Escuela de Ingeniería Electrónica
Licenciatura en Ingeniería Electrónica
Verificación funcional de circuitos
II Semestre 2024

TEC | Tecnológico de Costa Rica

Proyecto 1

Verificación del Bus Driver

David Medina Mayorga, Pedro Medinila Robles

1. Descripción del DUT

Para el diseño del ambiente de verificación, se toma en cuenta el funcionamiento del dispositivo, que en este caso es un manejador de bus de datos que permite la transmisión de información entre diferentes periféricos. Estos periféricos serán emulados mediante FIFOs, y el objetivo específico es verificar el correcto funcionamiento del bus. Las FIFOs serán descritas a nivel de software y los datos transmitidos serán definidos como específicos o aleatorios, dependiendo del tipo de prueba que se ejecute.

El ambiente de verificación se organiza en torno a dos módulos principales: el controlador del bus de datos, también conocido como Bus Driver, y los dispositivos FIFO. El Bus Driver tiene la responsabilidad de administrar el acceso de los diferentes periféricos al bus de datos, asegurando que las comunicaciones se realicen de manera ordenada y sin conflictos. Por otro lado, los dispositivos FIFO funcionan como una interfaz intermedia que almacena temporalmente los datos transmitidos entre el DUT y el ambiente de verificación, simulando el comportamiento de periféricos reales conectados al bus.

1.1. Señales

- **pndng:** si la FIFO está vacía, la señal `pndng` es 0.
- **push:** señal de control que indica que se debe realizar una operación de `push` (es decir, ingresar datos a la FIFO).
- **pop:** señal de control que indica que se debe realizar una operación de `pop` (es decir, extraer datos de la FIFO).

- **Dpush:** datos a ser almacenados en la FIFO.
- **Dpop:** datos a ser extraídos de la FIFO.
- **Broadcast:** representa una señal con un valor en los bits de ID por defecto de 8 bits (todo 1s: `11111111`). Esto significa que el módulo tiene una funcionalidad de transmisión de mensajes a múltiples destinatarios simultáneamente (*broadcast*).

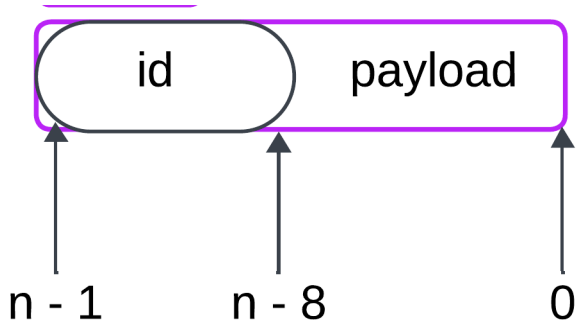


Figura 1: Formato del paquete de información

2. Escenarios

2.1. Casos Generales

Como casos generales o de uso común se tienen las pruebas con cada transacción:

- **Enviar datos:** cada periférico debe poder enviar un paquete a otro periférico a través del bus de datos.
- **Recepción de datos enviados:** si se envía un paquete a cierto periférico, este debe recibirlo dado su ID, y el *payload* debe coincidir con el enviado.
- **Funcionamiento del reset:** el dispositivo debe ser capaz de reiniciarse correctamente cuando se active la señal de reset, vaciando todas las FIFOs y anulando las transacciones pendientes.
- **broadcast:** si un periférico realiza un *Broadcast* con el identificador 8'd1, todos los periféricos deben recibir el *payload* enviado.

2.2. Casos de Esquina

- **Varios envían a la vez:** este caso prueba el comportamiento del bus cuando varios periféricos intentan enviar datos al mismo tiempo.
- **Todos envían a la vez:** este caso prueba el comportamiento del bus cuando cada periférico envía datos simultáneamente.

- **Se envía un paquete con un ID ilegal:** con un número determinado de periféricos, se utiliza un ID que no existe en el sistema.
- **Todos hacen broadcast simultáneamente:** similar al caso anterior, pero en este caso todos los periféricos intentan hacer *broadcast* al mismo tiempo.
- **Varios hacen broadcast:** prueba el comportamiento del bus cuando varios periféricos realizan *broadcast* simultáneamente.
- **Se envía a sí mismo:** consiste en enviar un dato desde un periférico para volver a recibirlo en el mismo periférico.
- **Uno solo envía una ráfaga de paquetes:** prueba el comportamiento del dispositivo cuando un periférico envía datos de forma continua y exhaustiva.
- **Todos le envían al mismo simultáneamente:** evalúa la capacidad de un periférico para recibir datos simultáneamente de todos los demás periféricos.

3. Aleatorización

Además de las pruebas descritas, se aleatorizarán distintas variables del test para evaluar el comportamiento del sistema bajo diferentes condiciones:

- **Número de transacciones:** se probarán distintas cantidades de transacciones para verificar el rendimiento y comportamiento del dispositivo.
- **Largo de paquetes:** se aleatorizará el largo de los paquetes entre 16, 32 y 64 bits.
- **Tiempo de espera entre eventos:** para las distintas transacciones se aleatorizarán los tiempos de retardo entre eventos.
- **Tipos de transacciones:** las transacciones pueden variar en orden y cantidad, cada una realizando diferentes acciones, lo que será aleatorizado.
- **Payload:** los datos enviados a los periféricos se aleatorizarán para probar diferentes contenidos.
- **Identificadores válidos:** cada periférico posee un ID, los cuales se asociarán a los periféricos presentes durante la prueba.
- **Identificadores no válidos:** como prueba adicional, se utilizarán identificadores asociados a periféricos que no existen para verificar el comportamiento del sistema ante situaciones no válidas.

4. Diseño del Ambiente

El ambiente consiste en varios transacciones, conformado por un generador de *test*, un agente, un *driver* y *monitor*, un *checker* y un *scoreboard*. Además, el *driver* incluye una interfaz para comunicarse con el DUT.

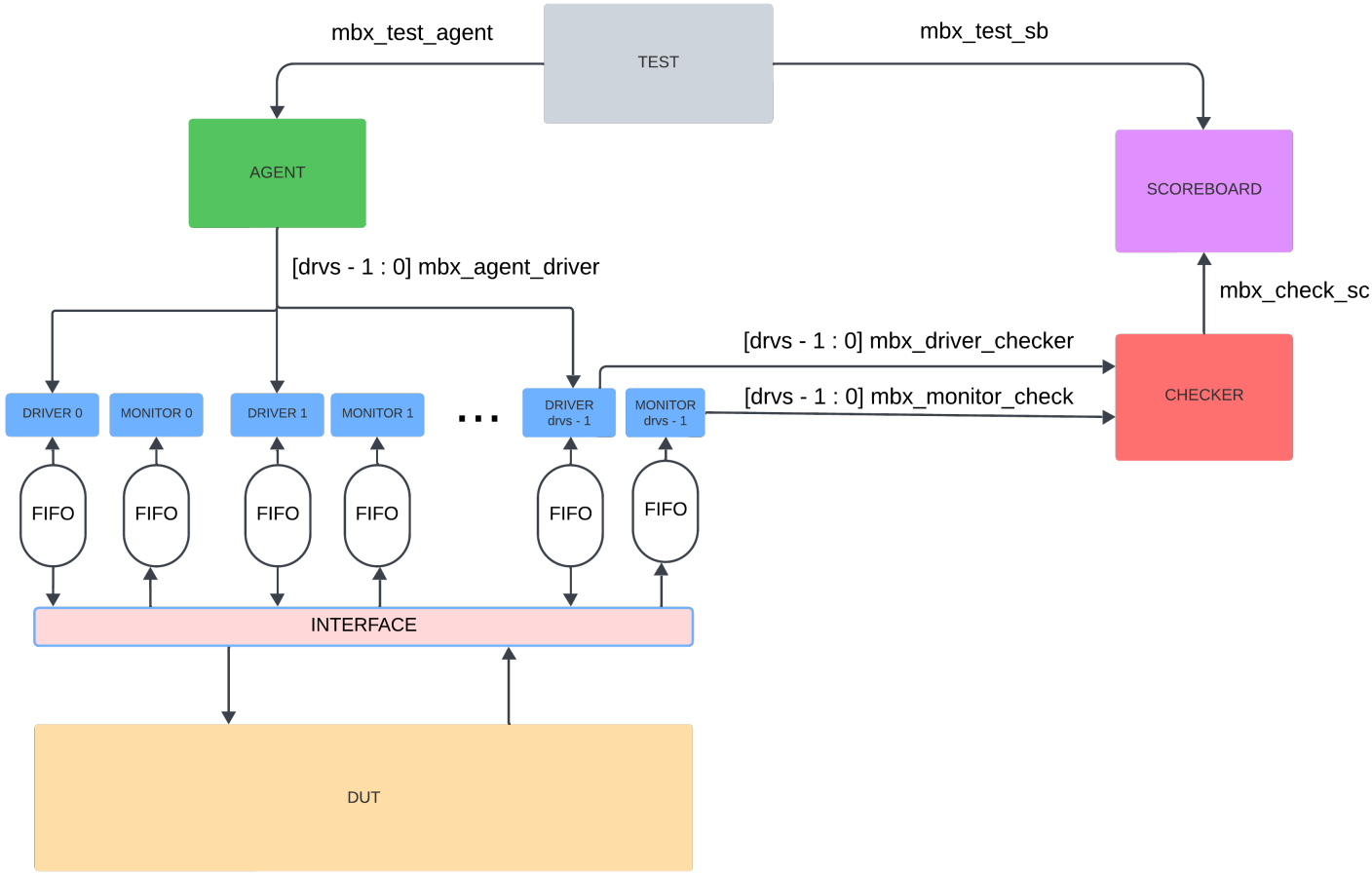


Figura 2: Diagrama del ambiente de verificación con el DUT

4.1. Interfaz del Driver con el DUT

La interfaz es el medio a través del cual el *driver* interactúa con el DUT. A través de esta interfaz, el *driver* envía los estímulos convertidos al DUT, y el DUT devuelve las respuestas correspondientes. La interfaz maneja las señales de bajo nivel necesarias para la comunicación adecuada entre ambos. En la siguiente imagen se detallan las señales de salida y entrada del DUT y su conexión a las FIFOs emuladas en el *Driver* y *Monitor*, esto se puede observar en la Figura 3.

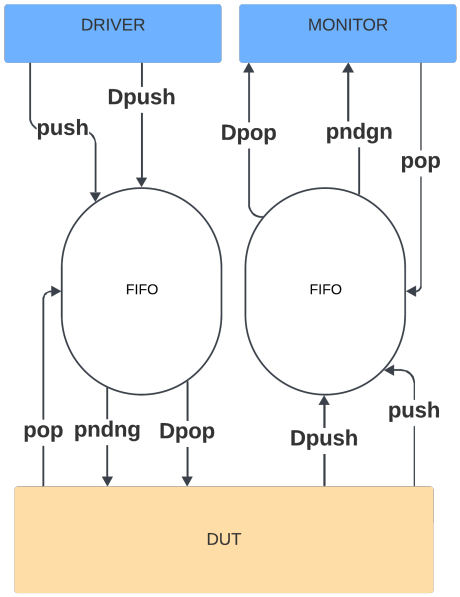


Figura 3: Interfase entre Driver y DUT

4.2. Generador de Tests

El generador es el bloque encargado de crear los estímulos o entradas que se enviarán al DUT. Estos estímulos pueden ser aleatorios o predefinidos, dependiendo del tipo de prueba. Este componente genera transacciones, que son estructuras de datos que representan las entradas para el DUT.

En este caso fue nombrado `secuencer` y tiene dos parámetros principales, `width` y `DRVS`, los cuales definen el ancho de los datos que serán utilizados en el DUT y el número de drivers involucrados en la prueba. Además, se definen otros parámetros, como `num_transacciones`, que indica el número de transacciones a realizar durante la prueba, y `max_retardo`, que establece el retardo máximo permitido entre transacciones.

El entorno de prueba se representa a través de la instancia `ambiente_inst`, la cual incluye los componentes esenciales como el driver, el agente y el *scoreboard*. Esta instancia permite que los diferentes módulos interactúen de forma coordinada. Adicionalmente, la interfaz virtual, `vif_test_fifo_dut` permite la comunicación con el DUT, encapsulando señales como el *reset* y otros parámetros de prueba para un mejor manejo con el DUT.

El generador de test utiliza dos *mailboxes*, `mbx_test_agent` y `mbx_test_sb`, para la comunicación entre el generador, el agente y el *scoreboard*. Estos *mailboxes* son fundamentales para enviar las instrucciones generadas por el secuenciador al agente, y para solicitar reportes al *scoreboard* una vez que las transacciones se han completado.

Una vez que el agente recibe las instrucciones, comienza a ejecutar transacciones hacia el DUT. Durante el proceso, el generador de test puede enviar instrucciones adicionales. Finalmente, cuando se alcanza el tiempo límite de la prueba, el generador envía una consulta al *scoreboard* para generar un reporte final y cierra el archivo CSV con los resultados.

4.2.1. Ambiente

Relacionado con lo anterior, el archivo de ambiente consiste en la conexión de los transactores del ambiente de verificación para el DUT.

Algo importante a mencionar es que se instanciaron en forma de stacks el driver, monitor y agente según la cantidad de periféricos.

Finalmente se realiza la inicialización del ambiente, la cual tiene como detalle importante la utilización de una variable automática, ya que al realizar el proceso de ejecución de las instancias en forma de stack se requiere que lo haga por cada valor en el `for` porque si fuera una variable estática utilizaría el último dato del `for` que quedó en memoria.

4.3. Agente

El agente es el transactor que recibe las instrucciones de alto nivel del generador test y las traduce a un nivel de abstracción menor para enviarlas a los *drivers*.

Este agente se comunica con el generador test y el driver a través de los mailboxes. Una vez que recibe una instrucción, el agente utiliza algún caso, que controlan el comportamiento aleatorio de las transacciones, como si se generarán IDs ilegales, resets o transacciones tipo broadcast. Es importante mencionar que para la aletorización de variables como los ID, se utilizó `$urandom`, la cual es una función similar a randomize.

Es importante mencionar que un caso del agente consiste en el proceso de inicialización de transacciones dentro del agente. Si el número de drivers disponibles (`DRVS`) es cero, el agente imprime un mensaje que indica que no se puede proceder con la inicialización, ya que solo hay un driver disponible. Caso contrario, el agente genera dos transacciones de manera aleatoria. Ambas transacciones se configuran con los siguientes parámetros: el campo `pkg_payload` se inicializa en 0, el identificador de paquete `pkg_id` se establece en broadcast 1.

Una vez configuradas, estas transacciones se envían a los drivers a través de las respectivas `mailboxes`. La primera transacción se dirige al último driver disponible (`agnt_drv_mbx[DRVS-1]`), mientras que la segunda se envía al primer driver (`agnt_drv_mbx[0]`), esto con el fin de que todos inicien con un valor 0.

4.4. Driver

El *driver* convierte las transacciones del agente en señales específicas que el DUT puede entender. Estas señales son enviadas al DUT a través de una interfaz, interactuando directamente con la lógica de entrada del DUT.

En este caso se utilizó un arreglo de colas `fifo_in`, diseñado para almacenar las transacciones de datos que el driver debe enviar al DUT. Su dimensión está definida por el parámetro `WIDTH`, que indica el número de bits en cada transacción. Al ser un arreglo dinámico, puede crecer o reducirse en tamaño según la cantidad de transacciones pendientes en un momento dado.

Cada vez que se recibe una nueva transacción, los datos se empaquetan y se colocan en el arreglo `fifo_in` mediante la operación `push_front`. Esto asegura que el driver esté preparado para enviar

los datos cuando el DUT esté listo para recibirlos. Mientras tanto, se verifica el tamaño de esta FIFO dinámica, controlando si existen transacciones pendientes por procesar, y se actualizan las señales de control que el DUT utiliza para solicitar nuevos datos.

Cuando el DUT indica que está listo para recibir, el driver extrae los datos del FIFO mediante `pop_back` y los envía al DUT. Este proceso garantiza que solo se envíen datos cuando el DUT los solicite, manteniendo un flujo sincronizado entre el driver y el DUT.

4.5. Monitor

El *monitor* observa las señales provenientes del DUT sin modificarlas. Su función es observar, reconstruir y enviar transacciones al *checker* para su posterior validación.

El funcionamiento del monitor comienza con su inicialización, donde se le asigna un identificador único denominado `mnt_id`. Este identificador permite que el monitor distinga y reciba únicamente las transacciones que le corresponden, facilitando así la coordinación entre múltiples monitores dentro del sistema.

Luego entra en un ciclo continuo de espera por nuevas transacciones, verifica constantemente la señal `push` del *DUT*, que indica cuándo el *DUT* ha preparado una transacción y está lista para ser procesada. Mientras esta señal esté inactiva (es decir, en valor 0), el monitor permanece en estado de espera. Sin embargo, cuando la señal se activa, el monitor detecta que hay datos disponibles y procede a capturarlos.

Tras recibir la señal de activación, el monitor extrae los componentes de la transacción desde el *DUT*, que incluyen el `pkg_id`, que identifica el paquete, y el `pkg_payload`, que representa el contenido de la transacción.

Una vez reconstruida la transacción, el monitor la envía al *checker* a través de un *mailbox* llamado `mnt_checker_mbx`. Este paso es esencial, ya que el *checker* se encarga de verificar que la transacción ha sido procesada correctamente por el sistema.

La variable `mnt_id` es esencial en este contexto, ya que asegura que el monitor únicamente procese las transacciones dirigidas específicamente a él, evitando así que se mezclen transacciones provenientes de otros monitores o dispositivos.

4.6. Checker

El `checker` es responsable de verificar la coincidencia de las transacciones entre los drivers y los monitores. Utiliza `mailboxes` para recibir transacciones desde los drivers y los monitores, además utiliza una FIFO para almacenar temporalmente las transacciones recibidas desde los drivers.

Cuando una transacción es recibida por el checker desde el `driver_checker_mbx`, esta transacción es almacenada en la `driver_fifo` mediante el método `push_front`. Este método agrega la transacción al frente de la cola, permitiendo que las transacciones se acumulen en el orden en que son recibidas.

Cuando se recibe una transacción de un monitor, el checker recorre la `driver_fifo` y compara cada transacción almacenada en la FIFO con la transacción del monitor. Si se encuentra una coincidencia

entre los campos pkg_id, pkg_payload, receiver_monitor, y tipo_transaccion, la transacción es considerada válida y se reconstruye para enviarla al scoreboard.

Además, la driver_fifo se revisa periódicamente mediante el método revisar_datos_descartados, que evalúa las transacciones pendientes. Si se detecta una transacción ilegal o sin propósito, como transacciones de tipo broadcast que ya han sido procesadas por todos los monitores o transacciones con un pkg_id no válido, se descartan. En cambio, si se encuentra una transacción válida que aún no ha sido evaluada, se genera un error en la simulación.

Si las transacciones coinciden, se reconstruye la transacción completa con información adicional (como los tiempos de envío y recepción) y se envía al scoreboard. Si no se encuentra coincidencia, se genera un error y se detiene la simulación.

4.7. Scoreboard

El *scoreboard* es una base de datos que contiene los resultados y reportes de las distintas pruebas, permitiendo hacer un seguimiento del desempeño del DUT durante el proceso de verificación.

4.8. Interfaces de Comunicación entre Bloques y Paquetes

Relacionado al diagrama de bloques del ambiente, los paquetes de transacción son los siguientes:

4.8.1. mbx_test_agent

Este *mailbox* comunica al test con el agente. El tipo de *mailbox* es paramétrico de datos instrucciones_agente, el cual es una estructura que contiene las siguientes características:

- **Tipos de secuencia:**
  - init
  - send_random_payload_legal_id
  - send_random_payload_ilegal_id
  - send_w_mid_reset
  - consecutive_send
  - *broadcast*_random
  - some_sending_random
  - some_*broadcast*
  - all_for_one
  - all_sending_random
  - all_*broadcast*
  - auto_send_random

4.8.2. Mailbox para las transacciones

Este es un arreglo de *mailboxes* de ancho `drvs` que comunica al agente con el *driver* de cada periférico según corresponda. Además, este mismo comunica a cada monitor de cada periférico con el *checker* y de cada driver al checker. Todos son del mismo tipo. Estos *mailboxes* contienen datos `instrucciones_driver_monitor`.

- **Datos:**
  - `max_delay`
  - `delay`
  - `pkg_id`
  - `send_time`
  - `receive_time`
  - `receiver_monitor`
  - `sender_driver`
  - `pkg_payload`
  - `tipo_transaccion`
- **Tipos de transacción:**
  - `send`
  - *`broadcast`*
  - `reset`
  - `receive`

4.8.3. mbx_check_sb

Este *mailbox* comunica al *checker* con el *scoreboard* a un nivel de abstracción más alto. El tipo de dato es `res_check`.

- **Tipos de secuencia:**
  - `send_time`
  - `receive_time`
  - `latency`
  - `data`
  - `id`
  - `result`

4.8.4. mbx_test_sb

Este *mailbox* comunica al *test* con el *scoreboard* para hacer las consultas de reportes. El tipo de dato es `consulta_sb`.

- **Tipo de reporte:**
  - Paquetes recibidos por terminal: pkg_rx
  - Paquetes enviados por terminal: pkg_tx
  - Retardo promedio por terminal: avg_delay
  - Retardo promedio total: total_avg_delay
  - Transacciones totales: total_transactions
  - Reporte de *broadcast*: broadcast_report
  - Reporte completo: total_report

5. Plan de Pruebas

5.1. Recursos generales

Para llevar a cabo el plan de pruebas, es necesario cumplir con los siguientes requisitos:

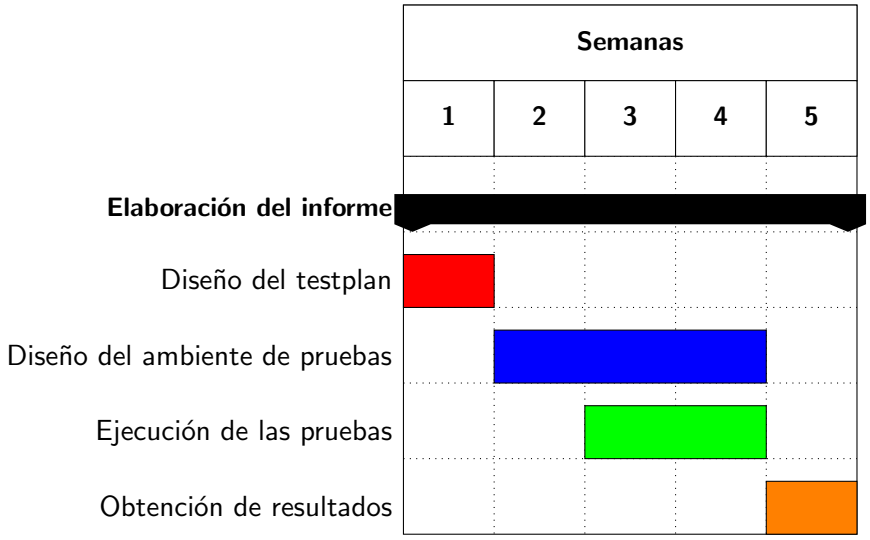
- Archivos RTL que contengan el DUT.
- FIFOs emuladas.
- Ambiente de verificación.
- Acceso al servidor con las herramientas de Synopsys, específicamente VCS y Verdi.
- Acceso al repositorio de git del proyecto.

Escenario	**Objetivo**	**Detalle**
DUT con drivers entre 8 y 16 unidades, con profundidades de FIFO infinitas y tamaños de palabra de 8 a 32 bits, todo aleatorizado, con entre 10 y 100 transacciones, cada paquete tendrá un ID fuera de los límites establecidos. Cada transacción tendrá un retardo de entre 1 y 10 ciclos y un driver que realizará la transacción.	Verificar que cuando el DUT recibe un paquete con un ID fuera de los límites de los drivers, **lo descarte** correctamente.	El ambiente debe aleatorizar el tamaño del DUT y de la palabra, con un payload aleatorio y un ID fuera de los límites. La cantidad de transacciones y retardos debe aleatorizarse, y cada transacción debe asignarse a un driver. Encardado de la prueba: Encargado de la prueba: Pedro Medinila

Escenario	Objetivo	Detalle
DUT con drivers entre 8 y 16 unidades, con entre 100 y 200 transacciones consecutivas enviadas desde un mismo driver, con un payload aleatorio y tiempos de transacción aleatorios de entre 1 y 10 ciclos.	Verificar que un mismo driver pueda enviar múltiples **transacciones consecutivas** sin perder datos, y que todos los datos lleguen a su destino correcto sin interferencias o pérdidas.	El ambiente debe aleatorizar la cantidad de drivers, la profundidad de las FIFOs y el tamaño de los paquetes. Debe establecer un número aleatorio de transacciones consecutivas desde un solo driver, donde cada transacción tenga su propio retardo y FIFO de origen. Encardado de la prueba: David Medina
DUT con drivers entre 8 y 16 unidades, con profundidades de FIFO infinitas y tamaños de palabra de 8 a 32 bits, todo aleatorizado, con entre 10 y 100 transacciones, cada palabra con un payload aleatorio y retardos de 1 a 10 ciclos. Tras el retardo, todos los drivers envían datos **a sí mismos**, manteniendo un ID constante.	Verificar que cuando un driver se envía un dato a sí mismo, no pase nada y se descarte correctamente bajo diferentes retardos.	El ambiente debe aleatorizar el tamaño de palabra y la cantidad de drivers. Las transacciones serán aleatorias, cada una con su retardo, y un vector de drivers realizará la transacción con un payload aleatorio. Encardado de la prueba: Encargado de la prueba: Pedro Medinila
DUT con drivers entre 8 y 16 unidades, con profundidades de FIFO infinitas y tamaños de palabra de 8 a 32 bits, todo aleatorizado, con entre 100 y 200 transacciones, cada palabra con un payload aleatorio y un ID de destino aleatorio. Cada transacción tendrá un retardo de entre 1 a 10 ciclos, tras el cual todos los drivers envían datos **al mismo tiempo** a un solo driver.	Verificar que un driver pueda recibir paquetes desde todos los demás drivers simultáneamente, y probar este escenario con cada driver.	El ambiente debe aleatorizar la cantidad de FIFOs, los tamaños de palabras, la cantidad de transacciones y el contenido del paquete, con IDs dentro de los límites de las FIFOs. El ambiente también debe aleatorizar el retardo en el que varias FIFOs envían datos al mismo tiempo y determinar cuáles envían los datos y cuál los recibe. Encardado de la prueba: Encargado de la prueba: Pedro Medinila

Escenario	**Objetivo**	**Detalle**
DUT con drivers entre 8 y 16 unidades y tamaños de palabra de 8, 16 y 32 bits (todos aleatorizados) y entre 100 y 500 transacciones, donde cada palabra y su payload son generados aleatoriamente según el tamaño, con IDs aleatorios que respetan el límite de los drivers, con **posibilidad** de *broadcast*, y tiempos de transacción entre 1 y 10 ciclos, todo aleatorio.	Verificar que los dispositivos conectados al DUT puedan intercambiar datos aleatoriamente desde cualquier driver, y que funcionen bajo diferentes condiciones: retardos, profundidades, tamaños de palabra, cantidad de FIFOs y número de transacciones. Además, asegurar que el *broadcast* desde cualquier driver sea recibido por todos los demás correctamente.	El ambiente debe poder aleatorizar la cantidad de drivers al inicio de cada prueba, así como la profundidad de las FIFOs y el tamaño de los paquetes. Además, debe generar un número aleatorio de transacciones, cada una con su propio retardo y FIFO de origen. El *broadcast* debe generarse aleatoriamente en algún punto. Encardado de la prueba: David Medina
DUT con drivers entre 8 y 16 unidades y tamaños de palabra de 8 a 32 bits, todos aleatorizados, y con entre 100 y 200 transacciones, cada una con payload aleatorio y un ID de *broadcast* **constante**. Los tiempos de transacción varían de 1 a 10 ciclos y el orden de las transacciones es aleatorio.	Verificar que, cuando cualquier driver realiza un *broadcast*, todos los demás dispositivos reciban los datos, excepto el driver emisor.	El ambiente debe aleatorizar la cantidad de drivers, la profundidad de las FIFOs y el tamaño de los paquetes al inicio de la prueba. Además, debe generar transacciones aleatorias, cada una con un retardo asociado y una FIFO de origen. En este caso, el *broadcast* será constante y su ID fijo durante la prueba para verificar su correcto funcionamiento. Encardado de la prueba: Encargado de la prueba: Pedro Medinila
DUT con drivers entre 8 y 16 unidades, con profundidades de FIFO infinitas y tamaños de palabra de 8 a 32 bits, todo aleatorizado, con entre 10 y 100 transacciones, cada palabra con un payload aleatorio y retardos de 1 a 10 ciclos. Tras el retardo, todos los drivers realizan **broadcast simultáneamente**, manteniendo un ID constante definido en la compilación. Un vector de drivers seleccionará cuáles harán *broadcast*.	Verificar que cuando varios drivers realizan *broadcast* simultáneamente, todos los demás dispositivos reciban los datos correctamente, excepto los emisores.	El ambiente debe aleatorizar el tamaño de palabra, la cantidad de FIFOs, el número de transacciones y los retardos. También debe generar un vector que indique qué drivers realizan el *broadcast*, sin que necesariamente sea al mismo tiempo. Encardado de la prueba: David Medina

6. Cronograma de actividades



7. Implementación y Resultados

Los archivos de simulación y código fuente del proyecto se encuentran en el repositorio.

```
git clone https://github.com/DJosueMM/bus_driver_verification.git
```

Al ir a la carpeta de simulación en: **src/sim/**

Se ejecuta el comando:

```
source vcs.sh
```

Esto simulará el test planteado, para observar los resultados se ejecuta:

```
./salida
```

Se puede apreciar que en los primeros instantes se inicializa el ambiente, cada evento es registrado.

Al final, se imprime un reporte general y se le pide al checker que verifique que todos los datos hayan sido correctamente evaluados o descartados según corresponda.

Se crea un archivo .csv en la misma carpeta de simulación que contiene datos del scoreboard, algunos de estos fueron graficados en GNU Plot y a continuacíon se muestran los resultados.

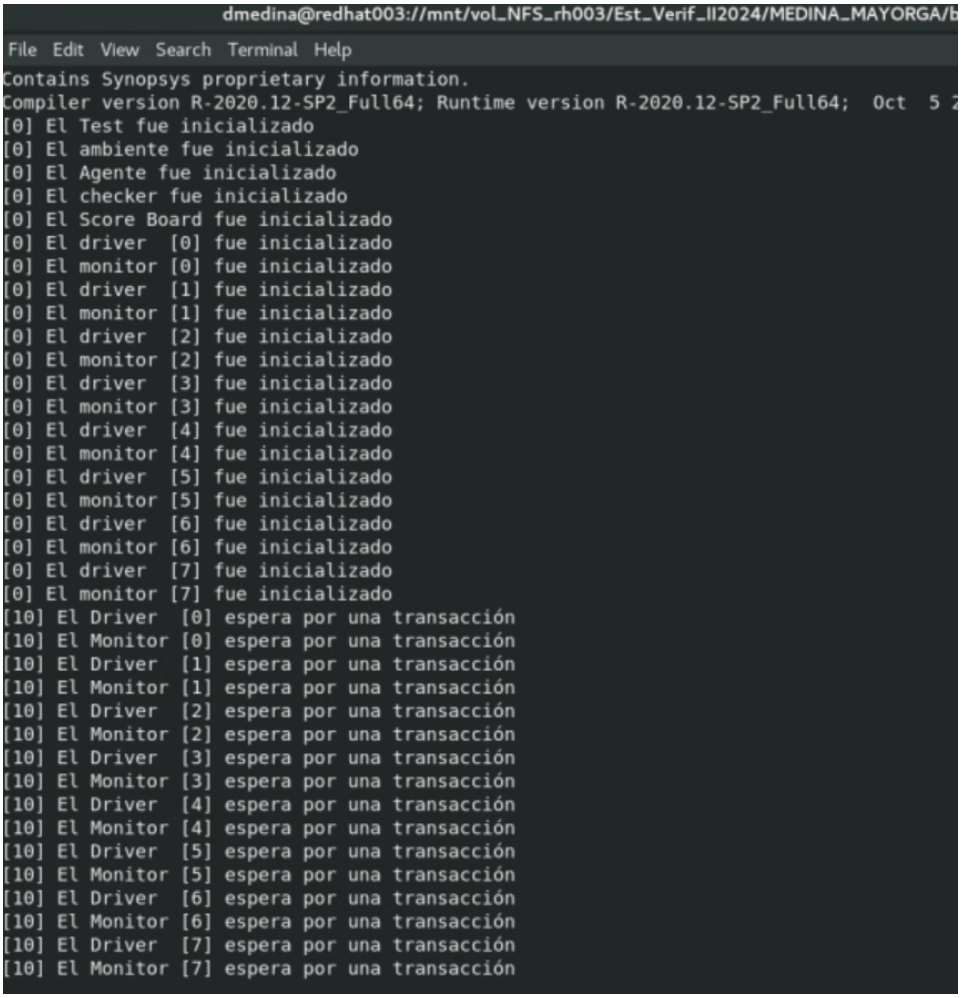


Figura 4: Log de la inicialización del ambiente

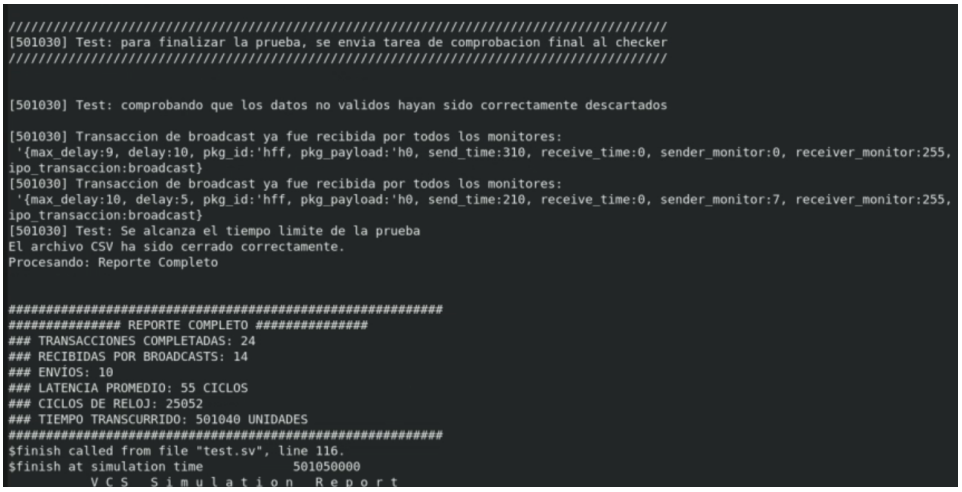


Figura 5: Log del reporte del Scoreboard después de realizar la simulación

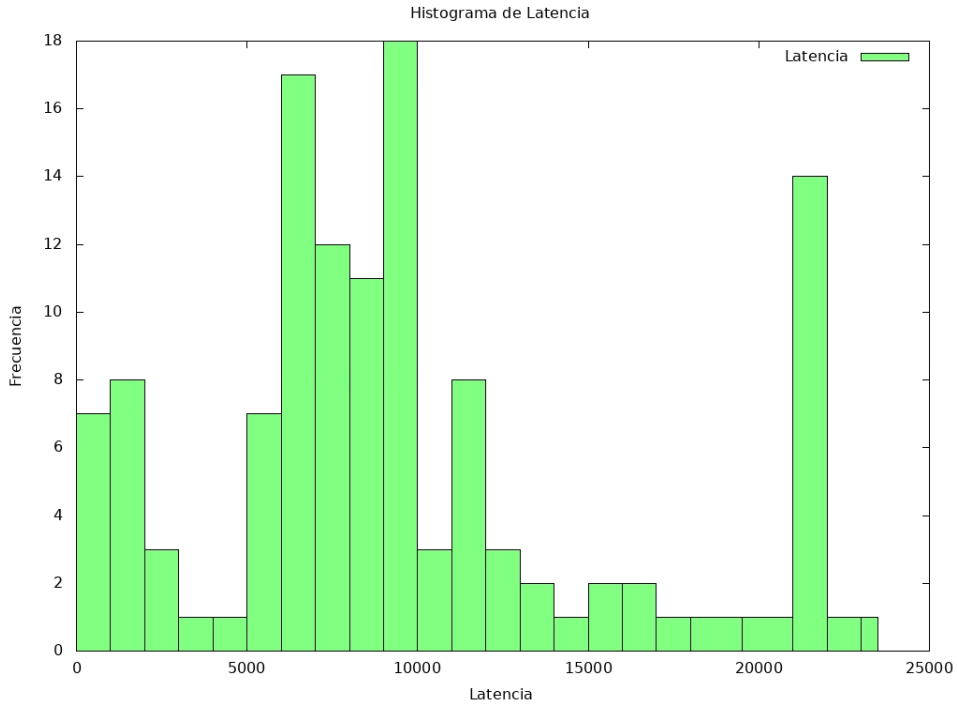


Figura 6: Tiempos de retardo de los paquetes recibidos para varios casos de prueba

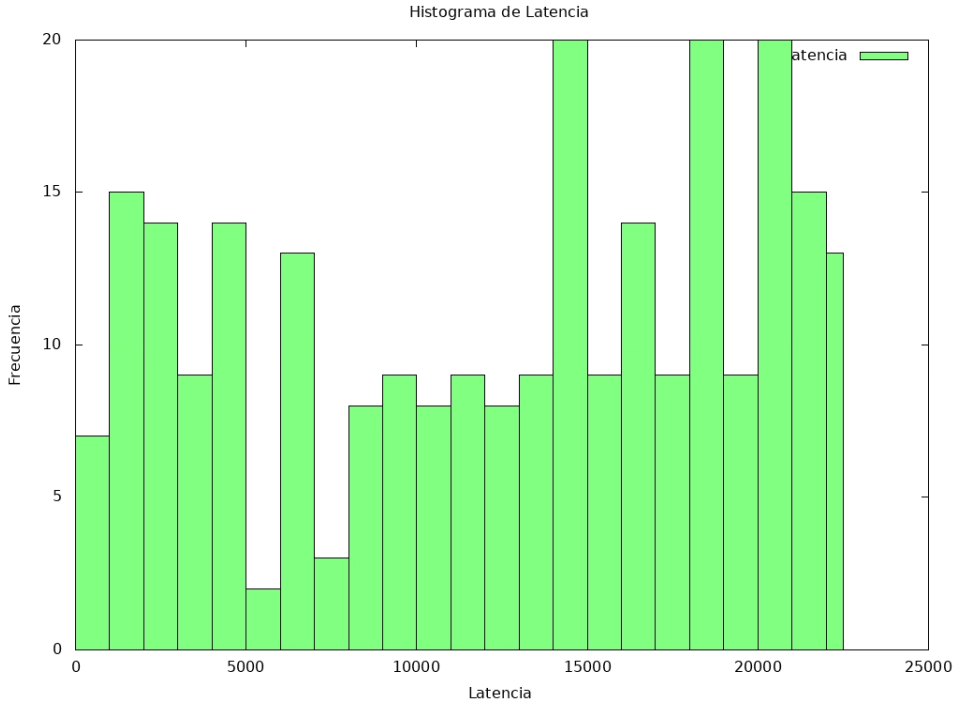


Figura 7: Tiempos de retardo de los paquetes recibidos para casos de broadcast

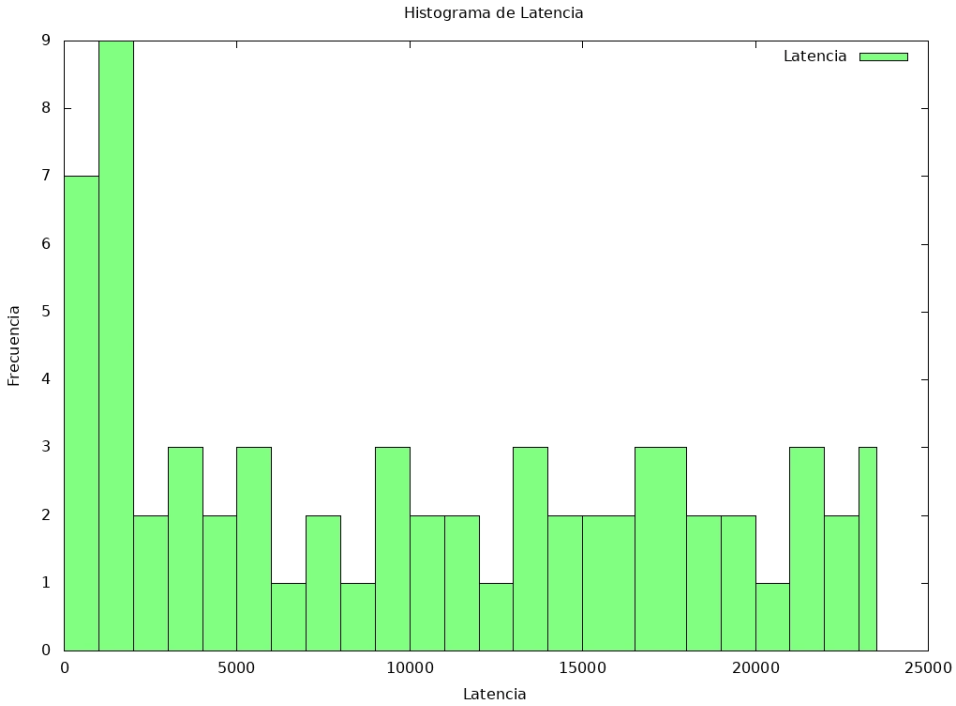


Figura 8: Tiempos de retardo de los paquetes recibidos para casos sin broadcast

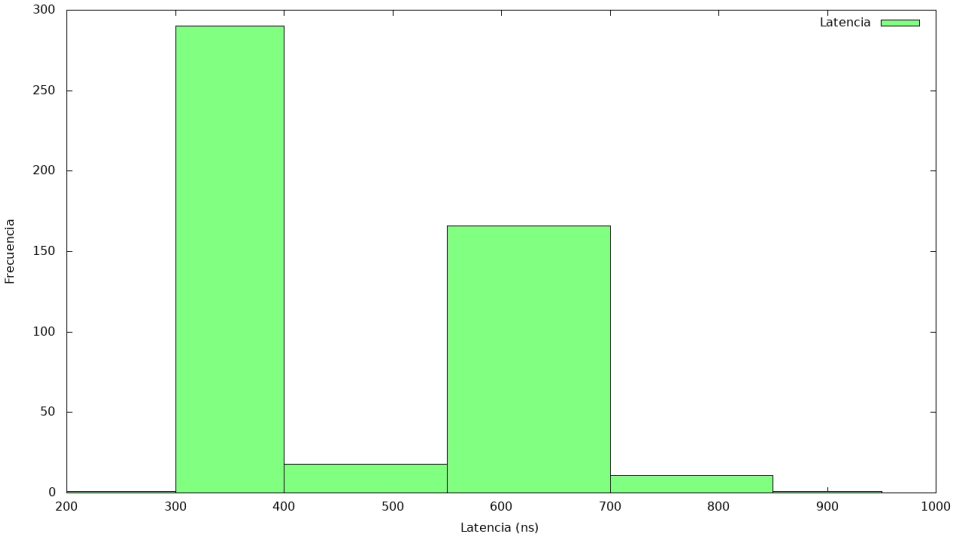


Figura 9: Tiempos de retardo de los paquetes recibidos con paquetes de 16 bits

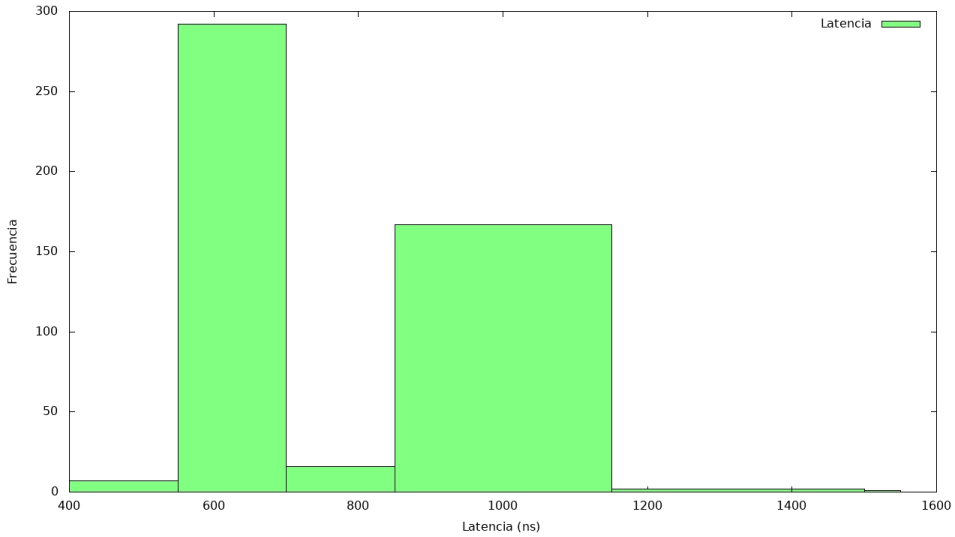


Figura 10: Tiempos de retardo de los paquetes recibidos con paquetes de 32 bits

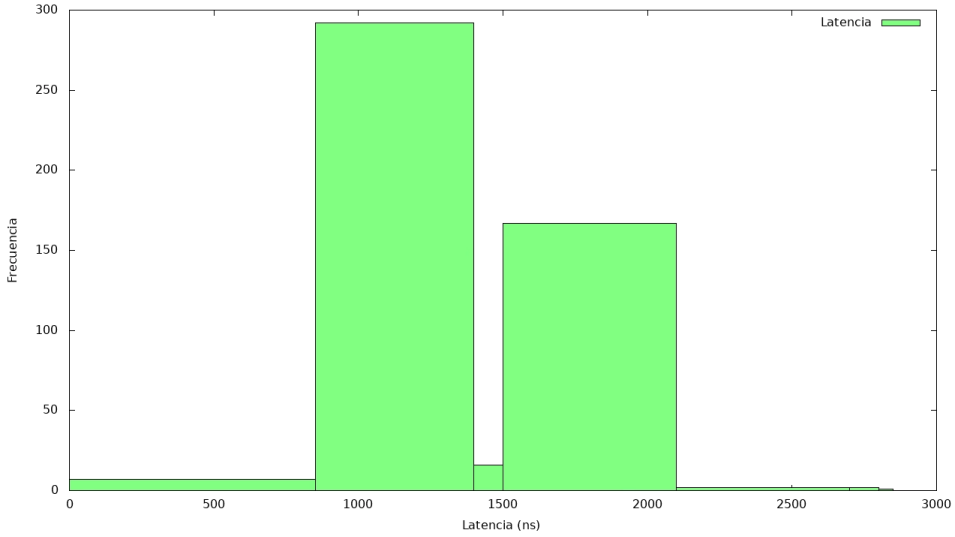


Figura 11: Tiempos de retardo de los paquetes recibidos con paquetes de 64 bits

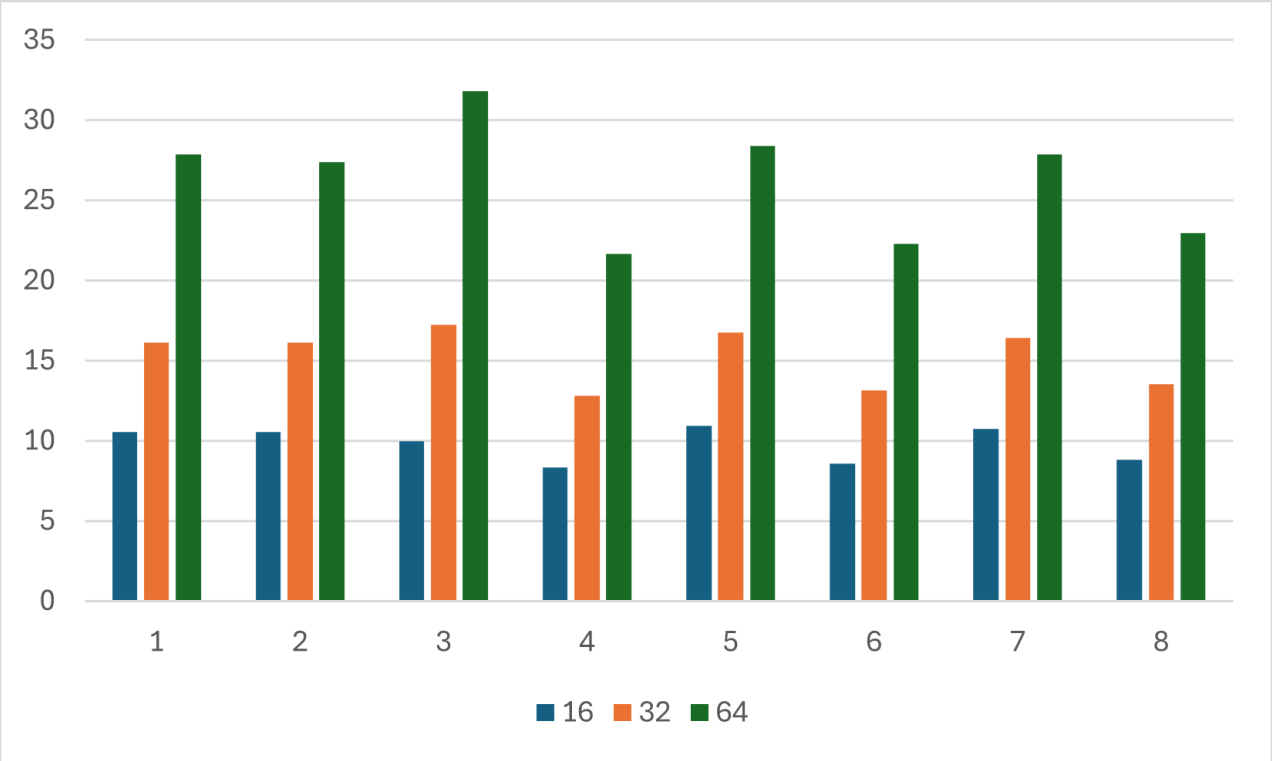


Figura 12: Latencia promedio por terminal para diferentes tamaños de paquetes en bits