Test Plan — Router en malla (`mesh_gnrtr`)

Verificación UVM — Francisco Javier Coto Alcazar y Andres Rojas Barboza

23 de Novimebre 2025

Índice

1. Resumen

Este documento define el plan de pruebas teórico para el diseño de un sistema de ruteo en malla parametrizable. Se describe el DUT, sus capacidades funcionales, las señales de interfaz, el ambiente UVM que se construirá, el plan de aleatorización y la matriz de casos de prueba y esquinas. También se incluye un diagrama de comunicación entre bloques y del ambiente UVM.
El alcance es *funcional*: correcto ruteo según *modo*, manejo de bordes y broadcast, arbitraje, handshakes de push/pop y colas FIFO. No se evalúa síntesis ni timing.

2. Descripción del DUT

2.1. Módulo Top: `mesh_gnrtr`

El DUT es un generador de malla de routers parametrizable:

- Parámetros: `ROWS`, `COLUMS`, `pckg_sz`, `fifo_depth`, `bdcst`.
- Puertos principales (tamaños dependen de los parámetros):
  - `clk`, `reset`.
  - `pndng[ROWS*2 + COLUMS*2]` (salida): pendiente por interfaz de borde.
  - `data_out[ROWS*2 + COLUMS*2][pckg_sz-1:0]` (salida): paquete hacia agentes de borde.
  - `popin[ROWS*2 + COLUMS*2]` (salida): consumo de paquetes recibidos en borde.
  - `pop[ROWS*2 + COLUMS*2]` (entrada): orden de consumir del borde.
  - `data_out_i_in[ROWS*2 + COLUMS*2][pckg_sz-1:0]` (entrada): paquetes provenientes de agentes de borde hacia la malla.
  - `pndng_i_in[ROWS*2 + COLUMS*2]` (entrada): señales de pendiente externas hacia la malla.

2.2. Arquitectura interna relevante

Cada router instancia un bloque de bus con 4 interfaces (UP, RIGHT, DOWN, LEFT) y una tabla de ruteo local (`s_routing_table`). El arbitraje (`Router_arbiter`) selecciona cíclicamente la interfaz con actividad y genera las señales `push_i`/`pop_i`. Las interfaces de bus (`router_bus_interface`) contienen una FIFO de salida (`fifo_flops_no_full`) y señales de pre-habilitación: `pre_psh` y `pre_pop`. La malla conecta routers entre sí y con agentes de borde mediante buffers (`conector`).

2.3. Formato del paquete (`pckg_sz`)

Los campos del paquete se encuentran codificados en los bits más significativos y el payload al final. Se propone la siguiente interpretación:

Campo	Rango de bits
ID de terminal destino (`ntrfs_id`)	[`pckg_sz` − 1 : `pckg_sz` − 8]
Fila destino (`trgt_r`)	[`pckg_sz` − 9 : `pckg_sz` − 12]
Columna destino (`trgt_c`)	[`pckg_sz` − 13 : `pckg_sz` − 16]
Modo de ruteo (1=fila primero, 0=columna primero)	[`pckg_sz` − 17]
Fuente (`src`)	[`pckg_sz` − 18 : `pckg_sz` − 25]
ID (`id`)	[`pckg_sz` − 26 : `pckg_sz` − 33]
Payload	[`pckg_sz` − 34 : 0]

Notas: El campo destino puede ser igual al valor de broadcast (`bdcst`=`{8{1'b1}}`), en cuyo caso todos los terminales aceptan el paquete.

3. Capacidades y requisitos funcionales

F1 Ruteo por modo: Si `mode`=1, priorizar fila; si `mode`=0, priorizar columna. Cuando fila/columna coinciden, seleccionar interfaz local (`ntrfs_id`=4).

F2 Manejo de bordes: En routers de marco limítrofe, las salidas pueden dirigirse hacia interfaces externas (agentes) según reglas de *salida del mesh*. Las opciones de UP/RIGHT/DOWN/LEFT se determinan por comparación de `trgt_r`/`trgt_c` contra `id_r`/`id_c`.

F3 Broadcast: Si el destino es `bdcst`, cualquier interfaz con coincidencia lo acepta (`pre_psh`=1) y el arbitro emite `push_i`/`pop_i` acorde.

F4 Arbitraje: Round-robin sobre `trn` con máquina de estados (`rtr_rbtr_cntrl`). Secuencia típica: *push1* → *pop1* → *cnt1* → *rst*.

F5 Handshakes: `psh = pre_psh & push_i`, `popin = pre_pop & pop_i`. `pre_psh` se activa si el campo destino coincide con el `ntrfs_id` o si destino es broadcast.

F6 FIFO: `pndng`=1 si el contador interno es distinto de cero. Profundidad parametrizable; se reporta overflow (mensajes de DEBUG) al intentar `push` con `count==depth`.

4. Interfaz y señales para el testbench

4.1. Agrupación en una interfaz SystemVerilog (plan)

Se definirá una interfaz SV (`mesh_if`) con los siguientes grupos de señales, parametrizada en `ROWS`, `COLUMS` y `pckg_sz`:

- Reloj y reset: `logic clk, reset`.
- Borde $N =$ `ROWS*2 + COLUMS*2`:
  - `logic pndng[N]`, `logic popin[N]`, `logic pop[N]`.
  - `logic [pckg_sz-1:0] data_out[N]`, `logic [pckg_sz-1:0] data_out_i_in[N]`.
  - `logic pndng_i_in[N]`.

5. Ambiente UVM

5.1. Listado de componentes

El ambiente usará exclusivamente:

- `uvm_env`, `uvm_agent`.
- `uvm_sequencer`, `uvm_driver`, `uvm_monitor`.
- `uvm_scoreboard`, `uvm_subscriber` (para cobertura funcional).
- Clases base: `uvm_object`, `uvm_component`, `uvm_sequence_item`, `uvm_sequence`.

5.2. Transacción y señalización

Cada `uvm_sequence_item` (`mesh_pkt`) contendrá:

- Campos: `dest_term (0..3,4,bdcst)`, `trgt_r`, `trgt_c`, `mode`, `src`, `id`, `payload[`k`]`.
- Metadatos: `edge_agent_id` (interfaz de borde por donde se inyecta), latencia esperada, ruta prevista.

- Señales conducidas por el `driver`: `data_out_i_in[edge]`, `pndng_i_in[edge]`, `pop[edge]`.
- Señales muestreadas por el `monitor`: `data_out[edge]`, `pndng[edge]`, `popin[edge]`.

6. Plan de aleatorización

- Aleatorizar `trgt_r` en [1..`ROWS`] y `trgt_c` en [1..`COLUMS`] con distribución sesgada a bordes (1 y `ROWS`/`COLUMS`).
- Aleatorizar `mode` con probabilidad 50/50, y reforzar casos consecutivos con cambios de modo.
- Aleatorizar `dest_term` en $\{0, 1, 2, 3, 4, \texttt{bdcst}\}$; usar pesos para cubrir cada dirección y broadcast.
- Aleatorizar `src`, `id` y `payload` (tamaño y contenido), con restricciones de *no overflow*: generación de tráfico respeta `fifo_depth` mediante pacing del `push`.
- Aleatorizar `edge_agent_id` (interfaz de inyección) para cubrir índices del arreglo de borde.
- Aleatorizar patrones de `pop` desde los agentes de borde (consumidores) para provocar backpressure y probar arbitraje.

7. Casos de prueba

7.1. Básicos (sanity)

1. **Smoke 1 paquete**: Inyectar un paquete con destino vecino inmediato; verificar que la interfaz esperada emite `popin`=1 y que el `payload` se conserva.
2. **Modo fila/columna**: Ejecutar dos pruebas con `mode`=1 & `mode`=0 y destinos equivalentes; verificar la elección de interfaz conforme a las reglas.
3. **Local delivery**: Destino `trgt_r`=`id_r`, `trgt_c`=`id_c` $\Rightarrow interfazlocal(\texttt{ntrfs\_id} = 4)$.
3. **Broadcast**: Destino `bdcst`; comprobar aceptación en múltiples terminales y comportamiento del arbitro.

7.2. Arbitraje y flujo

1. **Round-robin**: Tráfico simultáneo en 4 interfaces; verificar rotación de `trn` y secuencia `push_i`/`pop_i`.
2. **Backpressure**: Pausar `pop` en receptores; observar acumulación de `pndng` y prioridades del arbitro.
3. **FIFO profundidad**: Empujar hasta `fifo_depth`; verificar `pndng` y ausencia de desbordes. Habilitar variante con overflow para capturar log de DEBUG.

7.3. Bordes y salida de malla

1. **Upper/Lower/Left/Right edges**: Destinos que implican salida del mesh en cada borde; validar selección de interfaz externa correcta.
2. **Esquinas**: Routers en esquinas; ruteos a destinos fuera de rango deben mapear a la interfaz de borde adecuada.

7.4. Robustez

1. **Reset asíncrono**: Aplicar `reset` durante tráfico; verificar reinicio de contadores/estado y ausencia de paquetes fantasma.
2. **Tráfico aleatorio sostenido**: Semillas múltiples, mezcla de broadcast y unicast, variación de `ROWS`/`COLUMS`/`pckg_sz`.

8. Scoreboard y chequeos

El `scoreboard` implementará un modelo de referencia de nivel *paquete* con reglas de ruteo determinísticas. Para cada transacción se verifica:

- **Interfaz esperada**: Dirección emitida por `s_routing_table` (UP/RIGHT/DOWN/LEFT/LOCAL) dado (`mode`, `trgt_r`, `trgt_c`, `id_r`, `id_c`).
- **Integridad**: Campo `payload` y meta-datos intactos en `data_out` del receptor.
- **Arbitraje**: Ausencia de pérdida/reordenamiento no justificada; `popin` observado tras `pop_i`.

9. Cobertura funcional

Se definen *covergroups* en el `subscriber`:

- **Cobertura de modo**: bins {fila_primero, col_primero}.
- **Direcciones**: bins {UP, RIGHT, DOWN, LEFT, LOCAL, BDCST}.
- **Bordes**: bins para routers en {upper_edge, lower_edge, left_edge, right_edge}.
- **Arbitraje**: bins de `trn` y secuencia de estados (push1, pop1, cnt1, rst).
- **FIFO**: niveles `count` y transición a `pndng`=0/1.
- **Cruces**: (modo × dirección × borde).

10. Estrategia de estímulos y restricciones

- Los `driver` aplican paquetes construidos con los campos en Tabla de formato, asegurando que el campo destino sea el `ntrfs_id` del agente de recepción o `bdcst`.
- Los `sequencer` generarán secuencias:
  `smoke_seq`, `mode_sweep_seq`, `broadcast_burst_seq`, `edges_seq`, `stress_seq`.
- Pacing de tráfico para evitar desbordes: no más de `fifo_depth` pushes sin pops correspondientes.

11. Matriz de regresión (semillas)

- Dimensiones: (`ROWS`, `COLUMS`) $\in \{(4,4),(6,6),(8,8)\}$; `pckg_sz` $\in \{32,40,64\}$; `fifo_depth` $\in \{2,4,8\}$.
- random seeds por configuración; reporte de cobertura y `scoreboard` por build.

12. Suposiciones y pendientes

- El mapeo exacto de índices del arreglo de borde a posiciones físicas (UP/RIGHT/DOWN/LEFT) se define al construir el `mesh_if`.
- Se confirmará que `ntrfs_id`=4 representa interfaz local, según implementación de `s_routing_table`.
- La semántica de mensajes de DEBUG se usará sólo para diagnósticos en pruebas específicas.