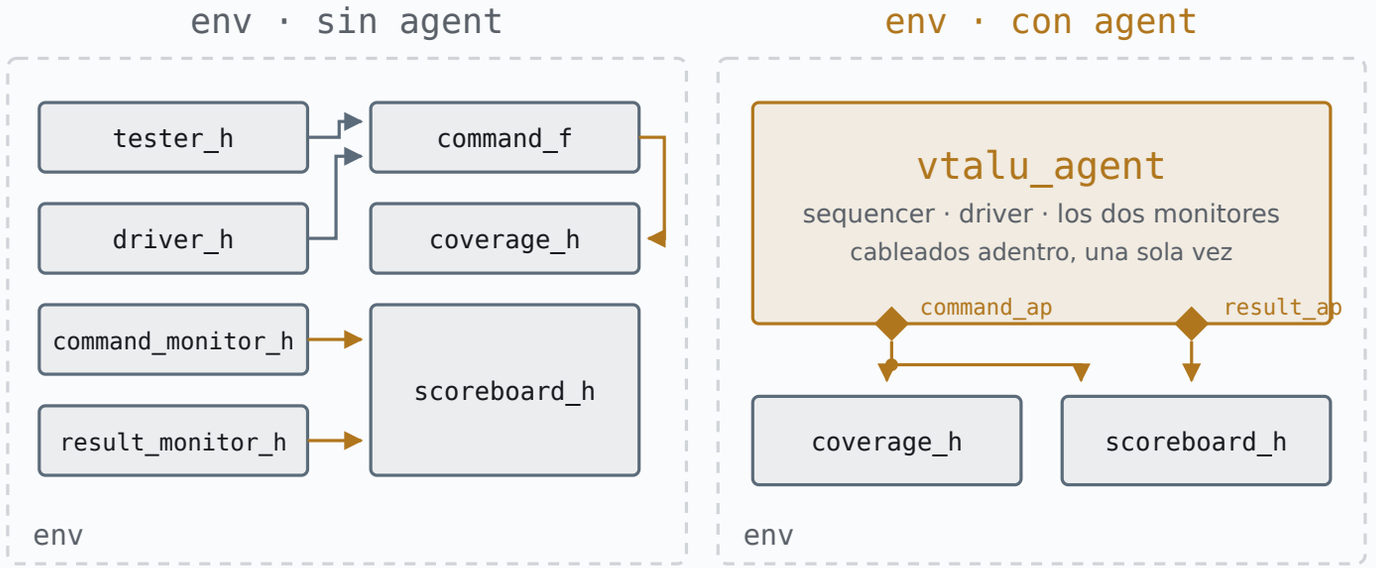
Al final del día podés:

- **Decir** qué construye y qué **no** un agent con `is_active = UVM_PASSIVE`
- **Explicar** por qué una `uvm_sequence` **no aparece** en `print_topology()`
- **Sacar** la última línea cableada de un test, con `default_sequence` por `uvm_config_db`

Agents

El problema: un testbench que no se puede copiar

- El `env` que dejamos en las transactions funciona, pero es un **bloque soldado**: siete componentes y cinco `connect()` escritos a mano
- Cada uno de esos componentes existe por una sola razón: **maneja o mira la misma interface**, la `vtalu_bfm`
- Preguntá qué pasa si el diseño trae *dos* VTALU: hay que copiar los siete objetos, las cinco conexiones, y renombrar todo
- Copiar y pegar estructura es exactamente lo que veníamos evitando desde el `env`
- UVM le pone nombre a la solución: **`uvm_agent`**



7 componentes y 5 conexiones, escritas a mano

Segunda VTALU: copiar todo el bloque

1 agent y 2 subscribers

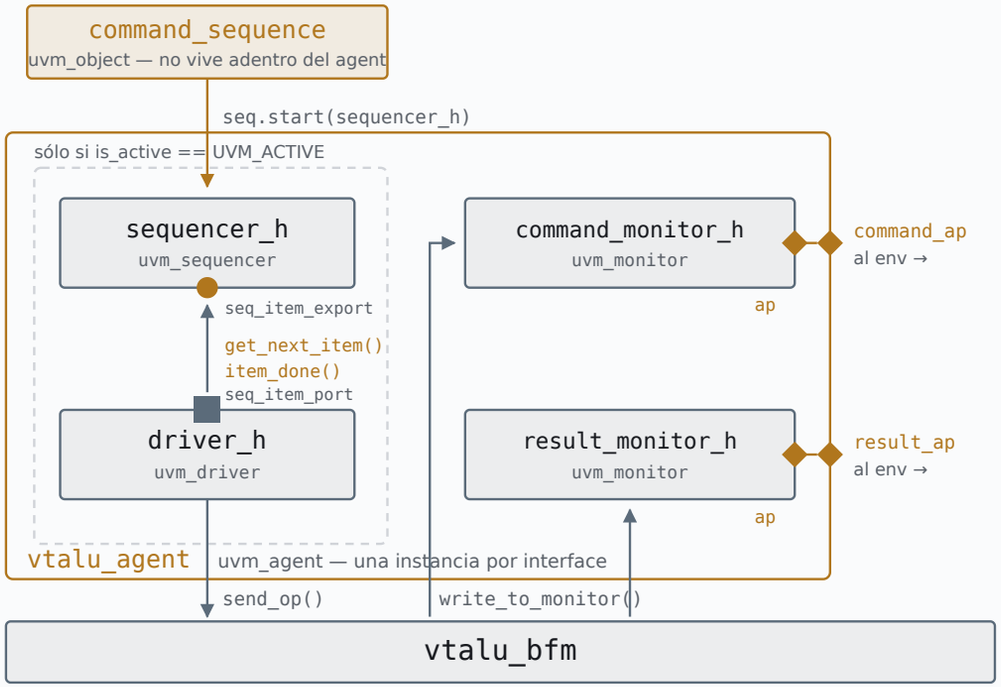
Segunda VTALU: una línea más, con otro config

El agent no es una clase más: es la unidad que se instancia una vez por interface.

Agents

Qué hay adentro de un agent

- Un `uvm_agent` es un `uvm_component` más: mismo árbol, mismas fases, misma factory
- Lo que cambia es **qué mete adentro**, siempre lo mismo:
  - un *sequencer*, que entrega estímulo
  - un *driver*, que lo convierte en señales de la BFM
  - los *monitores*, que miran el bus y publican transactions
- Y expone hacia afuera **dos** `uvm_analysis_port`: el que publica comandos y el que publica resultados
- El scoreboard y la cobertura **no van adentro**: son análisis, y viven en el `env` colgados de esos dos puertos
- Desde esta sección cada pieza usa **su** clase base: `uvm_driver #(T)`, `uvm_sequencer #(T)` y `uvm_monitor` en vez de `uvm_component` a secas



El scoreboard y el coverage viven en el env, colgados de esos dos rombos.

Agents

El salto: del tester a la sequence

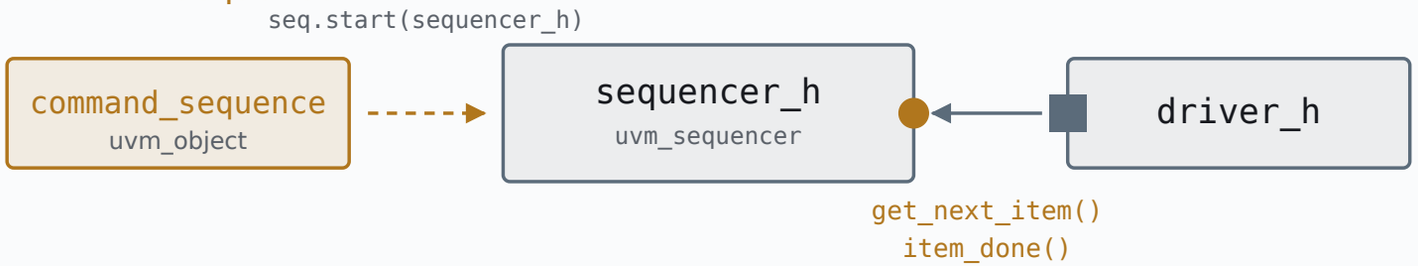
- En los put y get ports partimos el estímulo en dos: el `tester` elegía **qué** mandar, el `driver` sabía **cómo**
- Los uníamos con `uvm_put_port` / `uvm_get_port` y una `uvm_tlm_fifo` en el medio: tres componentes en el árbol y dos `connect()`
- UVM ya trae eso hecho, y con un nombre: el **`uvm_sequencer`**
- El `driver` deja de ser un `uvm_component` genérico y pasa a ser `uvm_driver #(T)`, que **ya viene con un `seq_item_port`**
- Y el `tester` desaparece: lo reemplaza una `uvm_sequence`, que **no es un componente** — es un objeto, no está en el árbol y no se conecta

día 4 · put / get



tres componentes en el árbol, dos connect(), y ningún aviso de vuelta

día 6 · sequencer



El sequencer reemplaza a la FIFO y agrega el `item_done()`: el driver avisa cuándo terminó.

La sequence no es un componente: no está en el árbol, no se conecta — se arranca.

Agents

El handshake: `get_next_item()` / `item_done()`

- El driver ya no hace `get()`: hace `get_next_item()`, que **bloquea** hasta que la sequence tenga algo
- Cuando terminó de manejar las señales, llama a `item_done()`
- Esa segunda llamada es lo que el par `put`/`get` **no tenía**: es el driver avisando *"ya está, mandame el próximo"*
- Sin `item_done()` la sequence se cuelga para siempre en su `finish_item()`, y la simulación se queda sin avanzar sin un solo mensaje de error
- El par siempre va completo, y en ese orden

```
task run_phase(uvm_phase phase);
   command_transaction command;
   forever begin : command_loop
      seq_item_port.get_next_item(command);
      bfm.send_op(command.A, command.B, command.op);
      seq_item_port.item_done();
   end : command_loop
endtask : run_phase
```

Agents

El driver, ahora `uvm_driver`

- `uvm_driver #(command_transaction)` trae el `seq_item_port` gratis: no se declara ni se instancia con `new()`
- El `build_phase` ya no busca la BFM directo en el `config_db`: se la pide al **config del agent**
- El `run_phase` es **tres líneas**: pedir el item, `bfm.send_op(...)`, avisar que terminó. El driver no toca un cable
- El protocolo sigue donde lo dejamos en interfaces y BFM: **adentro de la BFM**. Por eso el driver de un agent es tan corto, y por eso cambiar de protocolo no lo toca

```
class driver extends uvm_driver #(command_transaction);
   `uvm_component_utils(driver)

   virtual vtalu_bfm bfm;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      vtalu_agent_config cfg;
```

```
      // The BFM is not asked for on its own: ask for the agent config, and it comes from there.
      if (!uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg))
         `uvm_fatal("DRIVER", "Failed to get agent config")
      bfm = cfg.bfm;
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      command_transaction command;
      forever begin : command_loop
         seq_item_port.get_next_item(command);
         bfm.send_op(command.A, command.B, command.op);
         seq_item_port.item_done();
      end : command_loop
   endtask : run_phase

endclass : driver
```

Agents

El sequencer se declara con un `typedef`

- `uvm_sequencer` es una clase paramétrica: se parametriza con el tipo de item que va a entregar
- No hace falta extenderla para nada: se usa tal cual
- Un `typedef` en el package le pone nombre corto, y ese nombre es el que usan el agent y el `create()`

```
typedef uvm_sequencer #(command_transaction) sequencer;
```

- La `command_transaction` de las transactions ya extiende `uvm_sequence_item`, así que **no hay que tocarla**: por eso elegimos esa clase base y no `uvm_transaction`
- El `typedef` va **después** del `` `include `` de la transaction y **antes** del driver y del agent

Agents

La sequence mínima

- Una `uvm_sequence #(T)` tiene una task `body()`: eso es todo lo que hay que escribir
- Cada item va entre `start_item()` y `finish_item()`
- `start_item()` bloquea hasta que el sequencer nos da el turno; `finish_item()` bloquea hasta que el driver llamó a `item_done()`
- El `randomize()` va **entre los dos**: así las constraints se resuelven en el momento en que el item se va a entregar, no antes
- Ésta es la traducción directa del `tester` de las transactions, y nada más que eso — la sección de sequences es sobre lo que se puede hacer acá adentro

```
class command_sequence extends uvm_sequence #(command_transaction);
   `uvm_object_utils(command_sequence)

   function new(string name = "command_sequence");
      super.new(name);
   endfunction : new

   task body();
      command_transaction command;

      // All transactions come out of the factory, the directed ones too:
      // a new() would skip add_test's set_type_override.
```

```
    command = command_transaction::type_id::create("command");
    start_item(command);
    command.op = rst_op;
    finish_item(command);

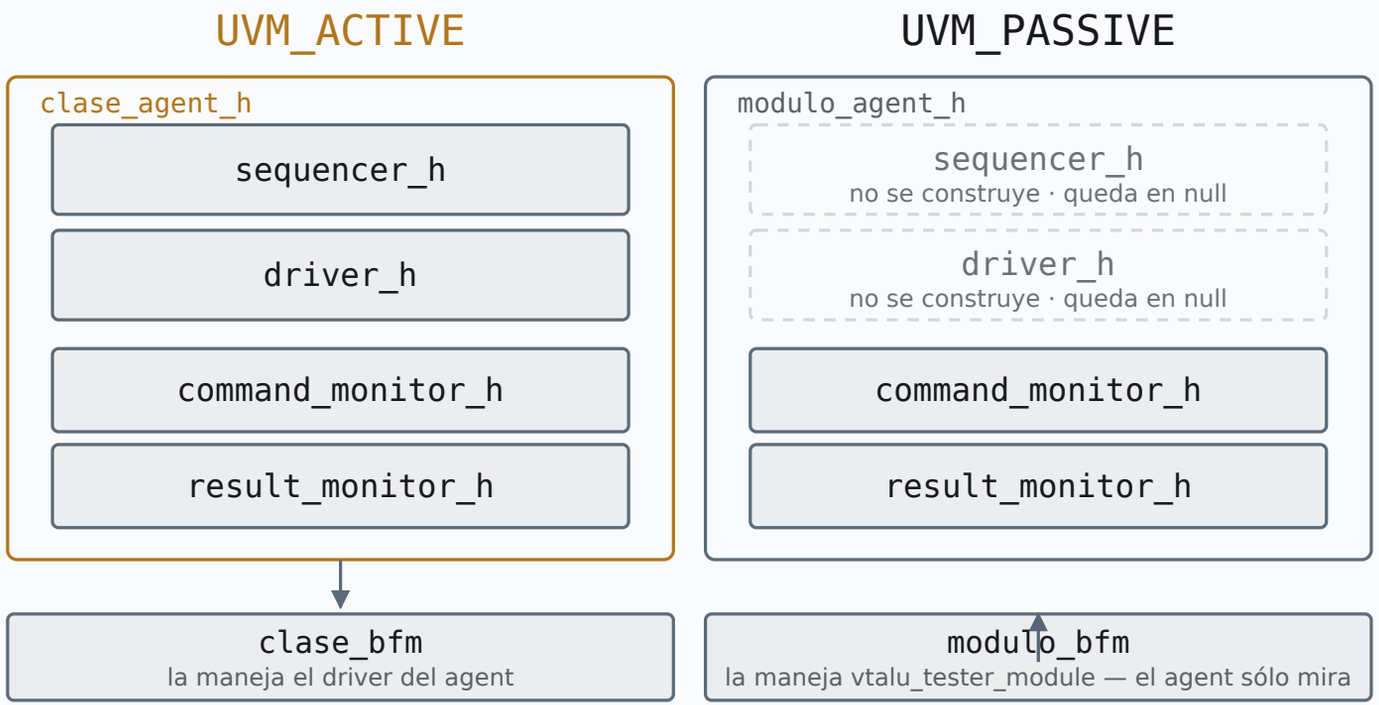
    repeat (1000) begin : random_loop
       command = command_transaction::type_id::create("command");
       start_item(command);
       // The randomize() goes BETWEEN start_item and finish_item: that way the
       // constraints are solved when the item is about to be handed over, and
       // not before. Today it makes no difference; the day a constraint
       // depends on the DUT state, it will.
       if (!command.randomize()) `uvm_fatal("COMMAND SEQ", "randomize() failed")
       finish_item(command);
    end : random_loop

    // Directed: the multiplier overflow, which 1000 random operations might
    // never touch. It comes out of the factory like the others, but since it
    // is not randomized, add_transaction's constraint does not run.
    command = command_transaction::type_id::create("command");
    start_item(command);
    command.op = mul_op;
    command.A  = 8'hFF;
    command.B  = 8'hFF;
    finish_item(command);
  endtask : body
```

Agents

`is_active`: activo y pasivo

- `uvm_agent` trae un campo `is_active` de tipo `uvm_active_passive_enum`, con dos valores: `UVM_ACTIVE` y `UVM_PASSIVE`
- **Activo**: se construyen el sequencer y el driver. El agent maneja la interface
- **Pasivo**: no se construyen. El agent **sólo mira** una interface que maneja otro
- Los monitores y los dos analysis ports se construyen **siempre**: un agent pasivo sigue alimentando scoreboard y cobertura
- Se lee con `get_is_active()` y se decide en el `build_phase`, con un `if`



Los monitores y los dos analysis ports se construyen SIEMPRE:
un agent pasivo sigue alimentando scoreboard y cobertura.

Agents

Y se puede ver

- `bash run.sh +TOPOLOGY` imprime el árbol de componentes que UVM armó

```
clase_agent_h          vtalu_agent
  command_ap           uvm_analysis_port
  command_monitor_h    command_monitor
  driver_h             driver
    seq_item_port      uvm_seq_item_pull_port
  result_ap            uvm_analysis_port
  result_monitor_h     result_monitor
  sequencer_h          uvm_sequencer
    seq_item_export    uvm_seq_item_pull_imp
modulo_agent_h         vtalu_agent
  command_ap           uvm_analysis_port
  command_monitor_h    command_monitor
  result_ap            uvm_analysis_port
  result_monitor_h     result_monitor
```

- Misma clase, dos instancias, y una tiene cuatro componentes menos
- Ahí están también el `seq_item_port` y el `seq_item_export` que el driver y el sequencer traen puestos, sin que nadie los declare — contra los put y get ports, donde el `uvm_put_port` había que declararlo y hacerle `new()`
- `print_topology()` es además la herramienta de debug cuando un `connect()` apunta a un componente que no existe

Agents

El config object del agent

- El agent necesita dos datos de afuera: **cuál BFM** y **si es activo**
- Los dos viajan juntos en un objeto de configuración, que es la interface pública del agent
- Es una **clase pelada**, no un `uvm_object`: por eso el constructor puede **exigir** los dos datos, y quien se olvide de uno no compila
- Un `uvm_object` se crea con `type_id::create()`, que sólo toma un `name`: la garantía del constructor se pierde
- El campo `is_active` va `protected` con un getter: nadie lo cambia después de que el agent se construyó

```
// The agent config is NOT a uvm_object: it is a plain class. That is why the
// constructor can DEMAND both values, and whoever forgets one does not compile.
// A uvm_object is built with type_id::create(), which only takes a name, and
// that guarantee is lost.
class vtalu_agent_config;

   virtual vtalu_bfm bfm;

   // protected + getter: nobody changes it after the agent was built.
   protected uvm_active_passive_enum is_active;

   function new(virtual vtalu_bfm bfm, uvm_active_passive_enum is_active);
```

```
      this.bfm = bfm;
      this.is_active = is_active;
   endfunction : new

   function uvm_active_passive_enum get_is_active();
      return is_active;
   endfunction : get_is_active

endclass : vtalu_agent_config
```

Agents

La clase agent: `build_phase`

- Pide su config al `uvm_config_db` y copia el `is_active` **a mano**: sin `super.build_phase()`, el de `uvm_agent` que lo leería no corre
- Construye sequencer y driver **sólo si es activo**; los monitores y los dos analysis ports, siempre
- Los analysis ports son ports: se instancian con `new()`, no con la factory

```
class vtalu_agent extends uvm_agent;
   `uvm_component_utils(vtalu_agent)

   vtalu_agent_config cfg;

   sequencer       sequencer_h;
   driver          driver_h;
   command_monitor command_monitor_h;
   result_monitor  result_monitor_h;

   uvm_analysis_port #(command_transaction) command_ap;
   uvm_analysis_port #(result_transaction)  result_ap;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      if (!uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg))
         `uvm_fatal("AGENT", "Failed to get agent config")
      is_active = cfg.get_is_active();
```

```
    if (get_is_active() == UVM_ACTIVE) begin : estimulo
       sequencer_h = sequencer::type_id::create("sequencer_h", this);
       driver_h    = driver::type_id::create("driver_h", this);
    end : estimulo

    command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
    result_monitor_h  = result_monitor::type_id::create("result_monitor_h", this);

    command_ap = new("command_ap", this);
    result_ap  = new("result_ap", this);
  endfunction : build_phase
```

Agents

La clase agent: `connect_phase`

- Adentro: el `seq_item_port` del driver contra el `seq_item_export` del sequencer — **una sola línea**, y sólo si es activo
- Hacia afuera: el `ap` de cada monitor se conecta al analysis port **del agent**
- Ese segundo `connect()` es el que hace que el agent sea una caja cerrada: el `env` nunca toca `command_monitor_h`, habla con `command_ap`
- Sigue valiendo el mantra *ports connect to exports*, con una vuelta más: un port también se conecta **a otro port del mismo tipo, hacia afuera**. Eso es `command_monitor_h.ap.connect(command_ap)`, y se llama *port forwarding*

Agents

Cómo queda el env

- Cuatro líneas de estructura en vez de siete objetos y cinco conexiones
- El `env` ya no sabe que existe un driver, ni un sequencer, ni una FIFO: sabe que hay un agent y que tiene dos analysis ports
- El scoreboard y la cobertura quedan igual que en los analysis ports: son subscribers, y no se enteran de nada

```
class env extends uvm_env;
   `uvm_component_utils(env)

   vtalu_agent        agent_h;
   vtalu_agent_config agent_cfg_h;

   scoreboard scoreboard_h;
   coverage   coverage_h;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      virtual vtalu_bfm bfm;

      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("ENV", "Failed to get BFM")

      agent_cfg_h = new(.bfm(bfm), .is_active(UVM_ACTIVE));
      uvm_config_db#(vtalu_agent_config)::set(this, "agent_h*", "config", agent_cfg_h);
```

```
   agent_h      = vtalu_agent::type_id::create("agent_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   coverage_h   = coverage::type_id::create("coverage_h", this);
endfunction : build_phase

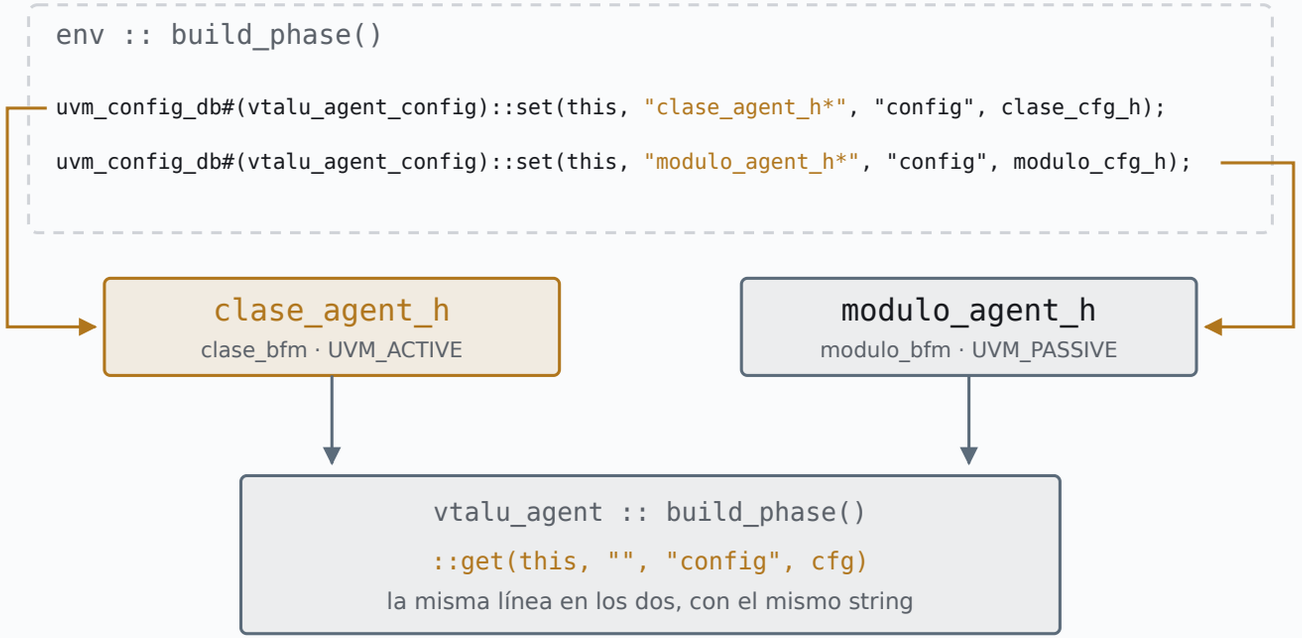
function void connect_phase(uvm_phase phase);
   agent_h.command_ap.connect(scoreboard_h.cmd_f.analysis_export);
   agent_h.command_ap.connect(coverage_h.analysis_export);
   agent_h.result_ap.connect(scoreboard_h.analysis_export);
endfunction : connect_phase
```

- Ese es el `env` de una sola VTALU. Con dos, cambia menos de lo que parece

Agents

Dos agents, un solo string

- Los dos agents ejecutan **la misma línea**: `uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg)`
- Y sin embargo tienen que recibir **objetos distintos**
- Lo resuelve el segundo argumento del `set()`: el **ámbito**, que es la ruta del componente que va a leer
- `"clase_agent_h*"` alcanza a ese agent y a todo lo que cuelga de él — por eso el driver y los monitores encuentran el mismo config sin que nadie se lo pase
- El `this` del `set()` es el punto de partida de esa ruta: el `env`



El ámbito del `set()` es la ruta del componente que va a leer, no un nombre libre.

Con "*" en las dos, el segundo pisa al primero y los dos agents arrancan pasivos.

Agents

Cuando el `config_db` no encuentra: `+UVM_CONFIG_DB_TRACE`

```
$ bash run.sh +UVM_CONFIG_DB_TRACE
UVM_INFO ... Configuration 'config' (type vtalu_agent_config) set by env_h
UVM_INFO ... Configuration 'config' read by
             accessor=uvm_test_top.env_h.clase_agent_h.driver_h
UVM_INFO ... Configuration 'config' read by
             accessor=uvm_test_top.env_h.modulo_agent_h.command_monitor_h
```

- El `config_db` es una base de datos por strings: cuando falla, **falla en silencio** o con un `uvm_fatal` que no dice por qué
- `+UVM_CONFIG_DB_TRACE` hace que UVM imprima **cada `set()` y cada `get()`**, con la ruta jerárquica completa del componente que lo pidió
- Sirve para las dos preguntas que uno se hace: *¿el `set()` llegó a este componente?* y *¿qué componentes existen de verdad?*
- Es un plusarg, no código: no hay que tocar ni recompilar nada

Agents

El ejemplo: dos VTALU y el módulo heredado

- El caso que justifica todo esto: hay que comparar **dos fuentes de estímulo** sobre el mismo diseño
- El `top` instancia dos VTALU y dos `vtalu_bfm`
- La primera la maneja nuestro agent, en `UVM_ACTIVE`
- La segunda la maneja `vtalu_tester_module`, un módulo de siempre sin una línea de UVM, y le colgamos un agent en `UVM_PASSIVE` para mirarlo
- Dos scoreboards y **dos coberturas**: eso es lo que responde cuál de los dos estímulos cubre más

```
module top;
    import uvm_pkg::*;
    import vtalu_pkg::*;
    `include "vtalu_macros.svh"
    `include "uvm_macros.svh"

    // The VTALU driven by the class testbench
    vtalu_bfm clase_bfm ();
    vtalu clase_dut (
        .A(clase_bfm.A), .B(clase_bfm.B), .op(clase_bfm.op_set),
        .clk(clase_bfm.clk), .reset_n(clase_bfm.reset_n), .start(clase_bfm.start),
        .done(clase_bfm.done),
        .ovf(clase_bfm.ovf), .result(clase_bfm.result)
    );
```

```
    // The VTALU driven by the legacy module
    vtalu_bfm modulo_bfm ();
    vtalu modulo_dut (
        .A(modulo_bfm.A), .B(modulo_bfm.B), .op(modulo_bfm.op_set),
        .clk(modulo_bfm.clk), .reset_n(modulo_bfm.reset_n), .start(modulo_bfm.start),
        .done(modulo_bfm.done),
        .ovf(modulo_bfm.ovf), .result(modulo_bfm.result)
    );

    vtalu_tester_module stim_module (modulo_bfm);

    initial begin
       // The top knows nothing about agents: it leaves the two interfaces with a
       // name and walks away. Handing them out is the test's job.
       uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "clase_bfm", clase_bfm);
       uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "modulo_bfm", modulo_bfm);
       run_test();
    end

endmodule : top
```

- El módulo llama **a la misma** `bfm.send_op()` que el driver: recibe la interface por su **puerto** en vez de por una virtual interface, pero el protocolo que ejecuta es exactamente el mismo código
- Eso es lo que hace justa la comparación: los dos estímulos hablan idéntico con el DUT. Lo único que se compara es **qué** manda cada uno, no cómo

Agents

El test, y lo que todavía falta

- El `test` saca las dos BFM del `config_db`, arma el `env_config` y lo baja
- Después arranca la sequence **a mano**, sobre el sequencer del agent activo:

```
seq = command_sequence::type_id::create("seq");
seq.start(env_h.clase_agent_h.sequencer_h);
```

- Funciona, pero mirá lo que hace esa línea: el test **atraviesa la jerarquía** para llegar al sequencer, y con eso vuelve a conocer el interior del env
- Todo lo que acabamos de encapsular se filtra en una línea
- La sección de sequences lo arregla: la sequence se le pasa al sequencer **por `config_db`**, y el test deja de saber dónde está

```
function void build_phase(uvm_phase phase);
   virtual vtalu_bfm clase_bfm, modulo_bfm;
   env_config env_config_h;

   // The top left the two interfaces loose; the test packs them up.
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "clase_bfm", clase_bfm))
      `uvm_fatal("DUAL TEST", "Failed to get clase_bfm")
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "modulo_bfm", modulo_bfm))
      `uvm_fatal("DUAL TEST", "Failed to get modulo_bfm")

   env_config_h = new(.clase_bfm(clase_bfm), .modulo_bfm(modulo_bfm));
```

```
    uvm_config_db#(env_config)::set(this, "env_h*", "config", env_config_h);

    env_h = env::type_id::create("env_h", this);
endfunction : build_phase

// +TOPOLOGY prints the component tree. It is how to SEE that the passive
// agent built neither a sequencer nor a driver.
function void end_of_elaboration_phase(uvm_phase phase);
    if ($test$plusargs("TOPOLOGY")) uvm_root::get().print_topology();
endfunction : end_of_elaboration_phase

task run_phase(uvm_phase phase);
    command_sequence seq;

    phase.raise_objection(this);

    // The only thing left wired by hand: the test reaches across the
    // hierarchy to get to the sequencer. The sequences section takes it out of here.
    seq = command_sequence::type_id::create("seq");
    seq.start(env_h.clase_agent_h.sequencer_h);

    #500;
    phase.drop_objection(this);
endtask : run_phase
```

Agents

Lo que esta sección hace distinto

- **El libro no usa sequencer acá.** En *The UVM Primer* el agent de esta sección todavía tiene el `tester` y la `uvm_tlm_fifo` adentro, y el sequencer recién aparece con las sequences — donde además el `tester` desaparece
- Nosotros armamos el agent **completo de una**: es la forma en que se escribe hoy, y hace que la sección de sequences sea sólo sobre sequences
- **El scoreboard y la cobertura quedan afuera del agent**, contra lo que hace el *UVM Primer*. Un agent pasivo no debería arrastrar un scoreboard
- **El config del agent no es un `uvm_object`**: es una clase pelada, para que el constructor obligue. La industria suele hacerlo `uvm_object`
- El protocolo vive en la BFM, igual que en el *UVM Primer*. Hasta Verilator 5.051 esto no se podía: una task de interface llamada por una virtual interface no propagaba. Se arregló en 5.052 — `docs/verilator.md`

Agents

Resumen de la unidad

- Un **agent** empaqueta todo lo que sabe hablar una interface: sequencer, driver y monitores, cableados adentro una sola vez
- Se instancia **una vez por interface**, y se configura con un **objeto de configuración**, no con handles sueltos
- `is_active` decide si además de mirar, maneja
- El `env` pasó de siete componentes cableados a mano a **un agent y dos subscribers**
- El `uvm_config_db` deja de ser un buzón global: el **ámbito** es una ruta en el árbol, y con más de un agent es lo que hace que cada uno reciba lo suyo
- Quedó una sola cosa cableada: el test todavía busca el sequencer a mano para arrancar la sequence. **Eso es la sección de sequences**

Callbacks

El tercer gancho

- El día 1 prometimos **tres** formas de escribir un test nuevo sin tocar el env: **constraints**, **factory override** y **callbacks**. Van dos
- Las tres son la misma idea a distinta altura: la constraint cambia **valores**, el override cambia **una clase entera**, el callback cambia **qué hace una clase en un punto**
- El caso que las separa: te dan un agent de VIP —de otro equipo, o comprado— y necesitás que el driver mande un dato roto **una vez cada tanto**
- Con override tenés que reescribir el driver entero para cambiar una línea. Con un callback enganchás esa línea y nada más
- Y hay una diferencia que no es de estilo: **los callbacks se apilan**. El override elige *una* clase; los callbacks corren **todos**, en orden

Callbacks

El gancho: una clase, un método virtual vacío

```
class driver_callback extends uvm_callback;
   `uvm_object_utils(driver_callback)

   function new(string name = "driver_callback");
      super.new(name);
   endfunction : new

   // A task, not a function: a callback that only edits the transaction could
   // be a function, but one that adds delay needs to consume time. Making the
   // hook a task costs nothing and leaves both open.
   virtual task pre_send(driver drv, command_transaction cmd);
   endtask : pre_send

endclass : driver_callback
```

- Un `uvm_callback` **no es un component**: no está en el árbol, no tiene fases y no aparece en `print_topology()` — igual que una sequence
- El método base está **vacío a propósito**: sin nadie registrado, el testbench corre exactamente como antes
- Es una `task` y no una `function` porque un callback de *delay* tiene que poder consumir tiempo. Uno que sólo edita la transaction podría ser función

Callbacks

Dos líneas en el driver, y ninguna más

```
class driver extends uvm_driver #(command_transaction);
   `uvm_component_utils(driver)
   // Declares "this component accepts callbacks of this type". Without it the
   // add() below still hooks up and the callback still runs, but with a CBUNREG
   // warning and without the type check: outside the contract, and nearly silent.
   `uvm_register_cb(driver, driver_callback)

   virtual vtalu_bfm bfm;
```

```
   task run_phase(uvm_phase phase);
      command_transaction command;
      forever begin : command_loop
         seq_item_port.get_next_item(command);
         // Every callback registered on this driver, in the order they were
         // added. With none registered this is a loop over an empty queue.
         `uvm_do_callbacks(driver, driver_callback, pre_send(this, command))
         bfm.send_op(command.A, command.B, command.op);
         seq_item_port.item_done();
      end : command_loop
   endtask : run_phase
```

- `` `uvm_register_cb `` declara el par tipo/callback. Sin él el `add()` engancha igual y el callback **corre** — con un `UVM_WARNING CBUNREG` perdido en el log, y sin el chequeo de tipos ni el `add_by_name`
- `` `uvm_do_callbacks `` es el punto de extensión: recorre la cola de **esa instancia** de driver, en orden

Callbacks

Ponerlos: el test, y nada más que el test

```
// end_of_elaboration and not build: the driver is a grandchild of the env,
// and build_phase runs top-down -- when this test's build_phase returns, the
// driver does not exist yet.
function void end_of_elaboration_phase(uvm_phase phase);
   super.end_of_elaboration_phase(phase);

   flip_h   = flip_bit_cb::type_id::create("flip_h");
   jitter_h = jitter_cb::type_id::create("jitter_h");

   // Both on the same driver, and both run: the queue is per component
   // instance, in insertion order. The passive agent has no driver, so there
   // is nothing to hook there.
   uvm_callbacks #(driver, driver_callback)::add(env_h.clase_agent_h.driver_h, jitter_h);
   uvm_callbacks #(driver, driver_callback)::add(env_h.clase_agent_h.driver_h, flip_h);

   if ($test$plusargs("CALLBACK_TRACE"))
      uvm_callbacks #(driver, driver_callback)::display(env_h.clase_agent_h.driver_h);
endfunction : end_of_elaboration_phase
```

- El test **no toca** el env, ni el agent, ni el driver, ni la sequence. Igual que el override, un nivel más abajo
- `end_of_elaboration_phase` y no `build_phase`: el driver es nieto del env, y cuando el `build_phase` del test termina **todavía no existe**
- Los dos callbacks van sobre la **misma** instancia y corren **los dos**, en el orden en que se agregaron

`$ bash run.sh +CALLBACK_TRACE` imprime la cola del driver

Callbacks

El bit dado vuelta que el scoreboard no ve

```
$ cd code/u7/callbacks && bash run.sh
```

- El `flip_bit_cb` da vuelta un bit de `A` una de cada ocho transactions. El scoreboard **no dice nada**, y las dos corridas cierran en 0 `UVM_ERROR`
- No es un agujero del corrector: es la prueba de que **el monitor mira el cable**, no lo que el driver creía que iba a mandar
- Un monitor que reconstruyera la transaction desde el driver —o que la copiara del sequencer— habría dado *PASS* sobre un dato que el DUT nunca vio
- Lo que sí se ve del `jitter_cb` es el **tiempo**: mismo estímulo, más ciclos

El gancho	Cambia	Cuántos a la vez
Constraint	los valores	los que quieras
Factory override	la clase entera	**uno**
Callback	qué hace una clase en un punto	los que quieras

Sequences

Lo único que quedó cableado

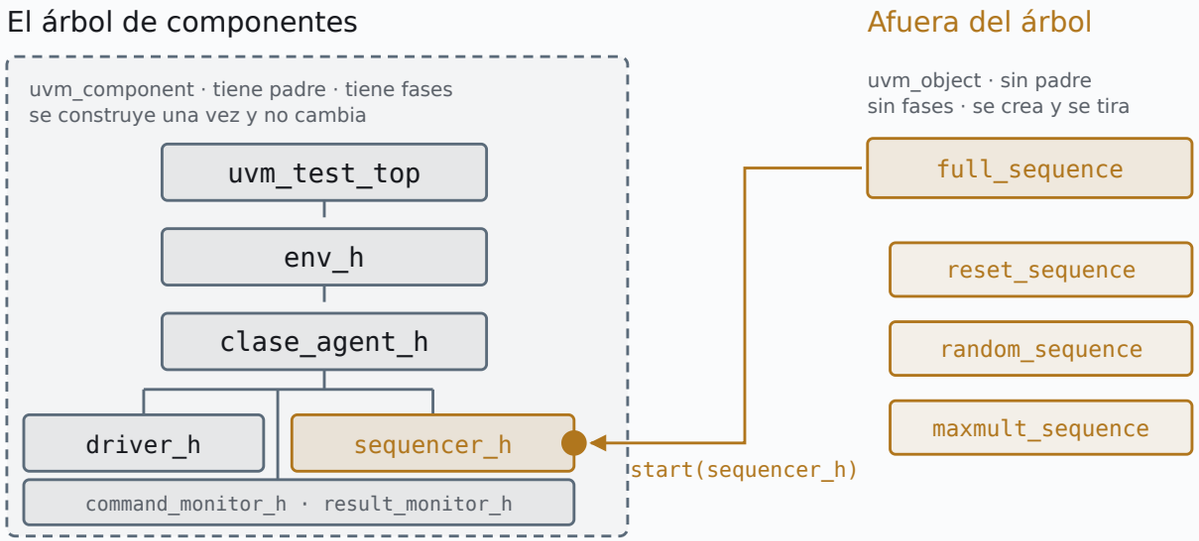
- Los agents dejaron el testbench encapsulado: un `agent` por interface, el análisis en el `env`, y nadie nombrando componentes ajenos
- Menos una línea, la del `run_phase` del test:

```
seq.start(env_h.clase_agent_h.sequencer_h);
```

- El test **atraviesa la jerarquía** para llegar al sequencer, y de paso conoce el interior del `env` que acabamos de cerrar
- Y hay algo más grande atrás: ese `command_sequence` es **una** clase que hace **tres** cosas — resetear, mandar mil operaciones al azar y una dirigida
- Esta sección es sobre lo que pasa adentro de `body()`, y sobre cómo se arranca una sequence sin cablearla

Sequences

Un objeto, no un componente



Aunque no esté en el árbol, UVM la reporta colgada del sequencer, con `@@` en el medio:

`uvm_test_top.env_h.clase_agent_h.sequencer_h@@full_seq.random_seq`

- El `uvm_sequencer` es un `uvm_component`: **está en el árbol**, tiene padre, tiene fases, lo construye el `build_phase` del agent
- La `uvm_sequence` es un `uvm_object`: **no está en el árbol**, no tiene padre, no tiene fases, y se crea con un solo argumento — igual que las transactions
- Consecuencia práctica: se crea, corre, termina y se tira. Podés correr otra a continuación, o dos a la vez, sin tocar la estructura

Sequences

La sequence más chica que existe

```
// The smallest sequence there is: a single item.
//
// It is a uvm_object, not a uvm_component: it is not in the tree, it has no
// parent, it has no phases. It gets created, it runs, and it gets thrown away.
class reset_sequence extends uvm_sequence #(command_transaction);
   // `uvm_object_utils, NOT `uvm_component_utils: the constructor takes a single
   // argument and there is no parent to pass.
   `uvm_object_utils(reset_sequence)

   function new(string name = "reset_sequence");
      super.new(name);
   endfunction : new

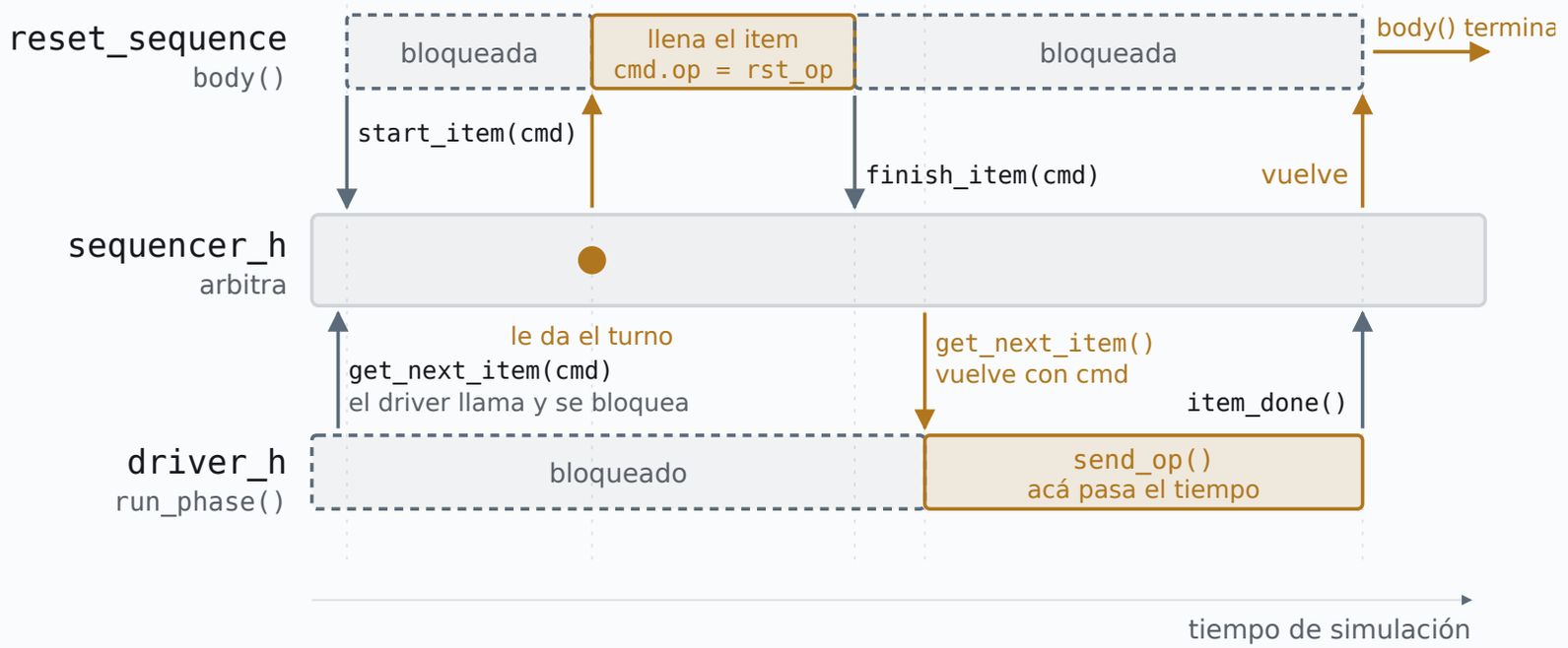
   // body() is a TASK, not a function: it blocks inside. Nobody calls it by
   // hand; UVM calls it when somebody starts the sequence.
   task body();
      command_transaction command;
      command = command_transaction::type_id::create("command");
      start_item(command);    // blocks until the sequencer gives us our turn
      command.op = rst_op;    // only now do we decide what to send
      finish_item(command);   // blocks until the driver calls item_done()
   endtask : body

endclass : reset_sequence
```

- Extiende `uvm_sequence #(T)`, parametrizada con el item que va a mandar
- Se registra con `` `uvm_object_utils `` — **no** con `` `uvm_component_utils ``
- Constructor de un solo argumento: `name`, sin `parent`
- Todo el trabajo vive en `body()`, que es una **task**: puede bloquear
- Nadie llama a `body()` a mano: lo llama UVM cuando alguien arranca la sequence

Sequences

El handshake, ahora desde el lado de la sequence



El tiempo sólo avanza en la barra ámbar del driver: llenar el item no cuesta un flanco.
Si falta el `item_done()`, la sequence queda en `finish_item()` para siempre.

- Las mismas cuatro llamadas de los agents, mirando el otro carril: `start_item()` bloquea a la **sequence** hasta que el sequencer le da el turno, y `finish_item()` la bloquea hasta el `item_done()` del driver
- El tiempo de simulación avanza **sólo** en la barra ámbar: llenar el item no cuesta un flanco de reloj
- Dos hilos, cuatro llamadas y ningún `#delay` en el medio

Sequences

Lo que pasa entre `start_item()` y `finish_item()`

```
task body();
   command_transaction command;

   repeat (count) begin : random_loop
      // Through the factory, same as in the Transactions section: that is what lets
      // add_test change the TYPE without touching a line of this sequence.
      command = command_transaction::type_id::create("command");

      start_item(command);

      // Late randomization: the values are picked AFTER getting the
      // sequencer's turn, not when the object was created.
      if (!command.randomize())
         `uvm_fatal("RANDOM SEQUENCE", "randomize() failed")

      finish_item(command);
   end : random_loop
endtask : body
```

- Cuando `start_item()` vuelve, la sequence **ya tiene el turno** del sequencer: nadie más le va a ganar el driver
- Recién ahí se llenan los campos, o se randomiza. Eso es la **randomización tardía**: los valores se eligen en el último momento posible
- Sirve porque a esa altura ya se sabe en qué estado está el DUT, y una constraint puede depender de eso
- `randomize()` sigue yendo con su `else`: devuelve 0 y no aborta nada — es lo mismo que vieron en la unidad de Constrained Random

Sequences

Randomización tardía dirigida, y el agujero de Verilator

```
start_item(command);
command.data.constraint_mode(0);                        // 1. apago el dist
if (!command.randomize() with {A inside {[1:10]};})     // 2. pido lo dirigido
   `uvm_fatal("SEQ", "randomize() fallo")
finish_item(command);
```

- Medido con Verilator 5.052, sobre una clase con las constraints de `command_transaction`, 100 randomizaciones de cada forma:

```
with {op == mul_op}                      100 de 100   op no tiene dist
with {A inside {[1:10]}}                   1 de 100   A si lo tiene
constraint_mode(0) + with {A inside ...}  100 de 100   el rodeo
```

- La regla, en una línea: **si el `with {}` toca un campo que tiene `dist`, apagá antes esa constraint**
- No es del lenguaje, es de la herramienta: `repro-dist-with.sv` en `code/verilator/`

Sequences

El item vuelve con el resultado adentro

```
task run_phase(uvm_phase phase);
   command_transaction command;
   shortint unsigned   alu_result;
   forever begin : command_loop
      seq_item_port.get_next_item(command);  // blocks until there is an item
      bfm.send_op(command.A, command.B, command.op, alu_result);
      command.result = alu_result;  // the way back
      seq_item_port.item_done();  // unblocks the finish_item()
   end : command_loop
endtask : run_phase
```

- El driver escribe `command.result` **antes** de llamar a `item_done()`
- La sequence todavía tiene el handle a ese mismo objeto: cuando `finish_item()` vuelve, el resultado está ahí
- Es el camino de vuelta, y por eso `command_transaction` gana un campo `result` en esta unidad — el único cambio a la transaction desde que se creó
- Rompe *MOOCOW* a propósito: el driver modifica un objeto que no creó. Está acordado entre las dos partes, y es el único lugar del testbench donde se hace

Sequences

El camino de vuelta formal: `get_response()`

```
// La sequence declara DOS tipos, y la respuesta ya no es el request
class fib_sequence extends uvm_sequence #(command_transaction, result_transaction);
   start_item(cmd);
   finish_item(cmd);
   get_response(rsp);              // bloquea hasta que el driver conteste

// Y el driver, del otro lado:
   seq_item_port.get_next_item(req);
   rsp = result_transaction::type_id::create("rsp");
   rsp.set_id_info(req);           // sin esto la respuesta no encuentra su sequence
   seq_item_port.item_done(rsp);   // o item_done() ahora y put_response(rsp) después
```

- El handle compartido de la slide anterior vale mientras el driver sea **bloqueante y atienda de a uno**. El del curso lo es
- Un driver *pipelined* llama `item_done()` apenas mete el comando y la respuesta llega N ciclos más tarde: para entonces la sequence ya mandó otro item, y escribir en `req.result` le escribe **al item equivocado**
- `set_id_info(req)` le copia a la respuesta el id de sequence y de transacción. Es lo que hace que `get_response()` sepa a quién contestarle
- Los otros dos casos donde hace falta: un VIP que **clona** el item, y un protocolo con **más de una respuesta** por request

Sequences

Fibonacci: cuando el estímulo depende del resultado

```
task body();
   byte unsigned n_menos_2 = 0;
   byte unsigned n_menos_1 = 1;
   command_transaction command;

   command = command_transaction::type_id::create("command");
   start_item(command);
   command.op = rst_op;
   finish_item(command);

   `uvm_info("FIBONACCI", "Fib(01) = 000", UVM_MEDIUM)
   `uvm_info("FIBONACCI", "Fib(02) = 001", UVM_MEDIUM)

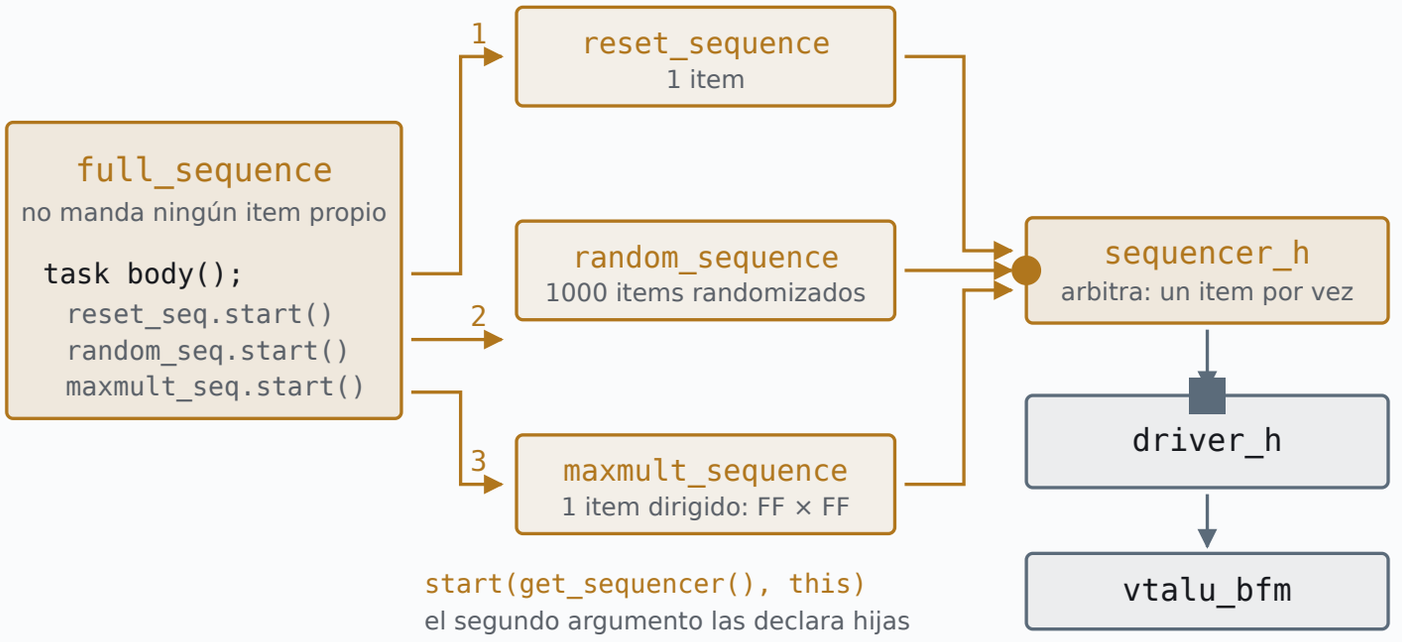
   // Up to 14 and no further: Fib(14) = 233 and Fib(15) = 377, which does not
   // fit in the 8 bits of A and B.
   for (int ff = 3; ff <= 14; ff++) begin : fib_loop
      command = command_transaction::type_id::create("command");
      start_item(command);
      command.A  = n_menos_2;
      command.B  = n_menos_1;
      command.op = add_op;
      finish_item(command);        // returns when the driver called item_done()

      n_menos_2 = n_menos_1;
      n_menos_1 = command.result;  // the driver wrote it inside the item
      `uvm_info("FIBONACCI", $sformatf("Fib(%02d) = %03d", ff, n_menos_1),
                UVM_MEDIUM)
   end : fib_loop
endtask : body
```

- Cada suma necesita el resultado de la anterior: sin camino de vuelta esto no se puede escribir
- `finish_item()` vuelve **después** de que la ALU terminó, así que `command.result` ya es válido
- Es el mismo `env`, el mismo agent, el mismo driver y el mismo BFM que el test random. Cambia sólo la sequence

Sequences

Sequences que llaman sequences



En serie, como está dibujado.
Con `fork` / `join` alrededor de los `start()`, el sequencer entrelaza los items.

- Una sequence puede arrancar otras: es la forma de componer estímulo sin duplicar código
- El `command_sequence` de los agents, partido en tres piezas que ahora se combinan como uno quiera
- El sequencer arbitra: reciba lo que reciba, al driver le entrega **un item por vez**

Sequences

full_sequence: tres piezas, un solo sequencer

```
task body();
   reset_sequence   reset_seq;
   random_sequence  random_seq;
   maxmult_sequence maxmult_seq;

   reset_seq   = reset_sequence::type_id::create("reset_seq");
   random_seq  = random_sequence::type_id::create("random_seq");
   maxmult_seq = maxmult_sequence::type_id::create("maxmult_seq");

   random_seq.count = count;

   // get_sequencer() returns the sequencer start() handed us: the children run
   // on the same one. The second argument declares them CHILDREN -- without it
   // they compete with us in the arbitration instead of inheriting our turn.
   reset_seq.start(get_sequencer(), this);
   random_seq.start(get_sequencer(), this);
   maxmult_seq.start(get_sequencer(), this);
endtask : body
```

- `get_sequencer()` devuelve el sequencer que le pasó `start()` a **esta** sequence: las hijas corren sobre el mismo
- El segundo argumento, `this`, las declara **hijas**. Sin él compiten con la madre en la arbitración en vez de heredar su turno
- `random_seq.count = count` es la ventaja de que la sequence sea un objeto: se configura entre el `create()` y el `start()`
- Para correr dos en paralelo, `fork` / `join` alrededor de los `start()`, y el sequencer entrelaza los items

Sequences

Arrancar una sequence, forma 1: `start()`

```
task run_phase(uvm_phase phase);
   full_sequence full_seq;
   full_seq = full_sequence::type_id::create("full_seq");

   // THE TEST raises the objection, not the sequence: the sequence knows
   // nothing about phases, and has to be able to run inside another sequence.
   phase.raise_objection(this);
   full_seq.start(sequencer_h);   // does not return until body() finished
   phase.drop_objection(this);
endtask : run_phase
```

- `start(sequencer)` le pasa a la sequence el sequencer sobre el que va a correr y **no vuelve hasta que `body()` terminó**
- El test levanta el objection antes y lo baja después: la corrida dura exactamente lo que dura el estímulo
- Es explícito y se lee de arriba a abajo. Para un test que corre una sola sequence, es lo que conviene
- Y sigue teniendo el problema de la primera slide: `sequencer_h` salió de `env_h.clase_agent_h.sequencer_h`

Sequences

Forma 2: `default_sequence` por `uvm_config_db`

```
function void build_phase(uvm_phase phase);
   full_sequence full_seq;

   // super.build_phase() of base_test, which is a class of OURS: it builds the
   // env. What the course never calls is uvm_component's build_phase.
   super.build_phase(phase);

   full_seq = full_sequence::type_id::create("full_seq");
   full_seq.count = 200;

   // WITHOUT this line main_phase ends at t=0: nobody raises the objection,
   // and a phase without an objection lasts zero. The test PASSES with no stimulus.
   full_seq.set_automatic_phase_objection(1);

   // The instance name carries the "_phase" suffix: that is the phase the
   // sequencer will start it in. Notice the test no longer names the sequencer
   // as a handle: it names it as a PATH, and that is a config, not code.
   uvm_config_db#(uvm_sequence_base)::set(
       this, "env_h.clase_agent_h.sequencer_h.main_phase", "default_sequence",
       full_seq);
endfunction : build_phase
```

- El test **no tiene** `run_phase`: el sequencer arranca la sequence solo, al empezar la fase
- El nombre de instancia lleva el sufijo `_phase`: `"...sequencer_h.main_phase"` quiere decir *"en la `main_phase` de ese sequencer"*
- Se puede configurar por **instancia** (como acá, con `uvm_config_db #(uvm_sequence_base)`) o por **tipo** (`uvm_config_db #(uvm_object_wrapper)` + `get_type()`). Si están las dos gana la de **mayor precedencia**, y a igual precedencia el **último** `set()`
- El sequencer dejó de ser un handle y pasó a ser una **ruta**: eso ya es configuración, no código

Sequences

Cuándo cada una

	`seq.start(sqr)`	`default_sequence`
Dónde se escribe	`run_phase` del test	`build_phase` del test, o el `top`
Objection	lo maneja el test	`set_automatic_phase_objection(1)`
Se lee	de arriba a abajo	hay que buscarlo
Varias sequences en orden	trivial	una por fase
Cambiarla sin recompilar	no	sí, es una config
Conoce el interior del env	sí, por el handle	no, sólo la ruta
Agent reusable de terceros	no siempre se puede	es el mecanismo previsto

- Para los tests del curso: `start()`, porque se ve
- Para un agent que le entregás a otro equipo: `default_sequence`, porque le deja cambiar el estímulo sin tocar tu código

Sequences

La macro `uvm_do y por qué el curso no la usa

```
`uvm_do(command)                          // una linea

command = command_transaction::type_id::create("command");   // cuatro
start_item(command);
if (!command.randomize()) `uvm_fatal("SEQ", "randomize() fallo")
finish_item(command);
```

- `` `uvm_do `` hace exactamente esas cuatro cosas: crear, `start_item`, randomizar, `finish_item`
- Lo que esconde: el `randomize()` **sin else**, así que una constraint sin solución pasa en silencio
- Lo que impide: llenar campos a mano entre las dos llamadas — o sea, todo lo de las slides anteriores
- Lo que rompe: el mensaje de error apunta a la macro expandida, no a tu código
- Existe una familia entera (`` `uvm_do_with ``, `` `uvm_create ``, `` `uvm_send ``) y el mismo argumento vale para todas

Sequences

El override sigue funcionando, y ninguna sequence se enteró

```
// The same add_test as the Transactions section, word for word. No sequence finds
// out: random_sequence still asks the factory for a command_transaction,
// and the factory hands it back an add_transaction.
class add_test extends full_test;
   `uvm_component_utils(add_test)

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      command_transaction::type_id::set_type_override(add_transaction::get_type());
      super.build_phase(phase);
   endfunction : build_phase

endclass : add_test
```

- Es el mismo `add_test` de las transactions, palabra por palabra
- `random_sequence` sigue creando `command_transaction::type_id::create()`; la factory le devuelve `add_transaction`
- Resultado medido: 1000 operaciones, **todas** `add_op`, 0 `UVM_ERROR`
- `maxmult_sequence` sigue mandando `mul_op` bajo `add_test` — porque no randomiza, y **la constraint sólo actúa en `randomize()`**

Sequences

Dos sequences sobre el mismo sequencer: la arbitración

- Hasta acá cada sequencer tuvo **una** sequence por vez. En un testbench real hay varias en paralelo sobre el mismo driver —tráfico de fondo, un test dirigido, un inyector de errores— y alguien tiene que repartir los turnos

```
fork
   fondo_seq.start(sqr);                // prioridad 100, la de por defecto
   urgente_seq.start(sqr, null, 500);   // el tercer argumento es la prioridad
join
```

El modo, en `sqr.set_arbitration(...)`	**Cómo reparte**
`UVM_SEQ_ARB_FIFO`	orden de llegada, y **la prioridad no se mira** (default)
`UVM_SEQ_ARB_STRICT_FIFO`	la prioridad manda; a igual prioridad, orden de llegada
`UVM_SEQ_ARB_WEIGHTED`	al azar, con la prioridad como peso
`UVM_SEQ_ARB_RANDOM`	al azar, todas igual

El modo, en `sqr.set_arbitration(...)`	**Cómo reparte**
`UVM_SEQ_ARB_STRICT_RANDOM`	al azar, pero sólo entre las de mayor prioridad
`UVM_SEQ_ARB_USER`	`user_priority_arbitration()`, la escribís vos

- El default **ignora la prioridad**: poner un 500 y no cambiar el modo es el error de esta slide, y no avisa nadie

Sequences

`lock()` y `grab()`: cuando la arbitración no alcanza

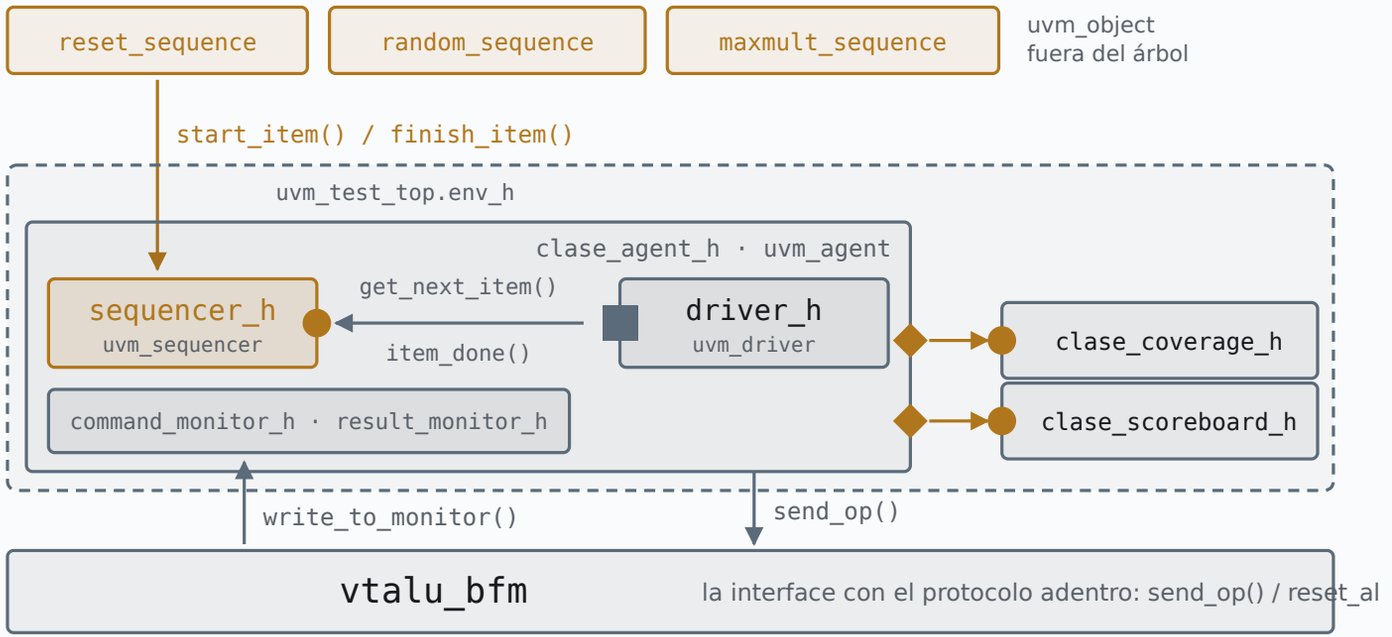
- Hay escenarios que **no se pueden intercalar**: una secuencia de configuración, un burst atómico, un lee-modifica-escribe. Si otra sequence mete un item en el medio, el escenario dejó de ser el que escribiste

```
task body();
   grab();                  // se cuela adelante de todas y cierra la puerta
   ... los items del escenario, sin nadie en el medio ...
   unlock();                // ungrab() es el mismo metodo, con otro nombre
endtask
```

- **`lock()` hace cola**: espera su turno como cualquiera y recién entonces cierra la puerta. **`grab()` se cuela**: pasa adelante de todas, incluso de las de más prioridad
- Los dos se sueltan con `unlock()`. Sin el `unlock()`, si la sequence **sigue viva** el sequencer queda cerrado para siempre y sin un solo error; si su `body()` termina, UVM lo saca y grita `SEQFINERR`
- La regla: `grab()` para reset y recuperación de error —lo que no puede esperar—, `lock()` para todo lo demás, y el `unlock()` en la misma `body()`

Sequences

El testbench, completo



Todo lo del recuadro punteado se construye una vez y no cambia.
Lo que cambia entre un test y otro es la fila de arriba.

- Es el diagrama que se dibuja hoy en cualquier proyecto de verificación, con los nombres que se usan hoy
- Todo lo que está adentro del `env` se construye una vez y no cambia. Lo que cambia entre un test y otro es la nube de sequences de arriba

Sequences

Resumen de la unidad · la sequence y el item

- El estímulo salió del árbol de componentes: es un `uvm_object`, no un `uvm_component`, y por eso se crea, se configura, corre y se tira
- Todo pasa adentro de `body()`, entre `start_item()` y `finish_item()`: ahí se llena el item, ahí va el `randomize()`, y ahí se enganchan `pre_do` y `mid_do`
- El driver escribe el resultado **adentro del item** antes de `item_done()`: es el camino de vuelta, y es lo que hace posible un estímulo que reacciona
- Dos formas de arrancar: `start(sequencer)` explícito, o `default_sequence` por `uvm_config_db` — que además saca la última línea cableada del testbench

Sequences

Resumen de la unidad · cómo se componen

- Las sequences se componen: una llama a otras, en serie o con `fork`/`join`
- Y cuando hay **más de un sequencer**, la que compone es una **sequence virtual**, que llega apenas empieza el día 7 ↪ día 7
- Con **dos sequences sobre el mismo sequencer** el que reparte es el sequencer: prioridad y `set_arbitration(...)`, y `lock()`/`grab()` para el escenario que no se puede intercalar
- Con esto el testbench del curso está completo —le falta una sola pieza, la de mañana— y es la forma en que se escribe hoy en la industria

Repaso · Día 6

1 de 10 · is_active

Un agent en `UVM_PASSIVE`, ¿qué construye su `build_phase`?

- **a** Los monitores y los analysis ports; el driver y el sequencer, no ✓
- **b** Nada: un agent pasivo es una cáscara vacía
- **c** Todo igual que uno activo, pero sin conectar el driver al sequencer
- **d** Sólo el sequencer, para poder recibir sequences de otro agent del mismo env

Los monitores, siempre; el sequencer y el driver quedan en `null` — mirar nunca es opcional: un agent pasivo sigue alimentando scoreboard y cobertura. Lo que se saltea es lo que *maneja* la interface, porque ahí ya hay otro manejándola.

Repaso · Día 6

2 de 10 · El handshake del driver

El driver llama a `get_next_item()`, maneja las señales y se olvida del `item_done()`. ¿Qué pasa?

a. Error de compilación: UVM exige el par completo

b. El sequencer entrega el siguiente item igual, con un warning

c. El item se descarta y el scoreboard reporta un mismatch en la comparación siguiente

d. La sequence se queda esperando en `finish_item()` y no avanza más ✓

Se cuelga en `finish_item()` y sin decir nada — es el error clásico de la primera semana, y el síntoma engaña: no hay error ni warning, el tiempo deja de avanzar y el objection nunca se baja. `get_next_item()` es un préstamo; `item_done()` es devolverlo.

Repaso · Día 6

3 de 10 · Ámbito del config_db

El `env` instancia dos agents y hace los dos `set()` con el ámbito `"*"`. ¿Qué recibe cada uno?

a. Los dos fallan con `uvm_fatal`: el string `"config"` está duplicado

b. Cada uno recibe el suyo, por orden de creación

c. Los dos reciben el mismo: el segundo `set()` pisa al primero ✓

d. El primero recibe su config y el segundo queda con `cfg == null`

Los dos reciben lo mismo — el `uvm_config_db` no empareja por orden ni por tipo: empareja por **ruta**. Con `"*"` las dos entradas describen a los mismos componentes, así que la última gana. El ámbito es una ruta en el árbol, no una etiqueta.

Repaso · Día 6

4 de 10 · El sequencer

`typedef uvm_sequencer #(command_transaction) sequencer;` compila sin tocar la transaction de las transactions. ¿Por qué?

a. Porque está registrada en la factory con `` `uvm_object_utils ``

b. Porque `uvm_sequencer` acepta cualquier `uvm_object`

c. Porque el driver hace el `$cast` por dentro

d. Porque `command_transaction` extiende `uvm_sequence_item` ✓

Por la clase base que elegimos en transactions — `uvm_sequencer #(T)` exige que `T` derive de `uvm_sequence_item`. Si aquel día la transaction hubiera extendido `uvm_transaction` a secas, este `typedef` hoy no compilaría. Una decisión de una sección habilitando la siguiente.

Repaso · Día 6

5 de 10 · super.build_phase()

En vez del config object, ponés `is_active` directo en el `uvm_config_db`. En este curso el agent arranca activo igual. ¿Por qué?

a Porque `is_active` es `protected` y el `config_db` no lo puede escribir

b Porque el `config_db` no acepta tipos enumerados, sólo `int` y `string`

c Porque `is_active` se fija en el constructor y `build_phase` llega tarde

d Porque quien lo lee es `uvm_agent::build_phase`, y nadie llama a `super` ✓

Ese mecanismo está apagado — `uvm_agent::build_phase` busca `is_active` en el resource pool (está en `code/.uvm/src/comps/uvm_agent.svh`, se puede abrir). Sin `super.build_phase()` esa línea no corre nunca, y nadie avisa. Las dos formas son válidas; lo que no funciona es la mitad de cada una.

Repaso · Día 6

6 de 10 · Object, no component

¿Cuál es la diferencia práctica de que una `uvm_sequence` sea un `uvm_object` y no un `uvm_component`?

- **a** Que se crea, corre y se tira ✓
- **b** Que no se puede registrar en la factory ni overridear
- **c** Que no puede tener campos `rand` ni constraints
- **d** Que UVM la construye en `build_phase`, como a cualquier otra clase del árbol

Se crea, corre y se tira — podés arrancar varias, una detrás de otra, sobre el mismo sequencer. Un componente se construye una vez en `build_phase` y vive hasta el final. Por eso el estímulo no puede ser un componente: cambia test a test, y a veces dentro del mismo test. De ahí sale también que se pueda configurar entre el `create()` y el `start()`, como hace `full_seq.count = 200`.

Repaso · Día 6

7 de 10 · start_item()

`start_item(command)` acaba de volver. ¿Qué es lo que eso garantiza?

a Que el driver ya recibió el item y está manejando las señales

b Que el sequencer le dio el turno a esta sequence ✓

c Que el `randomize()` ya se resolvió con las constraints de la clase

d Que el objection de la fase ya está levantado por el sequencer

Tenés el turno —nadie más te va a ganar el driver—, y todavía no entregaste nada — y por eso el `randomize()` va *después*: es el último momento posible para elegir los valores, cuando ya sabés en qué estado está el DUT. Eso es la randomización tardía, y es donde se enganchan `pre_do()` y `mid_do()`.

Repaso · Día 6

8 de 10 · El camino de vuelta

¿En qué momento `command.result` tiene un valor que se puede leer?

a. Cuando volvió `finish_item()`, porque el driver lo escribió antes de llamar a `item_done()` ✓

b. Apenas volvió `start_item()`

c. Cuando el `result_monitor` lo publica por su analysis port

d. Nunca: para recibir una respuesta hay que usar el par REQ/RSP de `uvm_sequence #(REQ, RSP)`

Después de `finish_item()` — no hay ningún canal de vuelta: hay un handle compartido y un acuerdo entre las dos partes. El par REQ/RSP existe y es el mecanismo formal, pero casi nadie lo usa: escribir el resultado en el request alcanza. Esto es lo que hace posible Fibonacci.

Repaso · Día 6

9 de 10 · default_sequence

Configurás una `default_sequence` por `uvm_config_db` y te olvidás del `set_automatic_phase_objection(1)`. ¿Qué pasa?

a `uvm_fatal` en `build_phase`: la sequence no encuentra el sequencer

b Corre igual: cuando hay `default_sequence`, el objection lo levanta el sequencer solo

c La `main_phase` termina en t=0 y el test pasa con 0 errores ✓

d La simulación se cuelga esperando un objection que nadie baja

Pasa en cero segundos sin mandar un solo estímulo, y miente — una fase sin objection termina en cuanto arranca. Es el mismo tipo de bug que el `new()` que se come el override: no rompe, engaña. Y en una regresión de mil tests, el que pasa en cero segundos no lo mira nadie.

Repaso · Día 6

10 de 10 · Sub-sequences

`full_sequence` arranca a sus hijas con `reset_seq.start(get_sequencer(), this)`. ¿Para qué está el segundo argumento?

a. Para pasarle el sequencer, porque `get_sequencer()` sólo devuelve el tipo

b. Para declarar a la hija sub-sequence de la madre ✓

c. Para que la hija corra en un hilo aparte, en paralelo con la madre

d. Para registrar a la hija en la factory con el nombre de la madre, y poder overridearla

Declara la relación madre-hija — sin él, el sequencer trata a las dos como sequences independientes y compiten por el turno. Con él, la hija hereda el turno de la madre y su prioridad en la arbitración. Y para el paralelo no alcanza con eso: hace falta un `fork` / `join` alrededor de los `start()` .

Ejercicio · Día 6 · 1 de 3

El agent que sólo mira

```
$ cd code/ejercicios/d6-agents && bash run.sh
```

- El `env` tiene un solo agent. Agregá el **pasivo** sobre la segunda VTALU
- Y hacé que `is_active` sirva de verdad: hoy el agent construye driver y sequencer siempre
- Se corrige con `+UVM_CONFIG_DB_TRACE`: el censo del árbol lo escribe UVM, no vos

Ejercicio · Día 6 · 2 de 3

Una sequence que sólo multiplica

```
$ cd code/ejercicios/d6-sequences && bash run.sh
```

- Escribí `mult_sequence` y el `mult_test` que la arranca
- El driver, el agent y el `env` ya están: agregar estímulo **no toca la estructura**, que es justamente lo que la sección vino a demostrar
- Se corrige cruzado contra lo que el `chequeo` ve en el bus, no contra tu cuenta

Ejercicio · Día 6 · 3 de 3

Tres bugs plantados, y ninguno se parece al otro

```
$ cd code/ejercicios/d6-debug && bash run.sh
```

	Archivo	Cómo se manifiesta
1	`driver.svh`	**cuelga**: muere en `[PH_TIMEOUT]`
2	`default_seq_test.svh`	**termina en `t=0`** y dice PASS
3	`random_sequence.svh`	**miente en verde**: `add_test` pasa con el estímulo equivocado

- El corrector dice **cuál** de los tres sigue roto. Cuál es la línea, es el ejercicio

Ayer quedó...

Dónde dejamos el testbench

- El testbench quedó **reutilizable para un agent**: agents con `is_active`, el análisis en el `env`, y el estímulo afuera del árbol, en sequences. La última pieza —coordinar dos— es hoy a la mañana
- Ninguna línea del test nombra un componente de adentro del `env`: la última se fue con el `default_sequence` por `uvm_config_db`
- Y falta la otra mitad del plan de verificación: las filas que **ningún scoreboard puede cerrar**, porque no son sobre el resultado sino sobre el protocolo

Agenda

Día 7 · ≈ 5 h 15 · unidad 8 · la otra mitad

- Calentamiento: cinco semillas y un merge, con `make regresion`
- Sequences virtuales: la última pieza del testbench reutilizable
- Assertions (SVA)
- El capstone: un testbench de APB, desde cero
- De la VTALU a un bus real
- La caja de herramientas de debug · Las 21 trampas mudas
- Glosario, referencias y cierre
- Autoevaluación: quince cosas que a esta altura tendrías que poder hacer
- *El lunes*: qué hacer con esto cuando se apaga la pantalla

*Y después, el **día 8**: opcional, y para el que ya entregó el capstone*

Al final del día podés:

- **Escribir** una property que vea una violación de protocolo, y decir por qué `--assert` no es opcional
- **Coordinar** dos interfaces desde una sola sequence virtual
- **Entregar** un testbench entero para un DUT que no viste antes

Ejercicio · Día 7 · 1 de 2

La misma sequence, otra semilla

```
$ cd code/ejercicios/d7-semillas && bash run.sh
```

- El único ejercicio del curso sin SystemVerilog: lo que se escribe es la **regresión**, seis líneas de shell
- Correr el mismo test con las semillas 1 a 5 y **mergear** las cinco coberturas con `verilator_coverage --write`
- Se corrige comprobando que el merge cubre más que la mejor de las cinco sola

Sequences virtuales

El problema: dos interfaces y un solo `start()`

- Una **sequence virtual** es una sequence que no manda items propios: sólo arranca otras sequences, sobre **varios sequencers**
- Aparece cuando el DUT tiene más de una interfaz —un bus de configuración y un stream de datos, por ejemplo— y el test necesita coordinarlas: *configurá primero, después mandá tráfico*
- Se le llama "virtual" porque no está atada a un tipo de item ni a un sequencer
- El `u7/agents` ya nos dejó el escenario armado: **dos VTALU**. Ahí el segundo agent era pasivo; con los dos activos hay **dos sequencers**, y algo que coordinar
- `full_sequence` ya es media sequence virtual: no manda items propios, sólo arranca otras. Lo único que le falta es el segundo sequencer

Sequences virtuales

El virtual sequencer: handles, no items

```
class virtual_sequencer extends uvm_sequencer;
   `uvm_component_utils(virtual_sequencer)

   sequencer clase_sequencer_h;
   sequencer modulo_sequencer_h;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass : virtual_sequencer
```

```
      // The handles are passed HERE and not looked up by string: in connect_phase
      // both agents already exist, and if somebody renames a component this does
      // not compile instead of returning null at simulation time.
      virtual_sequencer_h.clase_sequencer_h  = clase_agent_h.sequencer_h;
      virtual_sequencer_h.modulo_sequencer_h = modulo_agent_h.sequencer_h;
```

- No tiene cola, no arbitra, no habla con ningún driver. Es un `uvm_component` que existe para **tener los handles** y para vivir en el árbol con un nombre
- Extiende `uvm_sequencer` **sin parametrizar**: el item por defecto es `uvm_sequence_item` y nunca se manda ninguno
- Los handles se asignan en el `connect_phase`, donde los agents ya existen

Sequences virtuales

La sequence virtual: `p_sequencer` y dos ramas

```
// `uvm_declare_p_sequencer declares `p_sequencer` with the type above and casts
// it on its own inside m_set_p_sequencer. Without it, get_sequencer() returns a
// uvm_sequencer_base and every use would need a cast by hand.
class coordinada_sequence extends uvm_sequence;
   `uvm_object_utils(coordinada_sequence)
   `uvm_declare_p_sequencer(virtual_sequencer)
```

```
      // --- 1. Both ALUs get reset at the same time -----------------------------
      // fork/join over TWO sequencers. Each branch blocks on its own driver, and
      // the join waits for both: that cannot be written inside a normal
      // sequence, which only knows the sequencer that started it.
      fork
         reset_a.start(p_sequencer.clase_sequencer_h);
         reset_b.start(p_sequencer.modulo_sequencer_h);
      join

      // --- 2. Parallel traffic, different on each interface -------------------
      // A does the directed overflow case while B sends at random.
      fork
         maxmult.start(p_sequencer.clase_sequencer_h);
         random_b.start(p_sequencer.modulo_sequencer_h);
      join
```

- `` `uvm_declare_p_sequencer `` declara `p_sequencer` **con el tipo del sequencer virtual** y lo castea solo. Sin él, `get_sequencer()` devuelve un `uvm_sequencer_base` y hay que castear a mano en cada uso
- El `fork`/`join` es sobre **dos sequencers**: cada rama bloquea en su driver y el `join` espera a las dos
- Eso no se puede escribir adentro de una sequence normal, que conoce un solo sequencer

Sequences virtuales

Lo que ninguna sequence sola puede

```
// --- 3. What no single sequence can do ---------------------------------
// The result of VTALU A is the operand of B. It is a dependency BETWEEN
// interfaces: the second one cannot even start being built until the
// first one answered.
primera.A  = 8'h0F;
primera.B  = 8'h07;
primera.op = mul_op;
primera.start(p_sequencer.clase_sequencer_h);

`uvm_info("COORDINADA", $sformatf(
          "A returned %0d: it goes as an operand into B", primera.result), UVM_LOW)

segunda.A  = primera.result[7:0];
segunda.B  = 8'h01;
segunda.op = add_op;
segunda.start(p_sequencer.modulo_sequencer_h);
```

- El resultado de la VTALU **A** entra como operando de la **B**: una dependencia **entre interfaces**, y en serie. La segunda no puede ni armarse hasta que la primera contestó
- El `result` vuelve adentro del item, que es el camino de vuelta de esta unidad
- `0F × 07 = 105`, y la B contesta `106`. Corre: `code/u7/sequences/virtual/run.sh`

Sequences virtuales

¿Y si la rama B tiene que esperar a la A?

```
// uvm_event: un aviso, de uno a muchos. Sale del pool global, nadie lo construye
uvm_event listo = uvm_event_pool::get_global("dut_configurado");

listo.trigger();          // la rama que configuró avisa
listo.wait_trigger();     // la que espera bloquea acá
listo.wait_ptrigger();    // igual, pero si ya pasó, sigue de largo

// uvm_barrier: un punto de encuentro entre N ramas
uvm_barrier arranque = uvm_barrier_pool::get_global("arranque");
arranque.set_threshold(2);
arranque.wait_for();      // ninguna de las dos pasa hasta que llegaron las dos
```

- El `join` sincroniza **al final**. Cuando la rama B tiene que esperar a la A **en el medio**, hace falta un objeto de sincronización
- `uvm_event` es de **uno a muchos**: uno avisa, los que esperan siguen. Y `trigger(data)` puede llevar un `uvm_object` adjunto
- `uvm_barrier` es **simétrico**: nadie pasa hasta que llegaron todos. Es el "arranquen juntos" de dos agents que tienen que empezar en el mismo ciclo
- Un `bit` compartido con `wait(flag)` no alcanza: **pierde el pulso** si el que espera llegó tarde, y no lleva dato

Assertions (SVA)

El agujero que dejó el scoreboard

- El scoreboard de este curso chequea **el resultado**: `A op B` contra lo que devolvió el DUT. Mil operaciones, mil comparaciones, cero errores
- Nadie chequea **el protocolo**. Un DUT que devuelve bien el `result` pero baja `done` un ciclo antes, o lo levanta con `no_op`, **pasa todos los tests del curso**
- La regla del protocolo está escrita en tres lugares: en prosa en la spec, adentro del BFM, y en la cabeza del que lo escribió. En **ninguno** de los tres se chequea
- Una **assertion** es esa misma frase, en forma ejecutable, corriendo sola toda la simulación

Assertions (SVA)

Inmediatas y concurrentes: la que ya conocen y la nueva

```
// INMEDIATA: es una SENTENCIA. Corre cuando el hilo pasa por ahí,
// una vez, y nada más. La de las transactions es de esta familia.
assert (cmd.op inside {add_op, sub_op, and_op, xor_op, mul_op})
   else `uvm_error("SEQ", "operacion invalida");

// CONCURRENTE — la novedad. Es una DECLARACION con reloj: se enchufa
// al arrancar la simulacion y se evalua en cada flanco, para siempre.
a_done : assert property (@(posedge clk) start |=> ##[0:4] done);
```

- La **inmediata** vive adentro de un `begin/end`, es procedural, y sólo existe mientras el hilo está parado ahí
- La **concurrente** no vive en ningún hilo: es una declaración, como un `always`. Se escribe una vez y chequea cada flanco de la simulación entera
- Por eso una concurrente puede describir algo que **dura varios ciclos**, y una inmediata no

Assertions (SVA)

Anatomía de una property

```
a_done_llega :                      // <- el label: NO es opcional
assert property (
   @(posedge clk)                   // <- el reloj: cuando se muestrea
   disable iff (!reset_n)           // <- cuando NO chequear
   start && (op_set != no_op)       // <- el antecedente
   |->                              // <- la implicacion
   ##[1:5] done                     // <- el consecuente
) else `uvm_error("SVA", "...");    // <- que hacer si falla
```

- Se lee como una frase: *"en cada flanco de `clk`, mientras no haya reset, si `start` está arriba con una operación real, entonces `done` llega dentro de cinco ciclos"*
- El **label** sale en el mensaje de error y en el reporte de cobertura. Una property sin label es un error anónimo a las tres de la mañana
- Las siete partes son siempre las mismas. El resto de la sección es aprender qué se puede poner en cada una

Assertions (SVA)

|-> contra |=>, el error número uno

```
// |->  overlapping: el consecuente empieza en EL MISMO flanco
a : assert property (@(posedge clk) start |-> !done);

// |=>  non-overlapping: el consecuente empieza en el flanco SIGUIENTE
b : assert property (@(posedge clk) start |=> !done);
```

- `a |=> b` es exactamente `a |-> ##1 b`. No hay más misterio que ése
- ¿Cuál va? Depende de si la respuesta es **combinacional** (el mismo flanco) o **registrada** (el siguiente). En el VTALU, `done` sale de un `always_ff`: siempre es el siguiente
- Equivocarse no da un síntoma solo: `start |-> done` con `done` registrado **falla en cada transacción** —ve el `done` viejo—, y una property cuyo antecedente nunca ocurre pasa en vacío. Por eso va siempre con su `cover`

Assertions (SVA)

Mirar el pasado: `$rose`, `$fell`, `$stable`, `$past`

```
$rose(start)      // 0 -> 1 entre el flanco anterior y este
$fell(done)       // 1 -> 0
$stable(A)        // el valor es el mismo que en el flanco anterior
$past(result, 3)  // el valor que tenia hace 3 flancos
```

- Las cuatro comparan **contra el muestreo anterior**, no contra el instante anterior. En SVA el tiempo se cuenta en flancos del reloj de la property, no en nanosegundos
- `$stable` es la que escribe *"no se tocó"*, que es media biblioteca de reglas de protocolo
- `$past(x, n)` es una línea de retardo gratis: sirve para chequear una latencia fija sin escribir una máquina de estados

Assertions (SVA)

La property que vale la sección

```
// --- The stimulus ---

// The rule from slide 1 of day 1, executable at last: while start is up, the
// operands and the operation are not touched.
property p_operandos_estables;
   @(negedge clk) start |=> $stable(A) && $stable(B) && $stable(op_set);
endproperty : p_operandos_estables

a_operandos_estables :
assert property (p_operandos_estables)
else
   `uvm_error("SVA", $sformatf(
              "%m: operand changed with start up: A=%0d B=%0d op=%s", A, B, op2enum().name()))
```

- Es **la regla de *La spec del VTALU*, el día 1**: mientras `start` está arriba, los operandos no se tocan. Estuvo escrita en prosa seis días
- Once líneas, y chequean cada transacción de cada test, en los dos agents, sin que nadie las conecte a nada
- El scoreboard **no puede** escribir esta regla: para cuando la transaction le llega, el protocolo ya se borró

Assertions (SVA)

⚠ *Dos relojes: una assertion vale lo que vale su muestreo*

```
// TWO CLOCKS, and it is not an implementation detail: the BFM drives on
// negedge and the DUT registers on posedge. An assertion samples in the
// preponed region of the edge it is given, so a signal written ON the negedge
// is not seen on that negedge: it is seen on the next one. With everything on
// posedge, two consecutive no_op -- start goes down at t=111 and back up at
// t=120, between two posedges -- read as ONE transaction with the operands
// changing: 145 false positives every 1000 operations. With everything on
// negedge, the false positives move to the done properties.
//
//   stimulus (start, A, B, op_set)  -> negedge, where the BFM writes
//   response (done, result)         -> posedge, where the DUT registers

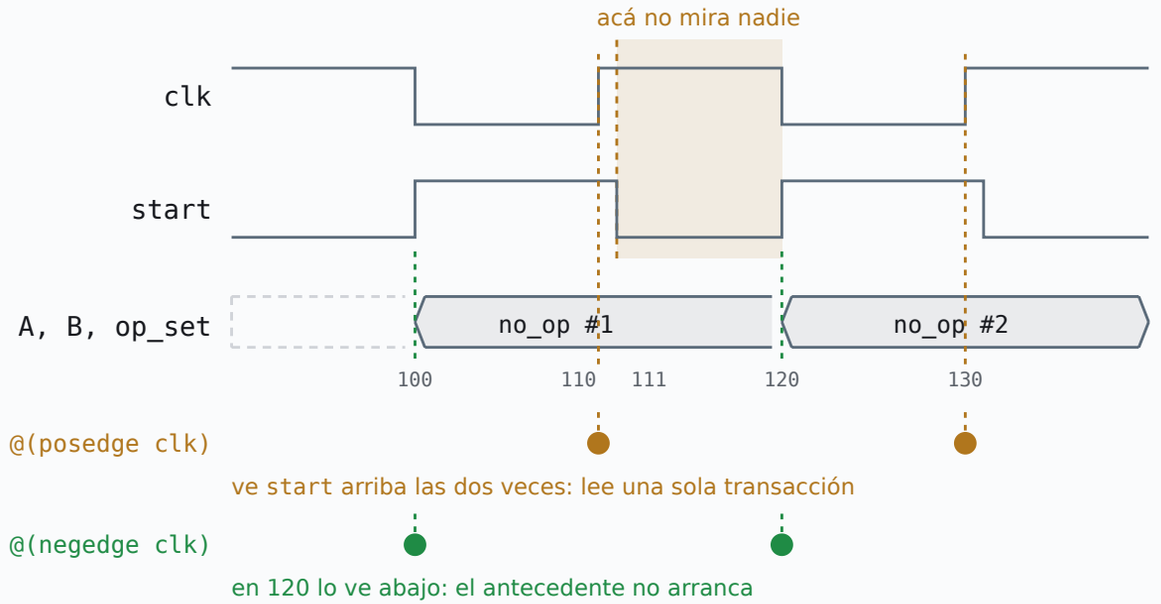
default disable iff (!reset_n);
```

Con un solo reloj	Sobre 1000 operaciones
todo en posedge	**145 falsos positivos** en la estabilidad de operandos
todo en negedge	falsos positivos en las properties de done
estímulo en negedge, respuesta en posedge	**0 errores** ✅

- La BFM maneja en `negedge` y el DUT registra en `posedge`. **No existe un flanco único que haga correctas a las cuatro properties**
- El muestreo de SVA es la región *preponed*: una señal escrita **en** el flanco no se ve en ese flanco, se ve en el siguiente

Assertions (SVA)

El pulso que el muestreo no ve



Dos `no_op` seguidas: `start` baja en 111 y vuelve a subir en 120, entre dos `posedge`.
El muestreo no lo ve bajar: las dos se leen como una sola, con los operandos cambiando en el medio
145 falsos positivos cada mil operaciones, por un solo carácter en el `@()`.

- Entre dos `posedge` cabe un pulso entero de `start` que el muestreo no ve
- Todo en `posedge`: las dos `no_op` se leen como **una sola**, y salen 145 gritos
- El estímulo en `negedge`: en `t=120` ya está abajo, el antecedente no arranca

Assertions (SVA)

La latencia variable, en una línea

```
// The variable latency from the VTALU spec section, in one line: one cycle for the
// one-cycle ops, four edges for the multiplication.
property p_done_llega;
   @(posedge clk) start && (op_set != no_op) |-> ##[1:5] done;
endproperty : p_done_llega

a_done_llega :
assert property (p_done_llega)
else `uvm_error("SVA", $sformatf("%m: start con op=%s y done no llego en 5 ciclos", op2enum().name()))

// The fine print: no_op is the only operation that does not answer.
property p_no_op_sin_done;
   @(posedge clk) start && (op_set == no_op) |=> !done;
endproperty : p_no_op_sin_done

a_no_op_sin_done :
assert property (p_no_op_sin_done)
else `uvm_error("SVA", $sformatf("%m: done subio para una no_op"))
```

- `##[1:5] done` dice *"entre uno y cinco flancos después"*. La VTALU tarda **uno** en `add`/`sub`/`and`/`xor` y **cuatro** en la multiplicación: una property las cubre a las dos
- `##n` es un retardo exacto, `##[a:b]` una ventana, `##[1:$]` *"en algún momento"* — que casi nunca es lo que se quiere: una property sin cota superior no puede fallar nunca por timeout
- El antecedente **excluye** `no_op` a propósito: es la única operación que no contesta, y meterla adentro convertiría la property en un generador de falsos positivos

Assertions (SVA)

`disable iff` y el reset

```
default disable iff (!reset_n);   // una vez, para toda la interface

property p;
   @(posedge clk) disable iff (!reset_n)   // o property por property
   start |=> done;
endproperty
```

- Durante un reset **todo está mal a propósito**: `done` se cae, `start` se cae, las transacciones a mitad de camino se abandonan. Sin `disable iff`, cada reset es una tanda de falsos positivos
- `disable iff` **aborta** las evaluaciones en vuelo, no las hace fallar. La property se olvida de lo que estaba esperando y arranca de cero
- El `default disable iff` al principio de la interface lo aplica a todas: es una línea contra veinte repeticiones

Assertions (SVA)

Dónde viven: adentro de la interface

- Van **con las señales**, en la `interface`, no en el testbench de clases. Ahí ven `clk`, `start`, `A`, `B` y `done` sin que nadie se las pase
- No hay que conectarlas, ni instanciarlas, ni construirlas en un `build_phase`. Existen porque la interface existe
- El `top` de los agents instancia **dos** BFM: las mismas properties chequean las dos VTALU, incluida la que maneja el módulo heredado que nadie escribió
- Un **agent pasivo** se las lleva de regalo: mirar una interface ahora también es chequearla

Assertions (SVA)

bind: cuando la interface no es tuya

```
// El checker vive afuera. El RTL no se toca — muchas veces no se puede.
module apb_checks (input bit PCLK, PSEL, PENABLE, PREADY);
   a_setup : assert property (@(posedge PCLK) PSEL && !PENABLE |=> PENABLE);
   c_setup : cover  property (@(posedge PCLK) PSEL && !PENABLE);
endmodule

// Y en el top, una línea por cada módulo que se quiera vigilar:
bind apb_regs apb_checks chk (.*);
```

- La slide anterior vale cuando la interface **es tuya**. El RTL ajeno, el IP comprado y el módulo heredado no se editan
- `bind` mete una instancia **adentro** de otro módulo desde afuera: el checker ve las señales internas del DUT como si estuviera escrito ahí
- El `.*` conecta por nombre. Un `bind` alcanza para **todas** las instancias de ese módulo — o `bind top.dut ...` para una sola
- También se puede bindear a una `interface`, y Verilator soporta las dos formas

Assertions (SVA)

La integración con UVM: el `else` que hace que cuente

```
// El default: $error, y en Verilator ademas $stop — se corta en la primera falla
a : assert property (p);

// Lo que se quiere en UVM: la falla cuenta y la simulacion sigue
a : assert property (p)
    else `uvm_error("SVA", $sformatf("%m: operando cambiado"));
```

- Sin `else`, la acción por defecto de una assertion que falla es `$error` —y en varios simuladores, `$stop`. La simulación se corta y el *Report Summary* **no se imprime**
- Con un `else` que llama a `` `uvm_error ``, la falla entra por el mismo canal que todo lo demás: cuenta en el resumen, respeta `+UVM_MAX_QUIT_COUNT`, y el `run.sh` la detecta sin cambiar una línea
- El `%m` del mensaje imprime **qué instancia** de la interface falló: `top.clase_bfm` o `top.modulo_bfm`

Assertions (SVA)

Toda assertion va con su `cover property`

```
// And the cover that goes with it: without this, a regression where the random
// never subtracted too much leaves the assertion green having checked nothing.
c_ovf : cover property (@(posedge clk) done && ovf);
c_sub_sin_borrow : cover property (@(posedge clk) done && (op_set == sub_op) && !ovf);

c_mult_3ciclos : cover property (@(posedge clk) $rose(start) && (op_set == mul_op) ##3 done);
c_mult_4ciclos : cover property (@(posedge clk) $rose(start) && (op_set == mul_op) ##4 done);
c_un_ciclo : cover property (@(posedge clk) $rose(start) && (op_set inside {add_op, sub_op, and_op, xor_op}) ##1 done);
```

```
covergroup : 86.8% (66/76)
user       : 80.0% ( 4/ 5)
```

- Una property cuyo **antecedente no ocurre nunca** pasa. Sin cobertura de assertions, un chequeo apagado se ve igual que un chequeo verde
- `c_mult_3ciclos` se queda en **cero para siempre**: la cadena del multiplicador es `done3 → done2 → done1 → done_mult`, o sea **cuatro** flancos, no tres. El cover en 0 avisa de que el modelo mental de la latencia estaba mal
- En Verilator los `cover property` caen en el **mismo** `coverage.dat` que los covergroups del testbench convencional: cobertura de código, funcional y de assertions, un solo reporte

Assertions (SVA)

Assertion o scoreboard: la misma regla, dos lugares

	Assertion	**Scoreboard**
Qué chequea	el **protocolo**: quién se mueve, cuándo	la **transformación**: `A op B`
Dónde vive	en la `interface`, con los cables	en el testbench, con las transactions
Cuándo grita	en el flanco exacto en que se rompe	cuando llega el resultado
Qué necesita	nada: se enchufa sola	monitor, analysis port, `connect_phase`

- Regla corta: **protocolo → assertion. Datos → scoreboard.** Escribir un chequeo de protocolo en el scoreboard es reconstruir el tiempo que el monitor se acaba de encargar de borrar
- Y al revés: escribir el modelo de referencia de un multiplicador en SVA es posible y es una mala idea

Assertions (SVA)

El ejemplo de la sección: el bug que el scoreboard no ve

```
        // +BUG=1: changes B halfway through the multiplication. The
        // multiplier already latched A and B on the first edge, so the
        // RESULT DOES NOT CHANGE: the scoreboard stays green. The only
        // thing that sees it is the assertion. That is the whole section.
        if (bug_operandos && iop == mul_op) begin
           @(negedge clk);
           @(negedge clk);
           B = ~iB;
        end
        do @(negedge clk); while (done == 0);
        oresult   = result;
        start = 1'b0;
```

```
** Report counts by severity
UVM_INFO :    2
UVM_WARNING :    0
UVM_ERROR :  154
UVM_FATAL :    0
** Report counts by id
[MAXMULT]     1
[RNTST]     1
[SVA]   154
```

- `+BUG=1` cambia `B` **a mitad de la multiplicación**. El multiplicador ya latcheó los operandos en el primer flanco: **el resultado sale bien igual**
- El scoreboard compara 1000 operaciones y no encuentra una sola diferencia: en el resumen **no hay un solo** `[SELF CHECKER]`. Está haciendo bien su trabajo — este bug no es de datos
- La assertion lo caza 154 veces, en el flanco exacto. Es el mismo ejemplo de reporting, dado vuelta: allá se rompía el scoreboard para enseñar reporting; acá se rompe el protocolo para mostrar quién lo ve

Assertions (SVA)

Las trampas de esta sección

El síntoma	La causa	Cómo se ataja
La property **pasa siempre**	el antecedente no ocurre nunca	un `cover property` por assertion
Falsos positivos en cada transacción	la muestreás con el flanco en que se escribe	estímulo y respuesta van en flancos distintos
Falsos positivos **al arrancar** cada test	falta el `disable iff (!reset)`	`default disable iff` una vez, arriba
El log dice **PASS** y las properties no corrieron	falta `--assert` al compilar	el `cover` en 0 es el que avisa

- Las cuatro comparten la forma del apéndice: **compilan, corren, y mienten**
- La cuarta es la más barata de cometer y la más cara: sin `--assert`, Verilator compila las properties y no las evalúa. Todo verde, cero chequeo

Assertions (SVA)

Resumen de la unidad

- Una **assertion concurrente** es una declaración con reloj: se escribe una vez y chequea cada flanco de la simulación, sola
- La anatomía no cambia nunca: `label : assert property (@(reloj) disable iff (reset) antecedente |-> consecuente) else acción;`
- `|->` es el mismo flanco y `|=>` **es** `|-> ##1`. La pregunta no es cuál de los dos, es cuántos flancos después lo promete la spec
- Las properties viven **en la** `interface`, con las señales. No se conectan, no se construyen, y viajan con ella
- En UVM la acción es `` `uvm_error ``, para que la falla **cuente** en el *Report Summary* en vez de matar la simulación
- **Toda assertion va con su** `cover property`: es el único chequeo del chequeo
- Y la que se lleva la sección: **una assertion vale lo que vale su muestreo**. El estímulo se muestrea donde el estímulo se escribe

Assertions (SVA)

Lo que esta sección hace distinto

- **El *UVM Primer* no tiene SVA.** Enseña el testbench de clases de punta a punta y deja las assertions afuera: son de la otra mitad de SystemVerilog
- Esta sección existe porque la pregunta llega igual —en la entrevista, y en el primer bloque real que a uno le toca verificar
- Los dos relojes, la tabla de los 145 falsos positivos y el `cover property` que nunca se cubre **no salieron de un tutorial**: salieron de escribir esta sección sobre el VTALU del curso y mirar por qué no daba
- Todo lo de acá corre en Verilator, sin licencias, con el mismo `run.sh` de los otros treinta y siete ejemplos

Ejercicio · Día 7 · 2 de 2

El módulo heredado viola el protocolo y nadie lo sabía

```
$ cd code/ejercicios/d7-sva && bash run.sh
```

- El "tester del jefe" de los agents maneja su VTALU **a mano** para la multiplicación, y va dejando listo el operando de la próxima mientras espera `done`. Está en producción hace años
- El scoreboard compara mil operaciones y no encuentra una sola diferencia: el multiplicador latcheó `A` y `B` en el primer flanco. **El bug no es de datos**
- Escribí la property que sí lo ve. Las dos de `done` ya están, de molde
- El corrector pide tres cosas: que dispare sobre `modulo_bfm`, que **no** dispare ni una vez sobre `clase_bfm`, y que el scoreboard siga en verde

Repaso · Día 7

1 de 5 · Inmediatas y concurrentes

¿Cuál es la diferencia de fondo entre `assert(x.randomize())` y `assert property (@(posedge clk) …)`?

- **a** Una es una **sentencia**; la otra, una **declaración con reloj** ✓
- **b** Ninguna: la segunda es azúcar sintáctico de la primera
- **c** La primera se puede apagar por línea de comandos y la segunda no
- **d** La primera sólo vale adentro de una clase y la segunda sólo adentro de un módulo, por el scheduler

Sentencia contra declaración — la inmediata corre cuando el hilo pasa por ahí; la concurrente se evalúa sola, en cada flanco de su reloj. Es a un `if` lo que la concurrente es a un `always_ff`: una se ejecuta, la otra se instancia. Por eso sólo la concurrente puede describir algo que dura varios ciclos.

Repaso · Día 7

2 de 5 · `|->` contra `|=>`

El `done` del VTALU sale de un `always_ff`. ¿Qué implicación va en `start |?? done`?

- **a** `|->`, porque el antecedente y el consecuente son de la misma transacción
- **b** `|=>`, porque lo que se escribe con `<=` se lee un flanco después ✓
- **c** Cualquiera de las dos: la diferencia es de estilo
- **d** Ninguna: para señales registradas hay que usar `$past()` sobre el antecedente

`|=>` cuando el consecuente sale de un `<=` — porque `|=>` *es* `|->` `##1`, y lo que hay que preguntarse es cuántos flancos después lo promete la spec. Con `|->` contra una señal registrada la property no pasa en vacío: **falla en cada transacción**, porque compara contra el `done` viejo. La que sí pasa callada es aquella cuyo antecedente nunca ocurre — por eso va siempre con su `cover property`.

Repaso · Día 7

3 de 5 · El flanco de muestreo

Todas las properties del VTALU muestreadas en `@(posedge clk)` dan 145 errores sobre 1000 operaciones, y el DUT está sano. ¿Por qué?

a. Falta el `disable iff (!reset_n)`

b. El `posedge` es demasiado rápido: hay que dividir el reloj

c. Los covergroups y las assertions no pueden compartir el reloj sin un `clocking block`

d. El estímulo se escribe en el `negedge` y el muestreo no lo ve ✓

Una assertion vale lo que vale su muestreo — la BFM escribe el estímulo en el `negedge`, y en dos `no_op` seguidas `start` baja y vuelve a subir entre dos `posedge`: el muestreo no lo ve bajar. El estímulo se muestrea donde el estímulo se escribe. Estímulo en `negedge`, respuesta del DUT en `posedge`: cero errores. Con un solo reloj no hay forma.

Repaso · Día 7

4 de 5 · La assertion que no chequea nada

Una property `assert` reporta 0 fallas durante toda la regresión. ¿Qué se sabe?

- **a** Nada todavía: puede que nunca se haya evaluado ✓
- **b** Que la regla que describe se cumple
- **c** Que el DUT está libre de bugs de protocolo
- **d** Que la property tiene un `disable iff` mal escrito y quedó apagada todo el tiempo

Cero fallas y cero evaluaciones se ven igual — puede que su antecedente no haya ocurrido nunca, o que falte `--assert` y ni siquiera se esté evaluando. Por eso toda assertion va con su `cover property` : es el único chequeo del chequeo. En la sección, `c_mult_3ciclos` se queda en 0 y delata que la latencia real son cuatro flancos, no tres.

Repaso · Día 7

5 de 5 · Assertion o scoreboard

El DUT devuelve el `result` correcto pero baja `done` un ciclo antes de lo que dice la especificación. ¿Quién lo caza?

a. El scoreboard, cuando compare el resultado

b. La cobertura funcional, porque el bin de `done` queda vacío

c. Una assertion en la interface: es un bug de **protocolo** ✓

d. El `uvm_fatal` del `command_monitor`, que dejaría de ver comandos en el bus

Protocolo → assertion. Datos → scoreboard — y no es una preferencia: para cuando la transaction llega al scoreboard, el protocolo ya no está. Escribir el chequeo ahí sería reconstruir a mano el tiempo que el monitor acaba de borrar.

Capstone · Día 7

El testbench entero, desde una hoja en blanco

```
$ cd code/ejercicios/d7-final && cat spec.md
```

- Los ejercicios anteriores daban un archivo con un agujero. Éste da un **DUT, una spec y nada más**
- El DUT no es la VTALU: es un **esclavo APB3** de cuatro registros. Tiene direcciones, dos fases por transferencia, un *wait state* y respuesta de error
- Se escribe todo: la interface con el protocolo, la transaction, el driver, el monitor, el agent, el scoreboard, el covergroup, las sequences y los tests
- El corrector va **por etapas**, y cada una imprime su `STAGE N OK`. Se puede terminar de a una — y conviene

Capstone · Día 7

El DUT: cuatro registros y la letra chica

Dir	Nombre	Acceso	Contenido
0x00	CTRL	RW	bit 0 = EN · bit 1 = CLR, **autoclear**
0x04	SCRATCH	RW	32 bits, y **suma a ACC si EN**
0x08	ACC	RO	el acumulador
0x0C	STATUS	RO	bit 0 = EN · bit 1 = OVF, pegajoso

```
// The handshake: the write does not wait, the read inserts ONE wait state.
// rd_wait re-arms as soon as PENABLE drops, that is between one transfer and
// the next, with no need to count cycles.
logic rd_wait;
always_ff @(posedge PCLK or negedge PRESETn) begin
   if (!PRESETn) rd_wait <= 1'b1;
   else if (!PENABLE) rd_wait <= 1'b1;
   else if (access && !PWRITE) rd_wait <= 1'b0;
end

assign PREADY  = !PSEL ? 1'b1 : (PWRITE ? 1'b1 : !rd_wait);
assign PSLVERR = access && !mapped;
```

- La lectura mete **un wait state**: el driver espera el handshake, no cuenta ciclos
- De `0x10` para arriba, `PSLVERR`. Escribir un RO, en cambio, **no** es error

Capstone · Día 7

El orden: monitor, driver, scoreboard, cobertura

- **1 · El monitor.** El `top.sv` trae **dos** esclavos: uno para tu testbench y otro que maneja un módulo de siempre, sin UVM. Un agent **pasivo** sobre ése, y a ver las ocho transferencias. Sin driver
- **2 · El driver.** El protocolo adentro de la interface, y una sequence dirigida que escriba y lea los cuatro registros
- **3 · El scoreboard.** El DUT modelado en software. El corrector lo corre dos veces: contra el DUT sano tiene que callarse, y con `+BUG=1` tiene que gritar
- **4 · La cobertura.** El covergroup con las nueve filas del plan de verificación de la spec: al menos 20 puntos, 90 % cubierto. Y **dos filas vienen vacías**
- **5 · Las properties.** El protocolo del APB adentro de `apb_if.sv`, con `--assert`. Con `+BUG=2` hay un bug que el scoreboard no puede ver

Capstone · Día 7

La etapa 5: el bug que el scoreboard no puede ver

```
// En apb_if.sv, con las señales. El payload no se mueve hasta que termina
property p_payload_estable;
   @(posedge PCLK) PSEL && !(PENABLE && PREADY) |=> $stable({PADDR, PWRITE, PWDATA});
endproperty

a_payload_estable : assert property (p_payload_estable)
   else `uvm_error("SVA", $sformatf("%m: el payload se movió antes del PREADY"))
```

- `+BUG=2` le hace al módulo de siempre lo que `d7-sva` le hacía al tester heredado: **mover `PADDR` en el medio de ACCESS**
- El DUT contesta en la dirección nueva, el monitor reconstruye **ocho transferencias impecables**, y el scoreboard no tiene nada que comparar mal
- Medido: **8 `UVM_ERROR [SVA]`** y las mismas 8 transferencias en el log del monitor. La única que se entera es la assertion
- Y las filas **8 y 9** del plan vienen vacías: *back to back* y dirección no alineada. Los bins se llaman `back_to_back` y `unaligned`

Apéndice · De la VTALU a un bus real

Lo que no cambia

- El DUT del curso tiene un handshake de tres señales. El lunes te vas a encontrar con **AXI, APB o AHB**, y la pregunta honesta es cuánto de esto sirve
- El árbol, entero: `test` → `env` → `agent` → sequencer, driver y monitores, con el análisis colgado de los analysis ports. **Es el mismo diagrama**
- El agent sigue siendo **uno por interface**. Un DUT con un APB de configuración y un AXI de datos son dos agents, no un agent más grande
- La configuración sigue bajando por **un objeto por nivel**, y el ámbito del `config_db` sigue siendo una ruta
- La sequence sigue siendo un objeto que se crea, corre y se tira

Apéndice · De la VTALU a un bus real

Lo que sí cambia

- **El driver deja de ser una task lineal.** Una operación de la VTALU es subir `start` y esperar `done`. Un bus tiene fases —dirección, dato, respuesta— y en AXI los cinco canales corren **independientes**: el driver pasa de un `send_op()` a varios threads con `fork`
- **El monitor tiene que reconstruir.** Ya no alcanza con mirar un flanco: hay que aparear la dirección de un canal con el dato de otro, por ID. Es la clase más difícil del agent, y la que más se subestima
- **La transaction crece**: dirección, tipo de burst, largo, strobes, respuesta. Y las constraints dejan de ser decorativas — en AXI un burst no puede cruzar un límite de 4 KB, y eso es una `constraint`, no un comentario
- **Las sequences virtuales dejan de ser un lujo**: configurá por APB, después mandá tráfico por AXI con lo que la configuración devolvió. La forma es la de las sequences — `virtual_sequencer`, `p_sequencer`, un `fork` por interfaz—; lo único que cambia es que los dos sequencers son de tipos distintos
- **Aparece RAL.** Si el DUT tiene registros —y los tiene—, `uvm_reg` te da el modelo y los tests de registro hechos

Apéndice · De la VTALU a un bus real

Reuso vertical: el env del bloque adentro del env del sistema

```
class soc_env extends uvm_env;
   apb_env apb_env_h;     // el env del bloque, sin tocarle una línea
   axi_env axi_env_h;
   soc_scoreboard sb_h;   // el de sistema mira las dos puntas, no las transferencias

   function void build_phase(uvm_phase phase);
      // Lo que cambia no es el código del env: es su CONFIGURACIÓN
      apb_cfg.is_active = UVM_PASSIVE;   // acá el APB lo maneja el CPU del SoC
      uvm_config_db #(apb_env_config)::set(this, "apb_env_h*", "cfg", apb_cfg);
      apb_env_h = apb_env::type_id::create("apb_env_h", this);
   endfunction
endclass
```

- El reuso de la sección de agents es **horizontal**: dos instancias del mismo agent en el mismo env. Éste es **vertical**, y es el que se cobra en un SoC
- El env del bloque no se edita. Lo que cambia es su config: el agent pasa a **pasivo** porque a nivel de sistema el bus lo maneja el DUT, no el testbench
- Por eso `is_active` y el objeto de config no son ceremonia: son la bisagra que hace que el mismo código sirva en los dos niveles
- Lo que **no** sube es el scoreboard de bloque: el de sistema compara entrada contra salida del SoC, y el de bloque sigue midiendo su bus

Apéndice · De la VTALU a un bus real

Por dónde empezar

- **Buscá un VIP antes de escribir un agent.** Casi nadie escribe un agent de AXI desde cero: los vendors los venden y hay libres. Escribir el tuyo es un proyecto de meses y no es el que te pidieron
- **El protocolo antes que la metodología.** El 90 % de los bugs de un agent nuevo son de protocolo mal leído, no de UVM. La spec del bus primero
- **Lo primero que se escribe es el monitor**, no el driver. Si no podés *ver* el bus no podés verificar nada — ni siquiera el estímulo de otro
- Y ahí el **agent pasivo** es tu primer entregable de verdad: mirar antes de manejar
- **SVA en paralelo.** Un bus tiene reglas que se chequean donde ocurren, no en el scoreboard, y en un bus con transacciones solapadas ésa es la diferencia entre media hora y dos días. La herramienta ya la tenés: **las assertions**, con la ventaja de que las properties viajan adentro de la interface del VIP

Apéndice · La caja de herramientas de debug

Las herramientas que vimos, todas juntas

Herramienta	Para qué	Dónde salió
`VLT_TRACE=1` + GTKWave	ver las señales de verdad	La spec del VTALU
`+UVM_VERBOSITY=UVM_HIGH`	ver los mensajes de los monitores	Reporting
`+UVM_CONFIG_DB_TRACE`	quién puso qué en el `config_db`, y quién lo leyó	Agents
`+TOPOLOGY` → `print_topology()`	el árbol que UVM armó **de verdad**	Agents
`+UVM_OBJECTION_TRACE`	quién levantó y quién bajó la objection	Tests
`+UVM_TIMEOUT=N,NO`	un techo para el cuelgue, en vez de esperar	Tests
`+UVM_MAX_QUIT_COUNT=N`	matar la corrida al N-ésimo error	Reporting

- Los seis de abajo son **plusargs**: no se toca código y no se recompila. En un testbench con UVM eso son minutos de diferencia por intento

Apéndice · La caja de herramientas de debug

Cuál usar, según el síntoma

El síntoma	El primer sospechoso	Con qué se mira
Termina en **t=0** y dice PASS	nadie levantó la objection	`+UVM_OBJECTION_TRACE`
No termina nunca	un `item_done()` que no se llamó, o una objection que no baja	`+UVM_TIMEOUT=5000000,NO` y después el trace
El scoreboard **grita en todas**	el monitor muestrea mal — el DUT casi nunca es	`+UVM_VERBOSITY=UVM_HIGH`
El `config_db` **no encuentra**	el ámbito del `set()`, no el `get()`	`+UVM_CONFIG_DB_TRACE`
La cobertura da **0 %**	falta el `new()` o el `sample()` del covergroup	leer el `.dat` con `verilator_coverage`

El síntoma	El primer sospechoso	Con qué se mira
El árbol **no es el que dibujaste**	un `create()` sin factory, o un override tardío	`+TOPOLOGY`

- La regla de oro está en la segunda columna: **el sospechoso número uno nunca es el DUT.** Es el testbench que lo mira

Apéndice · Las 21 trampas mudas

Compila, corre, y miente

El síntoma	La causa	Cómo se ataja	Sección
La cobertura da **0 %**	falta el `new()` o el `sample()` del covergroup	un `sample()` por transacción, en el subscriber	Cobertura funcional
La hija corre el método de **la madre**	sin `virtual`, SV resuelve por el **tipo de la variable**, en compilación	`virtual` en todo método que alguien vaya a extender	Polimorfismo
El component lee **la config de otro**	`get(null, "*", …)` devuelve lo primero que matchee	`get(this, "", …)`, y `+UVM_CONFIG_DB_TRACE`	Tests
El test pasa con **el estímulo equivocado**	el `set_type_override()` llegó **después** del `create()` del env	el override primero, siempre. `+TOPOLOGY` lo delata	El env
El modelo de registros da	el acceso arranca con `WO`: `bit_bash` lo saltea y	si el campo se lee, `WC` o `W1C`. `WO*` sólo si de verdad no se	RAL · d8

El síntoma	La causa	Cómo se ataja	Sección
verde y el campo nunca se testeó	do_check lo saca de la máscara	lee	

- Ninguna de estas cinco da un warning. Ninguna deja la regresión en rojo. Todas se descubren semanas después, o no se descubren

Apéndice · Las 21 trampas mudas

El que publica y el que escucha

El síntoma	La causa	Cómo se ataja	Sección
El subscriber cuenta **0** y el monitor imprime 1000	falta el `connect()` en el `connect_phase`	corregir cruzado: contra lo que dice el que ya andaba	Analysis ports · d4
El monitor **no publica nunca**, y grita el scoreboard	falta `bfm.command_monitor_h = this` en el `build_phase`	el `uvm_fatal` aparece en la clase que **no** tiene la culpa	Analysis ports
La copia **pierde los campos de la madre**	un `do_copy()` que no llama a `super.do_copy(rhs)`	todos los `do_copy()` llaman primero al de arriba	Jerarquías
Dos transactions distintas **comparan iguales**	`super.do_compare()` llamado en una línea suelta y descartado	encadenado con &&, nunca suelto	Transactions

- Las cuatro tienen la misma forma: **el que falla no es el que se equivocó.** Por eso el primer reflejo —abrir el archivo que gritó— es el reflejo equivocado

Apéndice · Las 21 trampas mudas

Estímulo, ámbito y objections

El síntoma	La causa	Cómo se ataja	Sección
Los bins de borde **no se llenan nunca**	`dist` con `:=`: el peso va a **cada** valor del rango	medir el histograma **una vez**, con `:/`	Constrained random
El `randomize()` **no corre** con las asserts apagadas	`assert(x.randomize())` — `assert` es una directiva de simulación	`if (!x.randomize()) uvm_fatal(…)`	Constrained random
Bajo el override, **algunas** transactions son del tipo viejo	un `new()` donde iba `type_id::create()`	por la factory pasa lo que se crea con `create()`	Transactions

El síntoma	La causa	Cómo se ataja	Sección
Los **dos agents** arrancan iguales	dos `set()` con ámbito `"*"`: el segundo pisa al primero	el ámbito es la **ruta** del que lee, con `*` al final	Agents · d6
El agent arranca **activo** aunque pusiste `is_active` en el `config_db`	falta el `super.build_phase()`: el que lo lee es `uvm_agent`	o config object, o `super` — nunca media de cada una	Agents
La simulación **no termina nunca**	un `item_done()` que no se llamó	`+UVM_TIMEOUT=5000000,NO` primero, el trace después	Agents
Termina en **t=0** y dice PASS	nadie levantó la objection alrededor de la sequence	`+UVM_OBJECTION_TRACE`	Sequences

Apéndice · Las 21 trampas mudas

Las cuatro de las assertions

El síntoma	La causa	Cómo se ataja	Sección
El log dice **PASS** y las properties no corrieron	falta `--assert` al compilar	el `cover property` en 0 es el único que avisa	Assertions
La property **pasa siempre**	el antecedente nunca ocurre, o la implicación es la que no va	un `cover property` por cada `assert property`	Assertions
145 falsos positivos y el DUT está sano	la muestreás con el flanco en el que se escribe	estímulo en negedge, respuesta del DUT en posedge	Assertions · d7
Falsos positivos **al arrancar** cada test	falta el `disable iff (!reset_n)`	`default disable iff`, una vez, arriba de todo	Assertions

- Las cuatro son de la misma familia que las de arriba, con un agravante: una assertion rota **se ve exactamente igual** que una que anda. No hay salida que mirar

Apéndice · Las 21 trampas mudas

Y la que trae el clocking block

El síntoma	La causa	Cómo se ataja	Sección
El scoreboard falla **una vez cada veinte** y el waveform se ve bien	media señal se lee por `cb.sig` y la otra media por `sig`	si entró al clocking block, **todo** el protocolo la lee por ahí	Interfaces y BFM

- Es la única de la lista que **la agrega una herramienta**: sin clocking block no existe. Mal usado es peor que no usarlo
- Dos nombres para el mismo cable, y difieren en un ciclo: `code/u2/clocking/mezcla.sv` imprime `3` y `4` en el mismo instante
- La variante del mismo error: esperar `@(posedge clk)` y después leer `cb.sig`. Si adoptás el clocking block, adoptás **también su evento**

Glosario · ES → EN

Lo que vas a tener que buscar en inglés

Lo que dijimos acá	Cómo se busca afuera
cobertura funcional / de código	*functional* / *code coverage*
plan de verificación	*verification plan*, *coverage plan*
cierre de cobertura	*coverage closure*
casos borde	*corner cases*
estímulo dirigido / al azar	*directed* / *constrained-random stimulus*
ondas	*waveforms*
semilla	*seed*
regresión	*regression*
primer silicio · rehacer las máscaras	*first silicon* · *respin*
aserción	*assertion* — y `assert property` no se traduce
muestreo	*sampling* — y el flanco en que ocurre, *sampling edge*

Lo que dijimos acá	Cómo se busca afuera
banco de pruebas	**testbench**, siempre — nadie dice otra cosa

- El curso está en español, pero el LRM, el *User Guide* y cada respuesta que vayas a encontrar están en inglés. Esta tabla es el puente

Glosario · Las piezas

Una línea cada una, para tenerlas juntas

Palabra	Qué es
handle	el nombre de un objeto — **no** es el objeto
factory	el diccionario de tipos: pedís la clase base y te dan la derivada
override	cambiar el tipo que la factory entrega, sin tocar quien lo pide
transaction	un dato que viaja por el testbench: un `uvm_sequence_item`
agent	todo lo que sabe hablar una interface, en una caja
sequence / *sequencer*	el estímulo (objeto) y el árbitro que lo entrega (componente)
virtual sequence	la que no manda items propios: coordina **varios** sequencers
driver	convierte una transaction en señales
monitor	mira el bus y publica transactions
scoreboard	compara lo que salió contra lo que debía salir
subscriber	cualquiera que escucha un analysis port

Palabra	Qué es
objection	el *"todavía tengo trabajo"* que mantiene viva la fase
assertion	una regla del protocolo escrita para que el simulador la chequee
property	la regla en sí: reloj, antecedente, implicación, consecuente
antecedent / *consequent*	el *"si"* y el *"entonces"* de una property
cover property	cuenta **cuántas veces se cumplió** una property. El chequeo del chequeo

Glosario · Tres palabras que engañan

- `virtual` — en SystemVerilog son **tres cosas sin relación**: un *virtual method* (se resuelve por el objeto), una *virtual class* (abstracta, no se instancia) y una *virtual interface* (un handle a una interface). Que compartan la palabra es un accidente del lenguaje
- `assert` — en SystemVerilog son **dos cosas**: la *inmediata*, una sentencia que corre donde el hilo pasa, y la *concurrente* (`assert property`), una declaración con reloj que se evalúa sola toda la simulación — las assertions. El `assert(randomize())` que se ve por todos lados es la primera, y es una forma pobre de chequear un valor de retorno: por eso el curso usa `` `uvm_fatal ``
- ***coverage*** — sin apellido no quiere decir nada. Si alguien te dice "estamos en 90 %", la pregunta es siempre *¿de código o funcional?*
- Y una de escala: un **test** en UVM es una **clase**, no un archivo ni una corrida. La corrida es un *run*; el conjunto de corridas, una *regression*

Referencias

- **IEEE 1800-2017** — SystemVerilog Language Reference Manual. El árbitro final de toda discusión de sintaxis.
- **IEEE 1800.2** / **Accellera UVM** — la metodología, y el *UVM User Guide*. El código de `uvm-core` lo tenés en disco: `make uvm`.
- **Verification Academy** — Siemens EDA · verificationacademy.com
- **Salemi, Ray.** *The UVM Primer* — Boston Light Press, 2013. El libro con el que aprendí UVM, y del que salieron varios de los ejemplos de `code/`.
- **2024 Siemens EDA / Wilson Research Group Functional Verification Study**. Fuente de los **datos** de Tendencias. Los gráficos son propios: se generan con `make figs` desde `res/trends/data.json`.
- **Sobre RAL**: el capítulo *Using the Register Layer Classes* del **UVM 1.2 User's Guide** de Accellera es la explicación larga, y la tabla de los veinticinco accesos de campo —`RW`, `WOC`, `W1C`…— está en el **IEEE 1800.2**, en `uvm_reg_field`. Lo que la unidad no usa vive en el mismo lugar: `add_hdl_path` para el backdoor, y **IP-XACT** (IEEE 1685) o **SystemRDL** (Accellera) para generar el modelo en vez de escribirlo.
- **Sobre los clocking blocks**, que es la discusión más viva del curso: los hilos de **Dave Rich** en Verification Academy (2014, 2022 y 2024), su paper *The Missing Link: The Testbench to DUT Connection*, y el **DV Coding Style Guide de lowRISC/OpenTitan**, que los prescribe como regla. El resumen, con los links y

Si querés seguir leyendo

EL LIBRO CON EL QUE APRENDÍ

The UVM Primer

Ray Salemi · Boston Light Press, 2013 · ISBN 978-0974164939

Este curso no es ese libro: tiene otra estructura, otro DUT, otro simulador y cuatro unidades de material que el libro no cubre. Lo que sí comparte está acreditado en `NOTICE`, y la lista completa de diferencias está en `docs/en-que-se-diferencia.md`.

Créditos

- Parte de los ejemplos de `code/` deriva de los del **UVM Primer**, publicados por su autor bajo **Apache-2.0**. Acá siguen bajo Apache-2.0, con el `NOTICE` que la licencia pide y la lista de cambios.
- El curso — `slides/` y `docs/` — es **CC BY 4.0**. Las herramientas de `tools/`, **MIT**.
- Los gráficos de tendencias son propios, hechos a partir de los porcentajes publicados del estudio de Wilson Research Group / Siemens EDA.
- **reveal.js** y **forkit.js** de Hakim El Hattab (MIT); las tipografías Chakra Petch e IBM Plex bajo SIL OFL 1.1.

Hasta acá llegamos

Qué te llevás

- Sabés **por qué** existe la verificación como disciplina, y qué dicen los números de la industria
- Escribiste un testbench convencional, lo medís con **cobertura funcional**, y sabés leer el número que sale
- Podés leer y escribir **OOP en SystemVerilog**: clases, herencia, polimorfismo, clases paramétricas, la factory
- Armaste un testbench **UVM completo** —test, env, componentes, monitores, driver, scoreboard, transactions, agents y sequences— pieza por pieza, entendiendo qué reemplaza a qué
- Escribiste **assertions**: la otra mitad del plan, la que ningún scoreboard puede cerrar porque no es sobre el resultado sino sobre el protocolo
- Y lo entregaste: el **capstone** es un DUT que no viste antes, su spec, y el testbench entero desde una hoja en blanco — que es literalmente lo que se pide en el trabajo
- Y todo eso corre en tu máquina, con **herramientas libres**, sin pedirle licencia a nadie

Lo que quedó afuera

y dónde seguir

Tema	Por qué te lo vas a cruzar
`uvm_event`, `uvm_barrier`	La sincronización que queda cuando la objection no alcanza
`uvm_heartbeat`	El que avisa que un componente dejó de dar señales de vida en una regresión de horas
`uvm_sequence_library`	Una bolsa de sequences que el sequencer elige al azar: el test de estrés sin escribirlo
`uvm_pool`, `uvm_queue`	Las colecciones de UVM. Un array asociativo alcanza hasta que hay que compartirlo
TLM2, phase jumps	El límite del curso, y es explícito
UPF, gate-level, PSS	Tres mundos aparte: bajo consumo, la netlist con retardos, y generar los tests desde un modelo

- Los tres lugares donde seguir: el **LRM IEEE 1800-2017**, el *UVM User Guide* de **Accellera**, y **Verification Academy**
- Y uno que casi nadie usa: el **código de `uvm-core`**. Está en tu disco, en `code/.uvm/src/`, y a esta altura lo podés leer
- Y si te trabás: **Discussions**, con una categoría por día. Un ejemplo que no corre o una explicación que no se entiende son **issues del curso**, no problemas tuyos
- Y si querés seguir acá mismo: el **día 8** es opcional y arranca después de la autoevaluación y de *El lunes* — RAL, el modelo de referencia en C, y un segundo capstone

Autoevaluación

Quince cosas que a esta altura tendrías que poder hacer · 1 de 3

¿Puedo…?	Dónde está
1 · Explicar por qué el **47 %** del tiempo de un verificador se va en debug, y qué hace este curso al respecto	Tendencias
2 · Escribir un **plan de verificación** de cinco columnas para un DUT que no vi antes	El plan de verificación
3 · Decir por qué **100 % de cobertura de código** no quiere decir que el DUT esté verificado	Cobertura funcional
4 · Decir qué mirar **primero** cuando un `covergroup` reporta 0 %	Cobertura funcional
5 · Explicar qué race evita un `clocking block` — y por qué igual son **opcionales**	Interfaces y BFM

Autoevaluación

Quince cosas que a esta altura tendrías que poder hacer · 2 de 3

¿Puedo…?	Dónde está
6 · Extender una clase y cambiarle el comportamiento **sin copiarla**, y decir qué hace `super`	Clases y extensiones
7 · Cambiar la clase que usa un testbench **sin tocar el env**, con un `set_type_override`	El patrón factory
8 · Explicar por qué una simulación de UVM **termina en t=0** si nadie levanta un objection	Tests
9 · Decir por qué `build_phase` corre de arriba hacia abajo y `connect_phase` al revés	Components y fases
10 · Hacer que un scoreboard que falla diga **algo más** que "falló"	Reporting

Autoevaluación

Quince cosas que a esta altura tendrías que poder hacer · 3 de 3

¿Puedo…?	Dónde está
11 · Colgar un subscriber nuevo de un analysis port **sin tocar al que publica**	Un solo lugar que mira el cable
12 · Escribir una constraint que llegue a un caso de borde, y **medir** si el `dist` hace lo que dice	Constrained random
13 · Decir qué construye y qué **no** un agent con `is_active = UVM_PASSIVE`	Agents
14 · Explicar por qué una `uvm_sequence` **no aparece** en `print_topology()`	Sequences
15 · Escribir una property que vea una violación de protocolo, y decir por qué `--assert` no es opcional	Assertions (SVA)

- ¿Las quince? Entonces la que queda es el **capstone**: un esclavo APB, su spec, y el testbench entero desde una hoja en blanco

El lunes

Qué hacer con esto cuando se apaga la pantalla

- **Lo primero: un testbench propio andando.** El capstone es el mínimo. Si ya está entregado, el que sigue es un DUT tuyo — el del trabajo, el de la tesis, el que sea. Un curso terminado no prueba nada; un testbench sí
- **Guardá dos páginas y tirá el resto:** el machete de una carilla y *Las 21 trampas mudas*. Son las dos que se abren con el simulador corriendo
- **Si te trabás, preguntá acá:** Discussions tiene una categoría por día. Un ejemplo que no corre es un **issue del curso**
- **Y si algo te resultó confuso, arreglalo.** Es CC BY 4.0 y el repo está entero: las slides, los ejemplos, los correctores y los linters. Un PR que aclara una slide vale tanto como uno que agrega una sección

Gracias por llegar hasta acá.

Ayer quedó...

Dónde dejamos el testbench

- El capstone entregado: un esclavo APB, su spec, y un testbench entero escrito desde una hoja en blanco — monitor, driver, scoreboard, cobertura y properties
- El scoreboard de ese testbench **predice a mano** el valor de cada registro, y la cobertura del bus se escribió fila por fila
- Y eso se hace igual en el trabajo... hasta que el DUT tiene sesenta registros. Ahí las dos cosas se modelan, y eso es todo el día 8

Agenda

Día 8 · ≈ 4 h · opcional · lo que sigue después del capstone

- RAL, la unidad 9 — el modelo de registros sobre el mismo APB
- El modelo de referencia en C, por DPI
- El segundo capstone: una FIFO con backpressure

Al final del día podés:

- **Leer y escribir** un registro por nombre, sin escribir un solo `PADDR`, y decir qué hacen el adapter y el predictor
- **Enchufar** un modelo de referencia escrito en C, y probarlo con una mutación
- **Escribir** un scoreboard para un DUT donde una tabla no alcanza

RAL

Unidad 9 · opcional · el registro deja de ser una dirección

- Todo chip tiene registros de configuración, y un SoC chico tiene **doscientos**
- Sin RAL, el testbench los escribe así: `addr = 8'h04`, `wdata = ...`, y el scoreboard guarda un `logic [31:0]` por cada uno
- El día que el arquitecto mueve `SCRATCH` de `0x04` a `0x24`, hay que buscar `8'h04` en todo el testbench — y `0x04` también era el valor de otra cosa
- **RAL** —*Register Abstraction Layer*— es la capa que le pone nombre a eso: `model.SCRATCH.write(status, 'h1000_0000)`
- Es opcional y va al final a propósito: **se entiende recién cuando ya escribiste el testbench que reemplaza**

RAL

Tres piezas, y una sola sabe qué bus hay abajo

Pieza	Qué es	Cuántas veces se escribe
`uvm_reg_block`	el mapa: registros, campos, direcciones	una por **DUT**
`uvm_reg_adapter`	traduce acceso ↔ transacción del bus	una por **protocolo**
`uvm_reg_predictor`	lo que el monitor vio, dicho en registros	ninguna: viene hecha

- El modelo **no sabe** que abajo hay un APB. Cambiás el adapter y el mismo modelo maneja AHB
- El adapter es el archivo que te viene con un VIP comprado. Son veinte líneas
- Y abajo de todo, sin un solo cambio: el driver, el monitor y el agent del capstone

RAL

El modelo: la tabla de la spec, en código

```
// 0x00 -- CTRL. EN is an ordinary bit. CLR is not: writing a 1 clears ACC and
// OVF, and the bit always reads back 0.
class ctrl_reg extends uvm_reg;
   `uvm_object_utils(ctrl_reg)

   rand uvm_reg_field EN;
   rand uvm_reg_field CLR;

   function new(string name = "ctrl_reg");
      super.new(name, 32, UVM_NO_COVERAGE);
   endfunction : new

   // configure(parent, size, lsb_pos, access, volatile, reset, has_reset,
   //           is_rand, individually_accessible)
   virtual function void build();
      EN = uvm_reg_field::type_id::create("EN");
      EN.configure(this, 1, 0, "RW", 0, 'h0, 1, 1, 0);

      // "WC" is the name UVM has for exactly this: the write clears the field,
      // the mirror drops to zero, and the read returns that zero. Modelling it as
      // "RW" is the classic mistake, and it is what +MAL does. "WOC" also passes,
      // for the wrong reason: any WO* access takes the field out of bit_bash and
      // out of do_check, so nobody looks at it. See the RAL unit slides.
      CLR = uvm_reg_field::type_id::create("CLR");
      CLR.configure(this, 1, 1, $test$plusargs("MAL") ? "RW" : "WC", 0, 'h0, 1, 1, 0);
   endfunction : build

endclass : ctrl_reg
```

- Un campo se declara con `configure()`, y el argumento que importa es el **string de acceso**: `"RW"`, `"RO"`, `"WC"`...
- Ese string **es la especificación**. No es documentación: es lo que UVM usa para predecir, y para generar los tests de *Los tests que no escribiste*

RAL

El adapter: lo único que sabe que hay un APB

```
// Downstream: the model asks for a register access, this builds the bus item
// the sequencer will hand to the driver.
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
   apb_transaction t = apb_transaction::type_id::create("t");
   t.write = (rw.kind == UVM_WRITE);
   t.addr  = rw.addr[7:0];
   t.wdata = rw.data[31:0];
   return t;
endfunction : reg2bus

// Upstream: a bus item comes back -- from the driver's response, or from the
// monitor when prediction is explicit -- and this fills the generic operation
// the model understands.
//
// PSLVERR is what makes status meaningful: an access to an unmapped address
// comes back UVM_NOT_OK, and the write()/read() call reports it.
virtual function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
   apb_transaction t;
   if (!$cast(t, bus_item)) `uvm_fatal("REG ADAPTER", "Not an apb_transaction")
   rw.kind   = t.write ? UVM_WRITE : UVM_READ;
   rw.addr   = t.addr;
   rw.data   = t.write ? t.wdata : t.rdata;
   rw.status = t.slverr ? UVM_NOT_OK : UVM_IS_OK;
endfunction : bus2reg
```

- `reg2bus` baja: el modelo pide un acceso genérico, esto arma la transaction del bus
- `bus2reg` sube: la transaction vuelve —del driver, o **del monitor**— y esto la cuenta en términos de acceso
- `PSLVERR` se convierte en `UVM_NOT_OK`, y ahí el `status` de cada `write()` deja de ser decorativo

RAL

Engancharlo: tres líneas y un `connect_phase`

```
function void connect_phase(uvm_phase phase);
   // The three lines that turn a model into a testbench component: which
   // sequencer carries the accesses, which adapter translates them, and where
   // the truth about the DUT comes from.
   model.default_map.set_sequencer(agent_h.sequencer_h, adapter);
   model.default_map.set_auto_predict(0);

   predictor.map     = model.default_map;
   predictor.adapter = adapter;
   agent_h.ap.connect(predictor.bus_in);
endfunction : connect_phase
```

- `set_sequencer` es lo que hace que `model.CTRL.write()` termine en el driver: el modelo manda items por **el sequencer del agent que ya existía**
- `set_auto_predict(0)` apaga la predicción automática, y el `uvm_reg_predictor` la reemplaza escuchando el **analysis port del monitor**
- Y el `model.build()` del `build_phase`: un `uvm_reg_block` es un `uvm_object`, no un component. **Nadie te lo llama**

RAL

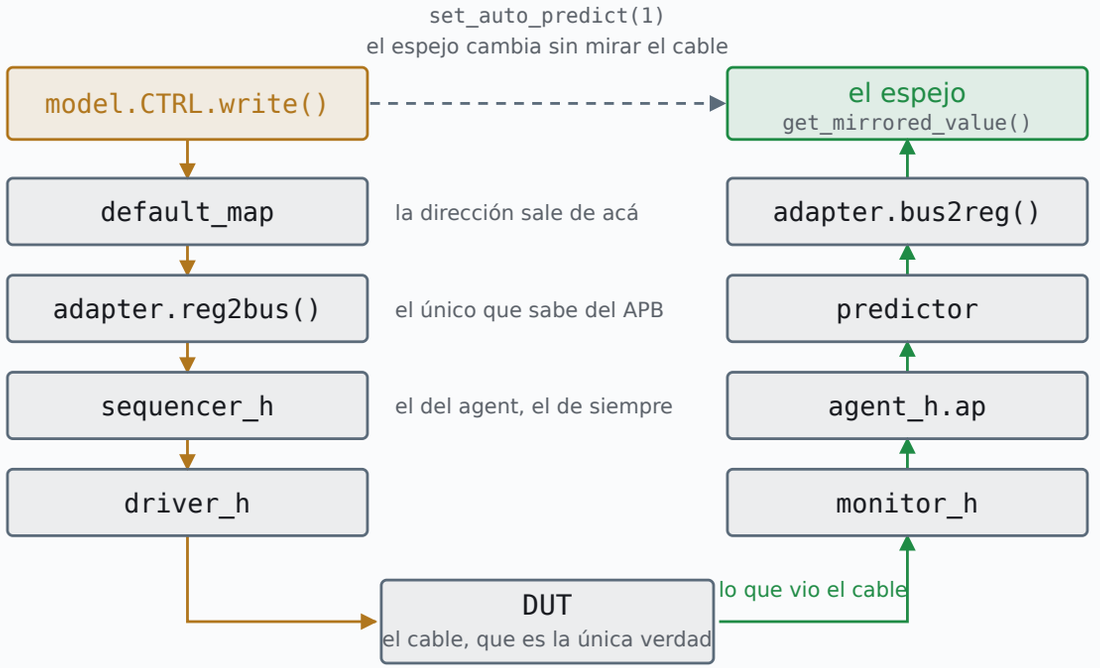
Predicción: quién le cuenta al modelo qué pasó

	Quién predice	Qué pasa si el DUT hace otra cosa
automática	la propia llamada `write()`	el modelo **no se entera**: cree lo que quiso mandar
explícita	el `uvm_reg_predictor`, desde el monitor	el modelo cree **lo que pasó en el cable**

- La automática es una línea menos y es la que sale en los tutoriales
- La explícita es la que hay que usar, y por la misma razón que el scoreboard del capstone mira el monitor y no el driver
- Y es gratis: el monitor y su analysis port **ya estaban**

RAL

El camino completo, de la llamada al espejo



Una sola pieza sabe que abajo hay un APB, y se escribe dos veces: `reg2bus` baja, `bus2reg` sube. Con predicción explícita el espejo cree **lo que pasó en el cable**, no lo que el modelo quiso mandar.

- La bajada se escribe; la subida —monitor, analysis port y `bus2reg`— **ya estaba**
- El `uvm_reg_predictor` no se escribe: se instancia y se le enchufa el mapa
- La punteada es `set_auto_predict(1)`: el atajo que no hay que tomar

RAL

El estímulo, sin un solo PADDR

```
    // --- the frontdoor -------------------------------------------------------
    // write() puts the value on the wire through the sequencer. There is no
    // second argument for the address: the address is in the map.
    model.CTRL.write(status, 'h1);              // EN = 1
    model.CTRL.read(status, data);
    `uvm_info("RAL", $sformatf("CTRL read back 0x%08h (status %s)", data, status.name()),
              UVM_MEDIUM)

    // --- prediction ----------------------------------------------------------
    // Nothing above updated the mirror by hand. The monitor saw the two
    // transfers, the predictor turned them into predictions, and the model
    // knows -- with set_auto_predict(0), which is the honest setting: what the
    // model believes comes from the BUS, not from what the test meant to send.
    if (model.CTRL.EN.get_mirrored_value() != 1)
       `uvm_error("RAL", "the predictor did not update CTRL.EN")
    `uvm_info("RAL", $sformatf("mirrored CTRL.EN = %0d, and nobody assigned it",
              model.CTRL.EN.get_mirrored_value()), UVM_MEDIUM)
```

- No hay dirección en ninguna llamada: la dirección está **en el mapa**
- Nadie asignó el espejo. `get_mirrored_value()` devuelve 1 porque el monitor vio la transferencia y el predictor la contó
- Abajo, el driver del capstone está haciendo SETUP, ACCESS y el *wait state* de la lectura, igual que ayer

```
$ bash run.sh
```

·

```
$ code/u9/ral/
```

RAL

Los tests que no escribiste

```
rst_seq = uvm_reg_hw_reset_seq::type_id::create("rst_seq");
rst_seq.model = model;
rst_seq.start(null);

bash_seq = uvm_reg_bit_bash_seq::type_id::create("bash_seq");
bash_seq.model = model;
bash_seq.start(null);
```

- Las dos salen de `uvm-core`. **No saben nada de este DUT**: leen el modelo y generan de ahí el estímulo y el chequeo
- `uvm_reg_hw_reset_seq` lee todo después del reset y lo compara con el valor de reset declarado
- `uvm_reg_bit_bash_seq` escribe y lee **cada bit escribible de cada registro**, uno por uno, en los dos sentidos

RAL

Y por eso el modelo es la spec

```
CTRL.CLR as "WC"   0 UVM_ERROR, and bit_bash did write bit 1
CTRL.CLR as "RW"   2 UVM_ERROR, from uvm_reg_bit_bash_seq
> Writing a 1 in bit #1 of register "model.CTRL" ...
  yielded 'h0000000000000000 instead of 'h0000000000000002
```

- Una palabra cambiada en un string, y una sequence que nadie escribió lo caza con el número del bit
- Es la **trampa 2** de la letra chica del capstone —`CLR` nunca se lee en 1— y ahí hacía falta un scoreboard entero para verla
- El modelo de registros es el único documento de un proyecto que es la spec **y** el código a la vez

`$ bash run.sh` corre las dos versiones y las compara

RAL

Lo que RAL no hace, y hay que decirlo

```
// The honest line of the whole unit. ACC and STATUS have an address, but
// they are not registers: their value is produced by writes to OTHER
// addresses. A register model predicts "what I wrote is what I will read",
// and for these two that sentence is false, so mirror(UVM_CHECK) is turned
// off on them. Reads still update the mirror -- a read is a prediction --
// and the accumulator itself stays where it was in the capstone: in the
// scoreboard, which is the thing that does know how to add.
ACC.VALUE.set_compare(UVM_NO_CHECK);
STATUS.EN.set_compare(UVM_NO_CHECK);
STATUS.OVF.set_compare(UVM_NO_CHECK);
```

- `ACC` y `STATUS` tienen dirección, pero **no son registros**: su valor lo produce una escritura a *otra* dirección
- Un modelo predice *"lo que escribí es lo que voy a leer"*, y para estos dos esa frase es falsa. `set_compare(UVM_NO_CHECK)` lo dice en el modelo
- El acumulador se sigue chequeando donde ya estaba: **en el scoreboard**, que es el que sabe sumar
- La lectura igual actualiza el espejo — una lectura *es* una predicción

RAL

Resumen de la unidad

- El **modelo** es el mapa de la spec en código, y el string de acceso de cada campo es la parte que verifica
- El **adapter** es lo único que sabe del bus: veinte líneas, una vez por protocolo
- La predicción va **por el monitor**, no por la llamada — misma regla que el scoreboard
- Te regala `hw_reset_seq` y `bit_bash_seq`, y te cobra saber **qué no es un registro**
- Y todo esto corre en **Verilator 5.052**, que hasta hoy nadie había mostrado

Ejercicio · Día 8 · 1 de 2

Modelar el mapa de registros

```
$ cd code/ejercicios/d8-ral && cat README.es.md
```

- Es **literalmente** lo primero que te piden en un proyecto con registros: te dan la tabla de la spec y devolvés el modelo
- El DUT y el testbench son los del capstone. El adapter, el predictor y los tests salen de la unidad. **Lo único que escribís es** `apb_reg_block.svh`
- Cuatro registros, seis campos, cuarenta líneas
- El corrector va por etapas: el **mapa** —impreso y comparado con la tabla, sin simular nada—, los **accesos** —las dos sequences de la librería— y **lo que el modelo no puede predecir**

El modelo de referencia en C

El scoreboard que reescribe la spec

- El scoreboard del curso predice **en SystemVerilog**: un `case` con la aritmética de la VTALU. Para cinco operaciones de ocho bits está perfecto
- Para un DSP, un codec o un motor cripto, esa aritmética **ya existe, y en C**: la escribió el equipo de algoritmo, y es contra ese modelo que se firmó la spec
- Reescribirla en SystemVerilog es mantener **dos** modelos y pasarse el proyecto debuggeando la diferencia entre ellos. Eso no es verificar: es tener dos specs
- La salida es no reescribirla. Verilator compila a C++, así que el modelo se **enlaza adentro de la simulación**: sin socket, sin proceso aparte, sin IPC

El modelo de referencia en C

import "DPI-C": una declaración de cada lado

```
   // The reference model lives in vtalu_golden.c and is called from here. This
   // is all that is needed on the SystemVerilog side: one declaration per
   // function, with the LRM types that cross the boundary.
   //
   //   input int  -> int          output int -> int *
   //   function   -> returns      task       -> may consume time (not here)
   //
   // The name after "DPI-C" is the symbol the linker looks for, so it has to
   // match the one in the .c letter by letter.
   import "DPI-C" function int  vtalu_golden(input int op, input int a,
                                             input int b, output int ovf);
   import "DPI-C" function void vtalu_golden_bug(input int on);
```

```
/* op, a and b arrive as SystemVerilog `int` (32-bit signed). The result goes
 * back as the return value and the borrow through a pointer, which is how an
 * `output int` argument crosses the boundary. */
int vtalu_golden(int op, int a, int b, int *ovf) {
   int r;

   a &= 0xff;
   b &= 0xff;
   *ovf = 0;

   switch (op) {
      case ADD_OP: r = a + b; break;
      /* 8-bit operands into a 16-bit result: a - b wraps, and the borrow is
       * the second half of the answer the scoreboard has to check. */
      case SUB_OP: r = a - b; *ovf = (a < b); break;
      case AND_OP: r = a & b; break;
      case XOR_OP: r = a ^ b; break;
      case MUL_OP: r = mutar ? (a * b) & 0xff : a * b; break;
      default:     r = 0; break;   /* no_op and rst_op do not answer */
   }
```

```
    return r & 0xffff;
}
```

- El nombre después de `"DPI-C"` es **el símbolo que busca el linker**: tiene que coincidir letra por letra con el del `.c`
- `input int` viaja como `int`; un `output int` llega como `int *`. Una `function` no puede consumir tiempo; una `task` de DPI puede, pero sólo declarada `context` y llamando de vuelta a una `export "DPI-C" task`
- El `.c` va en la línea de `verilator` como un fuente más. **No hay un flag de DPI**

El modelo de referencia en C

El scoreboard: la predicción se pide, no se escribe

```
// The ONLY difference with the sequences scoreboard, and it is the whole
// section: there is no case with the arithmetic of the spec here. The
// prediction is asked of the C reference model, the one that signed it.
//
// What does not change: the monitor, the analysis port, the FIFO, the
// do_compare and the uvm_error. The golden model replaces SIX LINES -- the
// ones that predict -- and no other piece of the testbench finds out.
function result_transaction predict_result(command_transaction cmd);
   result_transaction predicted;
   int                ovf;

   predicted = result_transaction::type_id::create("predicted");

   predicted.result = vtalu_golden(int'(cmd.op), int'(cmd.A), int'(cmd.B), ovf);
   predicted.ovf    = (ovf != 0);

   return predicted;

endfunction : predict_result
```

- Es **la única** diferencia con el scoreboard de las sequences. El monitor, el analysis port, la FIFO, el `do_compare` y el `uvm_error` no cambian
- El golden model reemplaza las seis líneas de predecir, y ninguna otra pieza del testbench se entera de que hay C del otro lado
- Y el tráfico va en los dos sentidos: el resultado vuelve de C, y `vtalu_golden_bug(1)` va de SystemVerilog a **cambiarle el estado** al modelo

El modelo de referencia en C

Lo que se gana, lo que se paga

Se gana	Se paga
Un solo modelo, el que firmó la spec	los opcodes se sincronizan **a mano**: un `enum` de cada lado
El modelo se testea con las herramientas de C	un `undefined reference` no dice cuál de las dos declaraciones está mal
Aritmética que en SystemVerilog es dolorosa — punto flotante, matrices, cripto—	el modelo de C corre en **tiempo cero**: no puede esperar un flanco
Rendimiento: es una llamada a función, no un IPC	un `segfault` en el modelo se lleva puesta la simulación entera

- La regla para decidir: si la predicción **ya existe en C**, DPI. Si la escribís vos de cero y son cinco líneas, un `case` en SystemVerilog y listo
- Corre: `make u8/dpi` — dos tests contra el modelo sano, y uno contra el mutado

Repaso · Día 8

1 de 8 · El string de acceso

`CLR` se limpia solo cuando le escribís un 1. Lo declarás `"WOC"`, y `bit_bash` y `mirror(UVM_CHECK)` dan cero errores. ¿Qué pasó?

a. Nada: `WOC` es el acceso que describe un campo que se limpia al escribirlo

b. `WOC` no está en el LRM, así que UVM lo trata como un `RW` cualquiera

c. Los dos saltean los accesos `WO*`: ese bit no se testeó ✓

d. El predictor deja el espejo en `x`, y una comparación contra `x` siempre pasa

Verde no es chequeado — `uvm_reg_bit_bash_seq.svh:129-133` saltea todo campo cuyo acceso empiece con `WO` (*"you are not supposed to read them"*), y `do_check` lo saca de la máscara de comparación (`uvm_reg.svh:2782-2788`). El único rastro está en los tiempos: el bashing de `CTRL` dura cuatro transferencias menos. El acceso correcto es `WC` (o `W1C`), y con él el bit sí se batea. Un acceso `WO*` es la forma más barata que hay de apagar un chequeo sin enterarse.

Repaso · Día 8

2 de 8 · Predicción automática o explícita

¿Qué cambia entre `set_auto_predict(1)` y enganchar un `uvm_reg_predictor` al monitor?

- **a** Con auto-predict el modelo cree lo que **quiso** mandar, no lo que pasó ✓
- **b** Nada: el predictor es la implementación interna del auto-predict
- **c** El predictor es más rápido: no arma la transaction del bus
- **d** El auto-predict sólo sirve para el frontdoor, y el predictor también para el backdoor

El que chequea no puede creerle al que estimula — con `set_auto_predict(1)`, `model.CTRL.write()` actualiza el espejo en el momento de la llamada, antes de que el bus haya hecho nada: si el driver manda mal la transferencia, el modelo sigue convencido. Con el predictor, el espejo cambia recién cuando el monitor vio el cable. Es el mismo dibujo del scoreboard del capstone, con una pieza de la librería en lugar de una clase a mano. Y es gratis: el monitor y su analysis port **ya estaban**.

Repaso · Día 8

3 de 8 · volatile

`STATUS` cambia solo, sin que nadie le escriba. ¿Qué querés decir al declararlo `volatile` en `configure()`?

a. Que el espejo no es evidencia: el valor pudo cambiar sin pasar por el bus ✓

b. Que UVM lo va a releer del DUT antes de cada comparación, para no equivocarse

c. Que el campo queda fuera del mapa y deja de tener dirección

d. Que hay que leerlo por backdoor, porque el frontdoor no llega a tiempo

El espejo deja de ser evidencia — un modelo predice *"lo que escribí es lo que voy a leer"*, y para un campo `volatile` esa frase es falsa. Los dos `UVM_WARNING GET_MIRRORED_VAL/VOL` del ejemplo son la librería diciendo exactamente eso, y están en la salida a propósito. Lo que sí hay que chequear se chequea donde se sabe: en el **scoreboard**, y en el modelo va `set_compare(UVM_NO_CHECK)` sobre ese campo para que RAL no invente un error.

Repaso · Día 8

4 de 8 · mirror() contra read()

`model.CTRL.read(status, data)` y `model.CTRL.mirror(status, UVM_CHECK)`. ¿En qué se diferencian?

a `read` va por el bus y `mirror` se queda en el espejo, sin generar una transferencia

b `mirror` escribe el espejo en el DUT, para dejar a los dos iguales

c `read` actualiza el espejo y `mirror` no lo toca, para no tapar un error

d Las dos leen del DUT; `mirror` además compara contra lo que el modelo creía ✓

`mirror` es un scoreboard de registros en una palabra — las dos leen del DUT; la diferencia es que `mirror` compara contra lo que el modelo creía **antes** de la lectura, y si no coincide reporta un `uvm_error` sin que nadie escriba un chequeo. Y una lectura *es* una predicción: las dos actualizan el espejo después.

Repaso · Día 8

5 de 8 · Quién sabe del bus

¿Qué sabe el `uvm_reg_block` sobre el APB?

a. La dirección base y el ancho de `PADDR`, que le llegan por el `uvm_reg_map`

b. Nada: el único que sabe del bus es el adapter ✓

c. Todo: por eso hay un modelo de registros por protocolo

d. Los tiempos de SETUP y ACCESS, para predecir el wait state de la lectura

El modelo no sabe que abajo hay un APB — cambiás el adapter y el mismo modelo maneja un AHB. El adapter son veinte líneas, una vez por protocolo, y es el archivo que te viene con un VIP comprado. Ojo con una: `bus2reg` lo llama **también el predictor**, con el item que vio el monitor, así que un adapter que dependa de algo que puso el driver funciona en un sentido y falla en el otro.

Repaso · Día 8

6 de 8 · DPI y el tiempo

Una `import "DPI-C" function` del golden model, ¿puede esperar un flanco de reloj?

a. Sí, poniendo un `#1` adentro del `.c`

b. No: una `function` corre en tiempo cero ✓

c. Sí, siempre que el `.c` se compile con el soporte de timing de Verilator

d. Sí: el simulador suspende el hilo de C mientras dura la llamada

Tiempo cero, como cualquier `function` — para consumir tiempo hace falta `import "DPI-C" task`, y con una aclaración que casi todos los tutoriales se saltean: el C no puede bloquear por sí mismo. Una task de DPI consume tiempo **sólo** si se declara `context` y desde el C llama de vuelta a una `export "DPI-C" task` de SystemVerilog, que es la que espera el flanco. Y ahí ya no es un golden model, es un modelo de bus.

Repaso · Día 8

7 de 8 · undefined reference

El `.c` está escrito y compila, y el link falla con *undefined reference* a la función del golden model. ¿Qué mirás primero?

a. Que falte el flag de DPI en la línea de `verilator`

b. Que el `.c` esté en un `-f` aparte y no mezclado con los `.sv`

c. El nombre, y el `extern "C"`: el símbolo pudo salir decorado ✓

d. Que la `function` de SystemVerilog esté declarada `virtual` para exportar el símbolo

El compilador no cruza las dos declaraciones: el que las junta es el linker — de ahí que el error hable de algo que está escrito, ahí, a la vista. Casi siempre es una de dos: el nombre no coincide letra por letra, o el archivo se compiló como C++ y el símbolo salió con el nombre decorado. Eso último es literal acá: Verilator le pasa los fuentes del usuario al compilador de C++, así que el `.c` necesita su `extern "C"`. Y no hay flag de DPI: el `.c` va en la línea de `verilator` como un fuente más.

Repaso · Día 8

8 de 8 · El scoreboard que nunca gritó

El scoreboard con golden model en C corre mil operaciones y no reporta ni una. ¿Alcanza?

a. Sí: mil comparaciones sin una sola diferencia es la definición de verificado

b. Sí, siempre que además la cobertura funcional haya cerrado al 100 %

c. No, porque los dos lados comparten el `enum` de opcodes y se anulan entre sí

d. No: un scoreboard que nunca vio un error no está probado ✓

Hay que romper el modelo a propósito — `vtalu_golden_bug(1)` muta la multiplicación y el `run.sh` exige que el scoreboard grite. Si no grita, el testbench está comparando contra sí mismo y nadie se iba a enterar: es el mismo test de mutación que el corrector del capstone hace con `+BUG=1`, del otro lado del cable. Y el `enum` duplicado es real pero es otro síntoma: si fallan **todas** las comparaciones a la vez, es el mapeo; si falla una de cada seis, es el DUT.

Ejercicio · Día 8 · 2 de 2

El modelo de referencia en C, y las dos mutaciones que lo prueban

```
$ cd code/ejercicios/d8-dpi && bash run.sh
```

- El único archivo del ejercicio es `vtalu_golden.c`. Del testbench no se toca una línea
- Faltan dos operaciones: la **resta con su borrow** —que sale por un puntero— y el producto con el gancho de mutación
- El corrector corre tres veces: DUT sano contra tu modelo, **DUT mutado** (`VTALU_BUG=1`), y tu modelo mutado (`+GOLDEN_BUG`)

Capstone 2 · Día 8

El segundo final: una FIFO con backpressure

```
$ cd code/ejercicios/d8-fifo && cat spec.md
```

- Para el que ya entregó el APB. **El protocolo es más simple** —no hay direcciones, ni wait states, ni respuesta de error— y aun así es más difícil
- El del APB es un esclavo **sin memoria útil**: dirección y valor, y el scoreboard puede ser una tabla de cuatro filas. Una FIFO tiene **orden** y **ocupación**, y las dos hay que llevarlas
- El bug de `+BUG=1` no es de datos: `almost_full` se levanta **un lugar tarde**. Los datos siguen saliendo bien y en orden, así que un scoreboard que sólo compara `rd_data` pasa en verde
- La letra chica: las banderas describen el estado **antes** del flanco, `rd_data` llega **un ciclo tarde**, escribir con la FIFO llena **no es un error**, y una lectura simultánea **le hace lugar** a la escritura