Proyecto 2 de Verificación funcional

Router

Sebastian Barrantes Perez
Tayron Ruiz Segura
Escuela de Ingeniería Electrónica, Tecnológico de Costa Rica (TEC)
Cartago, Costa Rica
Profesor: Dr.-Ing. Ronny García Ramírez

I. Introducción

Este proyecto valida un router en malla 4×4 (mesh) que interconecta dieciséis terminales periféricas mediante 16 nodos internos organizados en cuatro filas y cuatro columnas. Cada paquete contiene campos de origen, destino y modo de enrutamiento (fila/columna) para realizar tráfico unicast entre terminales del borde. El objetivo funcional del DUT es transportar correctamente los flits desde el puerto emisor hasta el puerto receptor atravesando la malla con la latencia y el protocolo de hand-shake especificados.

La verificación se centra en demostrar, con evidencia medible, que:

- Los paquetes válidos llegan al destino correcto, con integridad de cabecera y carga.
- Las rutas seleccionadas (por filas y por columnas) se aplican según el modo indicado en el encabezado.
- Las latencias extremo a extremo cumplen los límites temporales (aceptación y servicio).
- El DUT se comporta correctamente bajo carga, incluyendo escenarios de congestión/saturación y arbitraje.
- Se manejan colisiones (múltiples orígenes hacia un mismo destino) sin pérdida ni duplicación.
- Las direcciones inválidas son detectadas y tratadas conforme a la especificación (descartes/flags).
- El reset durante tráfico deja al sistema en un estado conocido y recuperable.

El enfoque es un ambiente UVM con 16 agentes (uno por terminal) que generan y capturan tráfico de manera concurrente. Cada agente incluye sequencer, driver y monitor; los monitores publican eventos a un scoreboard que empareja entradas/salidas por clave y verifica destino, orden y latencia. Además, se instrumenta cobertura funcional (covergroups sobre destino, fila, columna y modo) y aserciones de protocolo/temporización. Todas las métricas (matches, misses, latencias, claves de enrutamiento y conteos de cobertura) se consolidan en reportes CSV y en la base de datos de coverage para análisis y trazabilidad del avance hacia el objetivo de 95 % de cobertura del DUT.

II. Plan de Pruebas (Testplan)

El plan de pruebas se centra en tráfico unicast entre los 16 terminales del router en malla 4x4. Sobre un mismo banco de pruebas UVM se aleatorizan cuatro dimensiones principales: (1) la cantidad de mensajes que inyecta cada terminal, (2) el tiempo de espera entre envíos sucesivos (`delay_cycles`), (3) la presencia o no de errores deliberados en cada paquete y (4) la dirección de destino, repartida entre terminales válidos e inválidos. El objetivo es doble: por un lado, validar que los mensajes válidos lleguen exactamente al terminal que les corresponde; por otro, medir la latencia y el ancho de banda observados bajo distintos patrones de carga. Adicionalmente, se incluyen aserciones en la interfaz `router_if` para vigilar el protocolo de las FIFOs. Todas las simulaciones generan un archivo CSV con los datos de entradas y salidas, que luego se usa para graficar distribuciones de latencia y ancho de banda por escenario.

II-A. Escenario GENERAL

Objetivo: representar la operación nominal del router, con varios terminales activos, tráfico moderado y una pequeña cantidad de errores esporádicos.
Cantidades de transacciones: cada secuencia `gen_item_seq` genera entre 60 y 80 paquetes por terminal (`num_items = $urandom_range(60,80)`), de modo que con 16 terminales se obtienen del orden de 960–1280 mensajes por corrida.
Tiempos entre envíos: en `drv_item` se privilegian retardos medios, con algunos cortos y largos: 15 % de los mensajes usan `delay_cycles` entre 0–5 ciclos, 70 % entre 5–10 ciclos y 15 % entre 10–20 ciclos.
Errores y destinos: la tasa de error `error_rate` se limita al rango 0–10 %, y cada paquete decide si lleva error según esa tasa. La dirección de destino `dest_addr` nunca coincide con el origen (`dest_addr != src_id`) y, en este escenario, se escoge 90 % de las veces dentro del rango de terminales válidos (0–15) y 10 % como dirección inválida (16–32). El modo de ruteo (`ROW_FIRST` o `COL_FIRST`) se deja libre para repartir tráfico por ambos algoritmos.

II-B. Escenario SATURATION

Objetivo: forzar una situación de congestión donde muchos terminales intentan inyectar tráfico casi de manera continua, para estresar las FIFOs internas y observar el comportamiento del router al borde de su capacidad.
Cantidades de transacciones: cada terminal genera entre 80 y 100 mensajes (`$urandom_range(80,100)`), por lo que la malla completa puede movilizar del orden de 1280–1600

paquetes por escenario.
Tiempos entre envíos: se favorecen fuertemente los retardos cortos: 70 % de los mensajes usan una separación de 0–5 ciclos, y el resto se reparte en retardos medios y largos (5–10 y 10–20 ciclos). Esto hace que varios terminales estén empujando tráfico simultáneamente.
Errores y destinos: la tasa de error permitida se amplía al rango 0–20 %. Cuando se activa `error_flag`, se aumenta la probabilidad de que la dirección sea inválida (60 % válidas, 40 % inválidas); cuando no hay error, se mantiene una distribución más "limpia" (90 % válidas, 10 % inválidas). De esta forma se cubren dos casos dentro del mismo escenario: congestión con tráfico bien formado y congestión con cabeceras corruptas.

II-C. Escenario COLLISION

Objetivo: comprobar el comportamiento del router cuando muchos orígenes intentan llegar al mismo destino, es decir, se provoca una colisión de destino en un único terminal.
Cantidades de transacciones: cada terminal envía entre 50 y 60 paquetes (`$urandom_range(50,60)`), de modo que la prueba moviliza aproximadamente entre 800 y 960 mensajes por corrida.
Tiempos entre envíos: para este escenario se usan retardos siempre cortos, con `delay_cycles` en el rango 0–5 ciclos para todos los mensajes, de manera que los paquetes lleguen casi pegados entre sí.
Errores y destinos: en `drv_item` se fuerza `dest_addr == 5` por defecto, de forma que todos los terminales apunten al mismo receptor válido. La tasa de error se mantiene en el rango 0–20 %, pero el interés principal aquí no es la corrupción de datos sino cómo arbitra el DUT cuando se genera un "embudo" hacia un solo terminal.

II-D. Escenario INVALID

Objetivo: validar la robustez lógica del router ante direcciones fuera de rango y cabeceras ilegales, comprobando que no se entreguen paquetes inválidos como si fueran normales.
Cantidades de transacciones: cada secuencia genera entre 50 y 60 mensajes por terminal (`$urandom_range(50,60)`), lo que da entre 800 y 960 paquetes totales por escenario.
Tiempos entre envíos: igual que en colisión, se utilizan retardos cortos en el rango 0–5 ciclos, lo que mantiene una presión de tráfico razonable mientras se exploran errores.
Errores y destinos: se fuerza una tasa de error alta, con `error_rate` aleatorizado entre 30 % y 40 %. En este modo, la dirección de destino se elige de forma que sólo el 40 % de los paquetes vayan a terminales válidos y el 60 % usen direcciones fuera de rango. Además, cuando `error_flag` está activo, la función `term_to_rc()` asigna filas y columnas de destino en regiones inválidas de la malla, lo que permite observar cómo el DUT reacciona ante coordenadas imposibles.

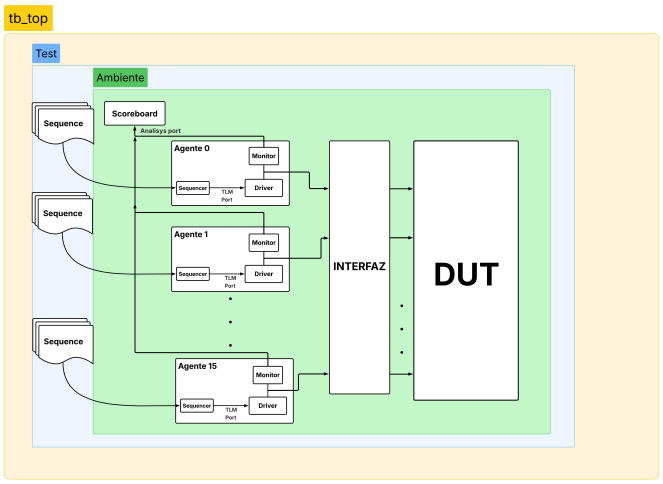

Figura 1. Diagrama del ambiente de pruebas para el Router

III. Diagrama del Ambiente de Verificación

IV. Clases Implementadas

IV-A. Testbench `tb_top`

Aquí se importan las librerías de UVM, se incluye la RTL del router y el resto de archivos del ambiente, como tipos y transacciones, secuencias, drivers, monitores, agente, *scoreboard*, ambiente, test y el archivo de aserciones `router_dut_sva.sv`.

En la parte superior se fijan los parámetros de la malla con `ROWS = 4`, `COLUMS = 4`, `PCK_SZ = 40` y `N_TERMS = 16`. A partir de ellos se declaran los arreglos de señales que conectan el DUT con el testbench y se genera un reloj de periodo 10 junto con la señal de `reset`. El DUT se instancia con el módulo `mesh_gnrtr`, configurado para la malla 4x4, una FIFO de profundidad 4 y una máscara de broadcast en uno. Sobre esta instancia se hace un `bind` del módulo `router_dut_sva`, que agrega aserciones a nivel RTL.

Para modelar los 16 terminales externos se crean 16 interfaces `router_if` en el arreglo `term_if`. Un bloque `generate` realiza el cableado uno a uno entre las señales del DUT y cada interfaz. Otro bloque `generate` usa `uvm_config_db` para pasar la interfaz correspondiente al driver y al monitor de cada agente. Al inicio de la simulación se aplica un pulso de `reset`, se registra la interfaz del terminal cero en `uvm_test_top` y se llama a `run_test("base_test")`. Finalmente, se habilita el volcado de ondas en el archivo `waves.fsdb` para facilitar la depuración.

IV-B. Interfaz `router_if`

La interfaz `router_if` encapsula las señales que conectan cada terminal del banco de pruebas con la malla de routers. Por el lado de entrada hacia el DUT se exponen `data_in`, `pndng_in` y `popin`, que representan el dato que llega al router, la indicación de dato pendiente y el acuse de recibo

cuando el DUT acepta el paquete. En el sentido contrario se usan data_out, pndng y pop, que modelan el dato que el router entrega hacia el terminal, el aviso de dato disponible y el acuse de consumo por parte del testbench.

Además de agrupar las señales, la interfaz incluye un pequeño conjunto de aserciones temporales. Con los parámetros MAX_IN_LAT y MAX_OUT_LAT se limita el número de ciclos que pueden pasar entre un pedido de transferencia y el acuse correspondiente en cada sentido. Otras propiedades revisan que los datos se mantengan estables mientras no llega el acuse y que no se aserte pop o popin si no hay un paquete pendiente. De esta forma la interfaz documenta el protocolo de uso y ayuda a detectar violaciones de handshake tanto en el DUT como en el entorno de verificación.

IV-C. Transacción de driver drv_item

La clase drv_item es el ítem que generan las secuencias y consumen los drivers para inyectar paquetes al router. Contiene campos aleatorizables para la dirección de destino, el retardo entre envíos, la tasa de error, el modo de ruteo y una bandera que indica si el paquete debe ir corrupto. También guarda el modo de prueba en test_mode, junto con los campos decodificados de la malla como fila, columna, identificadores de origen y destino y el pkt_id. El vector data_in y el flit interno de 40 bits son los que finalmente se mandan por la interfaz router_if.

Las restricciones de drv_item ajustan el comportamiento según el escenario. En *GENERAL* se favorecen retardos medios y destinos válidos, con una tasa de error baja. En *SATURATION* se privilegian retardos cortos y se permite aumentar la proporción de direcciones inválidas cuando se activa la inyección de errores. En *COLLISION* todos los paquetes se dirigen al mismo terminal, mientras que en *INVALID* se fuerza una mayoría de destinos fuera de rango. La función term_to_rc traduce el identificador de terminal a coordenadas fila–columna, generando coordenadas inválidas cuando error_flag está activo. Con esa información build_flit arma el paquete con los campos de ID, origen, destino, modo y coordenadas, y from_flit permite reconstruir esos campos a partir de un vector observado a la salida del DUT.

IV-D. Transacción de monitor mon_item

La clase mon_item es la transacción que usan los monitores para reportar lo que ven hacia el *scoreboard*. Cada objeto representa un evento puntual asociado a un terminal. El campo ev_kind indica si se trata de una entrada al router (EV_IN) o de una salida (EV_OUT). El campo data guarda el paquete completo de 40 bits capturado en la interfaz, y mon_id identifica el terminal que lo observó. Además, time_stamp registra el instante de simulación en que ocurrió el evento, lo que luego permite calcular latencias.

La clase se declara como uvm_sequence_item y usa las macros de registro de UVM para poder imprimir, copiar y trazar estos eventos de forma automática.

IV-E. Configuración de agente router_agent_cfg

La clase router_agent_cfg es un objeto de configuración ligero que se usa para parametrizar cada router_agent. Su campo principal es term_id, que indica a qué terminal externo del Router corresponde ese agente y permite que el monitor etiquete correctamente los eventos que observa. En env se crea una instancia distinta por cada agente, se asigna el term_id adecuado y se envía tanto al driver como al monitor mediante uvm_config_db, de modo que todos los componentes del agente compartan la misma vista de su terminal.

IV-F. Secuencia de generación gen_item_seq

La clase gen_item_seq es la secuencia que genera los ítems drv_item que consume cada driver. Tiene un campo scenario que selecciona el tipo de tráfico a producir y un seq_id que se usa como identificador de origen, de modo que cada terminal del Router marca sus paquetes con un src_id distinto. En body() se elige el número de mensajes a enviar según el escenario y se recorre un ciclo donde, para cada paquete, se crea un drv_item, se fija el test_mode apropiado, se asignan src_id y pkt_id, se llama a randomize() y luego a build_flit() para llenar data_in antes de finalizar el ítem.

La lógica del case sobre scenario controla tanto el rango de num_items como el tipo de tráfico: en GENERAL se envían entre 60 y 80 paquetes por terminal, en SATURATION entre 80 y 100, y en los escenarios COLLISION e INVALID entre 50 y 60 paquetes. Cada iteración arranca y termina el ítem con las llamadas estándar de UVM, de modo que el sequencer entregue al driver una secuencia coherente de paquetes ya construidos según el escenario activo.

IV-G. Driver

El driver toma los ítems drv_item que llegan desde el sequencer y los convierte en transacciones sobre la interfaz router_if. En build_phase obtiene la interfaz virtual y el objeto router_agent_cfg, de modo que conozca el terminal que está manejando. Durante run_phase inicializa las señales de entrada hacia el Router y limpia la FIFO interna fifo_in, donde se almacenan los paquetes en espera de ser enviados.

El comportamiento se divide en dos hilos. El primero recibe las transacciones del sequencer, aplica el retardo delay_cycles indicado en cada ítem, registra por log el origen y destino y va agregando los flits a fifo_in. El segundo hilo atiende la interfaz en cada flanco de reloj, si la FIFO tiene datos activa pndng_in, presenta en data_in el elemento más antiguo y espera a que el DUT aserte popin para retirarlo de la FIFO. Si no hay datos pendientes mantiene pndng_in en cero y pone data_in en cero, respetando el protocolo de handshake de la interfaz.

IV-H. Monitor

La clase monitor observa la interfaz router_if de cada terminal y traduce la actividad en objetos mon_item que

luego consume el *scoreboard*. En build_phase obtiene la interfaz virtual, la configuración router_agent_cfg con el term_id y crea el mon_analysis_port usado para publicar eventos.

El comportamiento se divide en dos tareas. watch_inputs espera a que salga del reset y, en cada flanco de reloj, cuando el DUT aserta popin, captura el paquete de entrada en data_in, marca el evento como EV_IN, asigna el identificador del terminal y registra el tiempo. consume_outputs hace algo similar en la salida: mientras pndng está en uno, aserta pop durante un ciclo, toma data_out, crea un mon_item con EV_OUT y vuelve a bajar pop. En esta tarea también se muestrea el *covergroup* cg_hdr, que acumula cobertura sobre destino, modo de ruteo y coordenadas fila–columna. Ambas tareas se lanzan en paralelo en run_phase, de modo que el monitor sigue al mismo tiempo lo que entra y lo que sale por el terminal.

IV-I. Agente router_agent

El agente router_agent encapsula los tres componentes básicos asociados a cada terminal del router: driver, monitor y sequencer. En build_phase se crean las instancias de estos bloques usando la fábrica de UVM, de modo que el ambiente pueda tener un agente por cada una de las dieciséis interfaces externas del DUT.

En connect_phase se enlaza el seq_item_port del driver con el seq_item_export del sequencer. Con esto, las transacciones generadas en las secuencias llegan al driver correspondiente y se aplican a la interfaz física asociada al agente. La configuración de la interfaz virtual y del identificador de terminal se realiza desde el ambiente mediante uvm_config_db.

IV-J. Scoreboard

El scoreboard recibe todos los mon_item publicados por los monitores mediante m_analysis_imp. Con el paquete de 40 bits arma una clave key_t que incluye fila, columna, modo y carga útil. Esa clave se usa como índice de un arreglo asociativo exp_q, donde cada entrada es una cola de eventos de entrada pendientes. Cuando llega un EV_IN, el paquete se guarda en la cola que corresponde a su clave y se actualizan los contadores de tráfico de entrada.

Cuando llega un EV_OUT, el scoreboard busca la misma clave. Si hay elementos, toma el más antiguo, calcula la latencia con las marcas de tiempo y escribe una fila "OK" en router_report.csv con tiempos, terminales, datos, fila, columna, modo y los campos de origen y destino. También revisa si la fila y columna describen un terminal válido y si el destino observado coincide con el terminal que recibió el paquete; en caso contrario lo reporta en los mensajes de log. Si aparece una salida sin entrada asociada, se marca como *UNEXPECTED_OUT* y también se registra en el CSV. Al final de la simulación report_phase recorre el arreglo asociativo, cuenta las entradas que aún están en las colas, las reporta como *NO_OUTPUT* y resume el número de entradas, salidas, coincidencias, fallos y paquetes pendientes antes de cerrar el archivo.

IV-K. Ambiente env

La clase env reúne los componentes principales del banco de pruebas. Aquí se declara un scoreboard único para todo el tráfico y un arreglo de router_agent con NUM_TERMS posiciones, de modo que haya un agente por cada terminal externo del router. En build_phase se crea el scoreboard y, dentro de un foreach, se instancia cada agente con un nombre indexado. Para cada uno se construye un objeto router_agent_cfg, se asigna el term_id correspondiente y se envía esta configuración al driver y al monitor mediante uvm_config_db.

En connect_phase el ambiente conecta el mon_analysis_port de cada monitor con la implementación de análisis del scoreboard. Con esto, todos los eventos capturados en los diferentes terminales llegan al mismo punto de chequeo, donde se emparejan entradas y salidas y se generan las métricas finales de la simulación.

IV-L. Test base_test

El test base_test arma el escenario global de verificación. En build_phase baja la verbosidad de reportes a UVM_LOW, toma la interfaz virtual router_if desde la config_db, crea el ambiente env con sus dieciséis agentes y el scoreboard, y construye un arreglo de secuencias gen_item_seq donde cada posición se asocia a un terminal del router y se aleatoriza una configuración inicial.

En run_phase define la cola de escenarios a ejecutar en este orden: GENERAL, SATURATION, COLLISION e INVALID. Para cada uno levanta la objeción de fase, recorre los dieciséis terminales y, en paralelo, arranca una secuencia por agente asignando el tipo de escenario y el identificador de origen. Después espera a que terminen todos los hilos asociados al escenario, deja un margen de ciclos para que el tráfico se drene en la malla y pasa al siguiente caso. Al final de la lista baja la objeción y permite que UVM cierre la simulación con el resumen del scoreboard.

V. Procesamiento de resultados

V-A. Archivo router_report.csv

El *scoreboard* abre el archivo router_report.csv en build_phase, escribe una cabecera con los nombres de las columnas y, durante la simulación, va agregando una fila por cada evento relevante. Cuando una salida EV_OUT encuentra su entrada EV_IN correspondiente, se registra una línea con estado OK; si aparece una salida sin entrada previa se marca como UNEXPECTED_OUT; y si al final quedan entradas sin salida se etiquetan como NO_OUTPUT en report_phase. Así el archivo resume, en texto plano, qué paquetes circularon correctamente y dónde hubo problemas.

Cada fila contiene el tiempo de transmisión y recepción, el identificador del terminal que envió y el que recibió, los datos completos observados en la entrada y en la salida, la

latencia en ciclos y la clave de ruteo formada por fila, columna, modo y carga útil. Además se guardan los campos `src_id` y `dst_id` decodificados del paquete, lo que permite revisar si el mensaje terminó en el terminal esperado. Con esta estructura se pueden filtrar fácilmente los paquetes correctos o los casos problemáticos usando cualquier herramienta de análisis.

Tras la simulación, `router_report.csv` se usa como entrada de scripts externos para generar gráficas y estadísticas. En particular, se procesan las filas con estado `OK` para obtener histogramas de latencia, curva de latencia máxima por escenario y estimaciones de ancho de banda por terminal. También se revisan las filas `UNEXPECTED_OUT` y `NO_OUTPUT` para localizar rápidamente patrones de error sin depender solo de las ondas del simulador.

V-B. Script de análisis en Python

El script de postproceso lee `router_report.csv`, filtra únicamente las filas con estado `OK` y calcula la latencia promedio por terminal de recepción, junto con una latencia global. Con esos datos genera una gráfica de barras donde se comparan las medias por terminal contra el valor global. A partir de los tiempos de recepción también construye una serie de paquetes por ciclo de reloj y, de ahí, obtiene el ancho de banda mínimo, promedio y máximo, que se resume en una segunda figura.

Opcionalmente el script produce una curva de ancho de banda en el tiempo en función del ciclo de reloj. Todas las imágenes se guardan en formato `.png`, con un prefijo configurable, y el periodo de reloj se ajusta mediante un parámetro de línea de comandos. De esta forma el análisis numérico y las gráficas quedan desacoplados del testbench y se pueden repetir sobre cualquier archivo `router_report.csv` generado por nuevas simulaciones.

Para ejecutarlo en Windows se coloca el archivo `gen_plots.py` en la misma carpeta donde se generó `router_report.csv`, se abre una terminal en ese directorio y se llama al script con `py -3.13 gen_plots.py`. En caso de usar un nombre de CSV diferente se puede pasar como argumento, por ejemplo `py -3.13 gen_plots.py otro_archivo.csv`. En un entorno Linux el flujo es el mismo, pero el comando típico sería `python3 gen_plots.py` o `python3 gen_plots.py router_report.csv`, siempre que el script y el archivo CSV se encuentren en el mismo directorio.

V-C. Gráficas generadas a partir del CSV

A partir de `router_report.csv` el script genera tres figuras en formato `.png`. La Figura 2 muestra la latencia promedio por terminal de destino como barras azules y, al final, una barra adicional que representa la latencia global. La línea punteada horizontal repite este valor global y sirve como referencia visual para ver qué terminales están por encima o por debajo del promedio.

La Figura 3 resume el ancho de banda global con tres barras: mínimo, promedio y máximo de paquetes por ciclo, lo que da una medida rápida de cuán ocupado estuvo el Router durante la simulación.

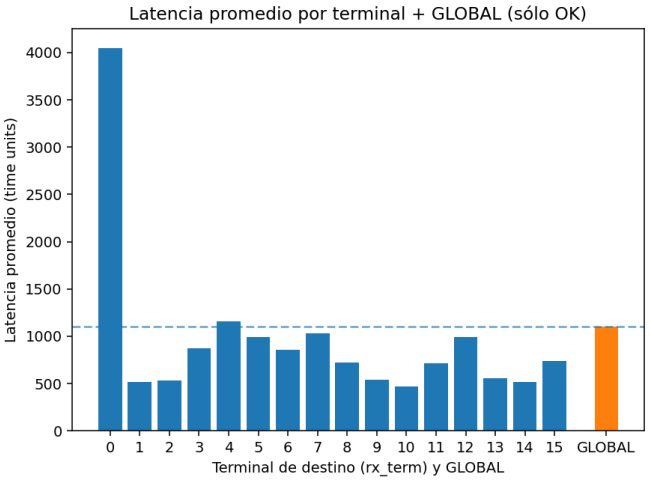

Figura 2. Latencia promedio por terminal y valor global a partir de `router_report.csv`.

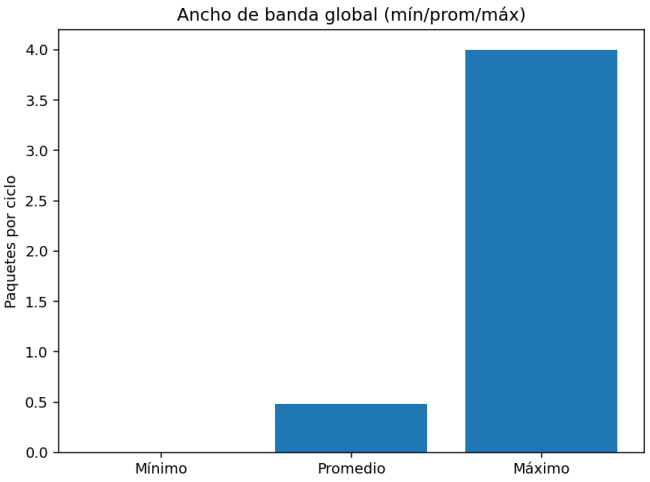

Figura 3. Ancho de banda global mínimo, promedio y máximo (paquetes por ciclo).

Por último, la Figura 4 grafica el ancho de banda global en función del ciclo de reloj, lo que permite identificar zonas de mayor actividad o periodos casi sin tráfico. Con estas tres vistas se obtiene, en forma compacta, una idea clara del desempeño del Router en términos de latencia y uso de ancho de banda.

VI. Aserciones

Con el fin de garantizar la correcta operación del protocolo de comunicación entre el banco de terminales del *testbench* y el diseño bajo prueba (DUT), se implementó un conjunto de aserciones.

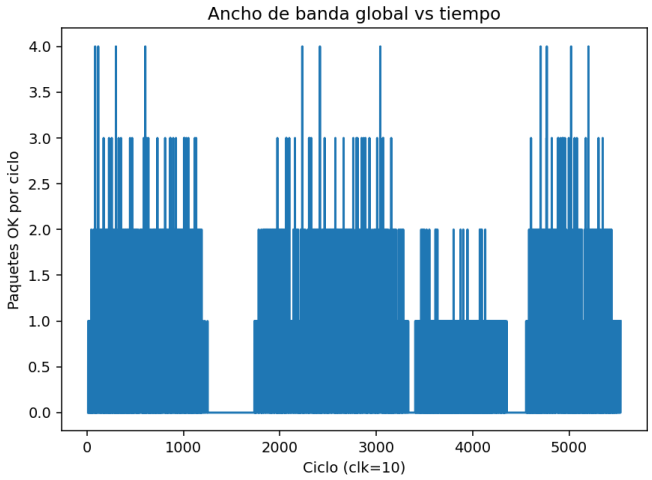

Figura 4. Evolución del ancho de banda global en función del ciclo de reloj.

VI-A. *Latencia máxima del handshake de entrada*

La primera propiedad comprueba que el mecanismo de *handshake* de entrada (`pndng_in` → `popin`) se complete en un número acotado de ciclos. En términos informales, la propiedad puede enunciarse como:

> Si en un flanco de subida del reloj la señal `pndng_in` pasa de 0 a 1 (es decir, el *testbench* anuncia un nuevo paquete), entonces debe existir un ciclo entre 1 y `MAX_IN_LAT` ciclos más tarde en el que la señal `popin` sea igual a 1.

Con ello se detectan bloqueos o latencias excesivas en la interfaz de entrada. En el código SVA esta regla se codifica mediante una implicación temporal disparada por el flanco de subida de `pndng_in`.

VI-B. *Estabilidad de datos durante el handshake de entrada*

El protocolo establece además que, una vez que `pndng_in` se mantiene en alto y el DUT aún no ha afirmado `popin`, los datos presentados en `data_in` deben permanecer estables. La propiedad puede resumirse así:

> Mientras `pndng_in` sea 1, `popin` sea 0, no se esté en el ciclo inmediatamente posterior al flanco de subida de `pndng_in` y tampoco se haya activado `popin` en el ciclo anterior, el vector `data_in` no debe cambiar de valor.

Esta aserción evita la corrupción de información en la ventana crítica en la que el DUT está decidiendo si consume o no el paquete anunciado.

VI-C. *Latencia máxima del handshake de salida*

De manera simétrica, se verifica un límite temporal para el canal de salida. Cuando el DUT coloca un nuevo paquete en la salida (flanco de subida de `pndng`), la señal de reconocimiento `pop` debe activarse dentro de un máximo de `MAX_OUT_LAT` ciclos. De forma informal:

> Si en un flanco de reloj `pndng` pasa de 0 a 1, entonces en algún ciclo entre 1 y `MAX_OUT_LAT` ciclos después, la señal `pop` debe tomar el valor 1.

Esta propiedad permite detectar problemas en el consumo de datos por parte del entorno de prueba o posibles inconsistencias internas del DUT.

VI-D. *Coherencia del protocolo de control*

Por último, se incluyen dos aserciones que imponen invariantes de coherencia sobre el protocolo de control:

- **Coherencia de salida**: cada vez que se observa un `pop` alto en un flanco de reloj, la señal `pndng` debe estar activa en ese mismo ciclo. Es decir, no se puede reconocer la lectura de un paquete si el DUT no había indicado previamente que existía un dato pendiente.
- **Coherencia de entrada**: de forma análoga, cada vez que `popin` es 1, la señal `pndng_in` debe ser también 1 en ese ciclo. Con ello se evita que el DUT reconozca entradas "fantasma" cuando el *testbench* no ha anunciado ningún paquete.

VII. Cobertura funcional de tráfico

Además de las aserciones temporales, se incorporó un conjunto de *covergroups* en el monitor de cada terminal con el objetivo de medir qué tan exhaustivamente se ejercita el espacio de tráfico generado por las secuencias. La intención es comprobar que todas las terminales lleguen a actuar como origen y destino, y que ambas modalidades de encaminamiento (`COL_FIRST` y `ROW_FIRST`) se utilicen de forma equilibrada.

VII-A. *Cobertura en la interfaz de entrada*

En cada instancia del monitor se definió un *covergroup* de entrada, `cg_entrada`, parametrizado mediante una función `sample( src, mode )`. En cada flanco de reloj se muestrean dos campos del paquete recibido en `data_in`:

- **ID de origen** (`src`): se toma el campo `SRC` del encabezado del paquete y se cubre con un *coverpoint* `cp_src`, cuyos *bins* abarcan los valores de terminal válidos en el rango $[0, 14]$. De esta forma se verifica que, a lo largo de la simulación, cada terminal de la malla llegue a utilizarse como origen de tráfico al menos una vez.
- **Modo de ruteo** (`mode`): se toma el bit de modo del encabezado y se cubre mediante el *coverpoint* `cp_mode`, con dos *bins* explícitos: uno para `0` (ruteo con prioridad de columnas) y otro para `1` (ruteo con prioridad de filas).

Adicionalmente se define el cruce `cp_src_mode`, que permite observar qué combinaciones {origen, modo} se han ejercitado. Esto aporta información sobre si ciertas terminales sólo se usan bajo un modo concreto o si efectivamente se exploró la matriz completa de posibilidades.

VII-B. Cobertura en la interfaz de salida

De manera análoga, cada monitor contiene un *covergroup* de salida, `cg_salida`, parametrizado mediante `sample( dst, mode )`. En este caso se muestrean los paquetes que el DUT entrega en `data_out`:

- **ID de destino** (`dst`): el *coverpoint* `cp_dst` define *bins* para los destinos válidos en el rango $[0, 14]$, garantizando que todos los terminales lleguen a recibir al menos un paquete durante las pruebas.
- **Modo de ruteo en salida**: se reutiliza el bit de modo para construir el *coverpoint* `cp_mode`, con los mismos dos *bins* que en la entrada.

El cruce `cp_dst_mode` permite analizar la distribución de tráfico que efectivamente salió por el DUT, distinguiendo si ciertas combinaciones {destino, modo} quedan sin ejercer o tienden a concentrarse en un subconjunto de terminales.

VII-C. Agregación global de cobertura por entorno

Para obtener una métrica global de cobertura funcional, el entorno UVM recorre todas las instancias de agentes al final de la simulación (fase `report_phase`) y consulta la cobertura de cada *covergroup* mediante el método estándar `get_coverage()`. A partir de estos valores se calculan:

- La cobertura media de entrada, *Cobertura GLOBAL ENTRADAS*, promediando la cobertura de `cg_entrada` en todos los monitores activos.
- La cobertura media de salida, *Cobertura GLOBAL SALIDAS*, promediando la cobertura de `cg_salida`.
- Una métrica global *TOTAL*, obtenida como el promedio simple entre la cobertura de entradas y de salidas.

Estos porcentajes se reportan mediante un mensaje `uvm_info` al finalizar el test, lo que permite disponer de una medida sintética de qué tan bien se ejercitó el espacio de tráfico sin necesidad de inspeccionar manualmente cada instancia en el visor de cobertura.

En las corridas realizadas para el escenario regresión final, los resultados obtenidos fueron:

- **Cobertura GLOBAL ENTRADAS**: 100,00 %,
- **Cobertura GLOBAL SALIDAS**: 100,00 %,
- **Cobertura TOTAL**: 100,00 %.

Estos valores indican que, bajo el conjunto de secuencias desarrollado, todas las terminales actuaron al menos una vez como origen y destino, en ambos modos de encaminamiento definidos, satisfaciendo el criterio de cobertura funcional planteado para el proyecto.

VIII. Comandos para correr la compilación y simulación

- compilar: vcs -Mupdate tb_top.sv -o salida -full64 -sverilog -kdb -lca -debug_access+all +acc -debug_region+cell+encrypt -l log_test +lint=TFIPC-L -cm line+tgl+cond+fsm+branch+assert -ntb_opts uvm-1.2 +UVM_VERBOSITY=UVM_LOW
- simular: ./salida -cm line+tgl+cond+fsm+branch+assert +UVM_TIMEOUT=58000
- cobertura: verdi -cov -covdir salida.vdb&

IX. Link GitHub

Link: https://github.com/SebasJB/Proyecto-2-Verificacion-Funcional.git

Referencias