ChatGPT

Diseño de un Esclavo SPI Modo 0 en Verilog – Explicación del Proyecto

1. Funcionamiento de un esclavo SPI Modo 0 (CPOL=0, CPHA=0)

El protocolo **SPI** (Serial Peripheral Interface) es una interfaz síncrona full-dúplex maestro-esclavo muy utilizada para comunicar microcontroladores con periféricos. Un sistema SPI de 4 hilos define cuatro señales principales: reloj de serie (`SCLK`), selección de chip (`SS`, típicamente activa en bajo), datos del maestro al esclavo (`MOSI`) y datos del esclavo al maestro (`MISO`). El maestro (dispositivo que genera `SCLK` y controla `SS`) inicia la comunicación poniendo `SS` en bajo para habilitar al esclavo seleccionado. Durante la comunicación SPI, en cada ciclo de reloj el maestro y el esclavo pueden **simultáneamente enviar y recibir bits** (full-dúplex) a través de MOSI y MISO. Sin embargo, dependiendo de la operación deseada (lectura o escritura), uno de los dos flujos de datos puede ser "vacío" o sin información útil (por ejemplo, en una escritura el maestro envía datos válidos por MOSI mientras MISO puede llevar datos irrelevantes, y en lectura ocurre al revés).

El **modo SPI 0** se caracteriza por tener la polaridad y fase de reloj CPOL=0 y CPHA=0. Esto significa que el reloj `SCLK` se mantiene en nivel bajo cuando está inactivo (CPOL=0, **idle low**) y que los datos se muestrean en el **flanco ascendente** (de subida) del reloj y se cambian en el **flanco descendente** (de bajada). En otras palabras, el receptor captura la línea de datos en cada flanco de subida de `SCLK` y el transmisor (quien envía el bit en ese instante) coloca el siguiente bit en la línea en el flanco de bajada. En el contexto de un esclavo SPI modo 0:

- El esclavo **lee/captura** el bit entrante de `MOSI` en cada flanco positivo de `SCLK` (cuando `SCLK` pasa de 0 a 1).
- El esclavo **actualiza** (prepara) su salida `MISO` en cada flanco negativo de `SCLK` (de 1 a 0), de modo que el nuevo valor esté listo antes del siguiente flanco positivo.

La sincronización así garantizada asegura que el maestro, que típicamente lee `MISO` en el mismo flanco de subida, obtenga un dato estable, y que el esclavo capture `MOSI` correctamente. Esta configuración de fase (CPHA=0) implica que el primer flanco activo tras habilitar el esclavo (bajar `SS`) ya es un flanco de subida válido para muestreo; por tanto, el maestro debe asegurar que el primer bit en MOSI esté válido **antes** de ese primer flanco de subida.

En SPI suele definirse también el orden de bits. Comúnmente se utiliza **MSB-first**, es decir, el bit más significativo se transmite primero por la línea serial. En nuestro diseño se ha adoptado MSB-first, lo que significa que el primer bit serial que el maestro envía (tras bajar `SS`) corresponde al bit más alto de la trama de datos, y el esclavo también responderá enviando primero el bit más significativo de su dato. Ambos maestro y esclavo deben acordar este orden; en modo MSB-first el registro de desplazamiento del esclavo irá incorporando los bits recibidos desplazándolos de manera que el primer bit recibido termine ocupando la posición más significativa del registro.

Adicionalmente, la señal `SS` (o `CS` – Chip Select) controla cuándo el esclavo está activo en el bus. `SS` es **activa en nivel bajo**; cuando está en alto, el esclavo debe ignorar cambios en `SCLK` y `MOSI` y colocar `MISO` en alta impedancia (`Z`) para no interferir en el bus. Solo mientras `SS` está baja, el

esclavo considera válidos los flancos de reloj para desplazar bits. Al final de una transacción, cuando el maestro sube `SS` de nuevo a 1, el esclavo típicamente reinicia su lógica de recepción para prepararse para la siguiente trama.

Resumen del ciclo de bits en modo 0: Justo antes de un flanco ascendente de `SCLK`, el transmisor habrá colocado un bit estable en la línea de datos. En el flanco ascendente, el receptor captura ese bit. Luego, en el flanco descendente siguiente, el dispositivo que estaba recibiendo puede ahora ser transmisor: actualiza la línea con el siguiente bit (en el caso del maestro en MOSI, en el caso del esclavo en MISO). De esta manera, para **cada ciclo de reloj** completo (flanco subida + bajada), se transfiere un bit en cada dirección, escalonados en el tiempo. Tras completar el número requerido de bits (por ejemplo 16 bits para una trama de 16 bits), el maestro suele desactivar `SS` (volverlo a 1) indicando fin de la trama. El esclavo en ese momento puede procesar los bits recibidos como un comando completo.

2. Módulo `SPI_slave.v` – Implementación del esclavo SPI en Verilog

El archivo `SPI_slave.v` contiene el diseño del módulo Verilog que implementa el esclavo SPI en modo 0. A continuación, se detalla su funcionamiento línea por línea, destacando el registro de desplazamiento para entrada/salida serial, la decodificación de la trama recibida (16 bits divididos en campo de control y datos) y la generación de señales de control como `write_enable`. Este módulo es parametrizable en cuanto al tamaño de la trama y campos, aunque en nuestro caso se instanciará con 16 bits totales (1 bit de R/W, 7 bits de dirección y 8 bits de datos).

2.1 Parámetros, puertos y señales internas

Las primeras líneas del módulo definen parámetros genéricos para permitir flexibilidad en el tamaño de la trama SPI y en la posición de cada campo. En nuestro caso, se usan: `FRAME_BITS=16` bits por trama, `ADDR_BITS=7` bits de dirección, `DATA_BITS=8` bits de datos. También se definen parámetros para la posición de cada campo dentro de la trama de 16 bits: el bit de R/W es el bit 15 (más significativo), la dirección ocupa los bits 14 a 8, y los datos los bits 7 a 0. Esto permite extraer fácilmente cada parte una vez recibida la trama completa.

Además, `RESP_START_BIT=7` indica el índice del bit tras cuyo recibo se puede preparar una respuesta de lectura. En este diseño, tras recibir el bit de índice 7 (es decir, cuando se han recibido los primeros 8 bits: R/W + dirección), el esclavo ya sabe si el comando es de lectura o escritura y qué dirección se accede, por lo que en caso de lectura puede **cargar el dato de respuesta** antes de que lleguen los bits dummy del maestro. Igualmente, se define `MSB_FIRST=1` para operar con bit más significativo primero (MSB-first).

Las señales de entrada/salida del módulo incluyen las líneas SPI estándar: `ss_n` (chip select, su "_n" indica activo bajo), `sclk` (reloj serial), `mosi` (entrada de datos del maestro) y `miso` (salida de datos del esclavo). También tiene conexiones hacia el sistema interno del FPGA: `addr_out`, `data_out` y `write_enable` son salidas que entregan la dirección, el dato recibido y un pulso de escritura cuando llega una trama de escritura; `data_in` es una entrada que debe proveer el dato a enviar por MISO en caso de lectura; `done` es una salida indicativa de fin de trama; y `rx_frame` es una salida opcional que contiene la trama completa recibida (16 bits).

Internamente, el esclavo declara **registros de desplazamiento** (`rx_shift` para recepción y `tx_shift` para transmisión) del ancho de la trama (16 bits). También se implementa un contador de

bits `bit_cnt` para llevar la cuenta de cuántos bits de la trama se han transferido. El ancho de este contador (`CNT_W`) se calcula con una función `clog2` para acomodar el número de bits de la trama. Por ejemplo, para 16 bits `CNT_W` será 4 bits (ya que log2(16)=4).

Se definen constantes locales `LAST_BIT = FRAME_BITS-1` (en nuestro caso 15, índice del último bit) y `RESP_LATCH = RESP_START_BIT` (7 por defecto) y `RESP_LOAD = RESP_START_BIT+1` (8). Estas se usan para identificar momentos clave durante la recepción: `RESP_LATCH=7` indica el ciclo en que se terminan de recibir los primeros 8 bits (cuando `bit_cnt == 7`), y `RESP_LOAD=8` indica el ciclo inmediatamente siguiente, cuando empieza la segunda mitad de la trama (bit 8) en la que, si es lectura, debemos cargar el dato de respuesta.

También hay registros auxiliares `rw_latched` y `addr_latched` para retener el valor del bit R/W y la dirección tras recibirlos. Esto es útil porque el proceso de transmisión necesitará saber *antes de terminar la trama* si debe cargar un dato para enviar (lectura) o no (escritura).

Finalmente, la salida `miso` se asigna combinacionalmente al bit más significativo o menos significativo de `tx_shift` dependiendo de `MSB_FIRST`. Dado que trabajamos MSB-first, `tx_bit` toma el bit más alto de `tx_shift`. La salida `miso` solo se activa (no tri-estado) cuando el esclavo está seleccionado (`!ss_n` activo); de lo contrario, si `ss_n` está alto, `miso` se coloca en alta impedancia (`1'bz`). Esto previene conflictos en el bus MISO cuando el esclavo no está activo.

2.2 Reinicio del esclavo y reset por deselección (`SS` alto)

El comportamiento asíncrono del esclavo ante un reset o al desactivar el chip select se maneja con un bloque `always @(posedge ss_n or negedge rst_n)`. Aquí se aprovecha el flanco de subida de `ss_n` (cuando el maestro pone `SS` en alto) para **reinicializar el esclavo al término de cada transacción**, similar a lo que haría un reset. Si ocurre un reset global (`rst_n` baja), también se reinician todos los registros internos. Dentro de este bloque se limpia `rx_shift`, `tx_shift`, `bit_cnt`, `rw_latched`, `addr_latched`, y las salidas `addr_out`, `data_out`, `write_enable`, `done`, `rx_frame` se ponen a 0. En la rama del else (cuando `ss_n` sube, es decir fin de trama) se hace prácticamente lo mismo: se resetean los registros de desplazamiento, contador y señales de control excepto que en este caso no se toca `rx_frame` (permanece con la última trama recibida para posibles usos de nivel superior). En resumen, cada vez que `SS` pasa de activo a inactivo, el esclavo se prepara limpiamente para no retener datos viejos en una siguiente operación.

2.3 Recepción de bits (`rx_shift`) en flanco ascendente de `SCLK`

El bloque principal de **recepción** es `always @(posedge sclk or negedge rst_n)`. Aquí, en cada flanco de subida de `sclk`, si el sistema no está en reset y el esclavo está activo (`!ss_n`), se desplaza el registro de entrada `rx_shift` incorporando el nuevo bit de `MOSI`. El código distingue según el orden de bits configurado:

- En MSB-first (`MSB_FIRST=1`): el bit recién leído de `mosi` se coloca en la posición menos significativa de `rx_shift`, y todo el registro se corre a la izquierda (hacia bits más significativos). Esto se logra con `next_rx = {rx_shift[FRAME_BITS-2:0], mosi}`: concatena los bits previos de `rx_shift` (quitando el MSB) con el nuevo bit en la posición LSB.
- (En caso LSB-first, sería al revés: desplazar a la derecha y colocar el nuevo bit en el MSB, con la rama else correspondiente en el código).

Luego `rx_shift <= next_rx` almacena el nuevo valor desplazado. Efectivamente, después de 16 flancos de subida, `rx_shift` contendrá la trama completa recibida. Cabe notar que al desplazar de esta manera en modo MSB-first, el primer bit recibido (bit15) "viaja" hacia la izquierda en el registro a medida que entran más bits, de modo que al final quedará en la posición de bit 15 del registro, preservando el orden correcto de la trama.

A continuación, el código verifica si se ha llegado a ciertos conteos de bits importantes:

- **Latcheo intermedio (bit 7 recibido):** Si `bit_cnt == RESP_LATCH` (7), significa que acabamos de recibir el bit de índice 7, es decir, los primeros 8 bits de la trama ya están completos (R/W + dirección). En ese momento se **latchean** los valores de control: `rw_latched <= next_rx[RW_BIT]` y `addr_latched <= next_rx[ADDR_MSB:ADDR_LSB]`. Esto guarda el bit de lectura/escritura y la dirección recibida hasta ahora. `next_rx` en este instante ya incluye el bit recién capturado, por lo que `next_rx[15]` contiene el R/W y `next_rx[14:8]` la dirección. Guardarlos ahora es útil para preparar la respuesta si es lectura.
- **Fin de trama (bit 15 recibido):** Si `bit_cnt == LAST_BIT` (15, el índice del último bit de la trama de 16), significa que el bit actual que acabamos de recibir completa la trama. En ese caso se ejecuta la lógica de fin de recepción:
- Se hace `frame_full = next_rx`, así se guarda la trama completa recibida en un registro temporal. Luego `rx_frame <= frame_full` para exponer la trama completa al exterior (por la salida `rx_frame`).
- Se extraen los campos de dirección y datos de esa trama: `addr_out <= frame_full[ADDR_MSB:ADDR_LSB]` y `data_out <= frame_full[DATA_MSB:DATA_LSB]`. Así, `addr_out` contendrá la dirección de registro destino y `data_out` contendrá el byte de datos que envió el maestro (en caso de comando de escritura) o el "dummy" enviado (en caso de lectura).
- Si el bit R/W de la trama es 1 (lo que indicaría una **escritura**, según convención que veremos), entonces se activa la señal `write_enable <= 1'b1`. En nuestro diseño se ha definido que R/W = 1 significa *Write* (escritura) y R/W = 0 significa *Read* (lectura). Por lo tanto, `write_enable` se pone en 1 justo al finalizar una trama si dicha trama era una orden de escritura. Esta señal durará apenas un ciclo de reloj (se limpia al siguiente flanco de `sclk` o al desactivar `SS`), sirviendo típicamente como un **pulso de escritura** para actualizar un registro de memoria con `data_out`.
- Se activa la señal de `done <= 1'b1` para indicar que la trama completa fue recibida. Esta señal también es pulsante de un ciclo (se baja posteriormente), y podría usarse en lógica superior para detectar que llegó un comando completo.
- Finalmente, se reinicia el contador de bits `bit_cnt <= 0` para prepararse a una posible siguiente trama continua. (En la práctica, lo usual es que `SS` se desactive después de cada 16 bits y en ese momento también se resetea `bit_cnt` en el otro bloque, pero este reset adicional cubre el caso de tramas consecutivas con `SS` mantenido activo, aunque eso no es habitual sin relajar protocolo).

Si no estamos en el último bit, entonces simplemente se incrementa el contador `bit_cnt <= bit_cnt + 1` para la siguiente iteración.

En este bloque de recepción también se asegura que en cada nuevo flanco de reloj se limpien `write_enable` y `done` de vuelta a 0 (excepto en el caso especial del último bit donde se setean a 1). Esto garantiza que esas señales sean pulsos breves y evita que queden pegadas más de lo debido.

2.4 Transmisión de bits (`tx_shift`) en flanco descendente de `SCLK`

El bloque `always @(negedge sclk or negedge rst_n)` implementa la lógica de **transmisión** de datos por `MISO`. En cada flanco de bajada del reloj, si el esclavo está activo (`!ss_n`), el módulo determina qué bit debe ponerse en `MISO` a continuación preparando el registro `tx_shift`. Recordemos que el bit efectivamente puesto en la línea `MISO` viene de `tx_shift` vía `tx_bit` (bit más significativo en MSB-first), así que gestionar `tx_shift` adecuadamente es crucial.

Dentro de este bloque, la operación difiere según el valor del contador de bits `bit_cnt` (que indica qué número de bit *se va a enviar o se envió recientemente*):

- **Inicio de la trama (`bit_cnt == 0`):** Si el contador está en 0 en un flanco descendente, significa que estamos justo antes de enviar/recibir el primer bit o que acabamos de reiniciar la cuenta. En tal caso el código carga `tx_shift <= {FRAME_BITS{1'b0}}`, es decir, pone el registro de transmisión a 0 completo. Esto asegura que antes de que se haya identificado si la operación es lectura o escritura, la salida MISO estará enviando 0s. En efecto, durante los primeros 8 bits (cabecera) de cualquier transacción, el esclavo no tiene datos útiles que transmitir, así que simplemente mantendrá MISO en 0 (se podría también dejar en high-impedance si no seleccionado, pero estando seleccionado se suele enviar algo definido, a menudo 0).
- **Preparación de respuesta de lectura (`bit_cnt == RESP_LOAD`, que equivale a 8):** Cuando el contador llega a 8 en un flanco de bajada, significa que justo antes, en el flanco de subida anterior, se recibió el bit de índice 7 y se incrementó el contador a 8. Es en este **primer flanco de bajada inmediatamente después de recibir la cabecera** cuando se decide si se va a responder con datos (lectura) o no (escritura). El código comprueba `if (!rw_latched)`:
- Si `rw_latched == 0` implica que el bit R/W recibido era 0, lo que hemos definido como operación de **lectura**. Por lo tanto, el esclavo debe **cargar el dato de lectura en el registro de transmisión** para enviarlo de vuelta al maestro en los próximos 8 bits. Se toma el valor de `data_in` (provisto por el entorno, típicamente leído del banco de registros en `top_spi`) y se coloca en `tx_shift`. En MSB-first, ese byte de `data_in` se ubica en los bits más significativos de `tx_shift`. La concatenación `{ data_in, {(FRAME_BITS-DATA_BITS){1'b0}} }` significa que pone `data_in` (8 bits) en los bits [15:8] de `tx_shift` y llena con 0 los [7:0]. De este modo, el bit más significativo de `data_in` estará en la posición 15 de `tx_shift`, que es el próximo bit que saldrá por MISO (recordemos que `tx_bit` toma `tx_shift[15]`).
- Si `rw_latched == 1` implica una operación de **escritura**, o sea que los últimos 8 bits de la trama son datos entrantes del maestro. En este caso el esclavo **no tiene que enviar ningún dato significativo** de vuelta. El código maneja esto simplemente cargando ceros en `tx_shift` (ya sea dejándolo en su valor actual que ya era 0, o asignando explícitamente `{FRAME_BITS{1'b0}}` nuevamente). Aquí efectivamente no cambia nada porque `tx_shift` ya estaba en 0s desde el inicio, pero se incluye por claridad.
- **Ciclos intermedios (otros flancos de bajada):** En los flancos de bajada donde `bit_cnt` no es 0 ni 8, el código entra en el `else` final. Esto corresponde a todos los casos en que ya se cargó el `tx_shift` apropiadamente y debemos seguir **desplazándolo** para que los bits salgan secuencialmente. En MSB-first, el comportamiento deseado es desplazar a la izquierda (sacar el bit actual por el MSB y preparar el siguiente bit en esa posición). Así, se hace `tx_shift <= {tx_shift[FRAME_BITS-2:0], 1'b0}`, que desplaza a la izquierda metiendo un 0 en LSB. Cada flanco descendente eliminará el bit ya transmitido (antes en MSB) y moverá el siguiente bit hacia el MSB para el próximo ciclo. En LSB-first sería la lógica inversa (desplazar a la derecha).

El bloque de transmisión también reinicia `tx_shift` en reset (`rst_n` bajo) poniendo todos los bits en 0, y se asegura de cargar 0 al inicio de trama. Combinado con el bloque de recepción, esta lógica de

transmisión garantiza que si el maestro realiza una operación de lectura, el esclavo tendrá colocado en `MISO` los bits del dato correcto en los momentos adecuados: el primer bit significativo del dato de lectura se pone en `MISO` justo a tiempo para el flanco de subida del bit 8 (recordemos que la carga ocurrió en el flanco de bajada previo, index 8, así que en el siguiente flanco de subida – bit index 8 – el maestro lee el MSB del dato de `data_in` a través de MISO). Luego en cada siguiente bit, el registro se desplaza sacando bit por bit del dato hasta completar los 8 bits de respuesta.

Para una **escritura**, `tx_shift` se mantiene en cero, por lo que el esclavo esencialmente envía 0x00 durante el segundo byte, mientras recibe los datos del maestro por MOSI.

2.5 Decodificación de la trama (1 bit R/W, 7 bits dirección, 8 bits datos)

Una vez completada la recepción de 16 bits, el módulo SPI_slave entrega los campos decodificados. Como vimos, `frame_full[15]` es el bit R/W, `frame_full[14:8]` son los 7 bits de dirección, y `frame_full[7:0]` son los 8 bits de datos. La convención en este diseño es: **R/W = 1 indica escritura**, **R/W = 0 indica lectura**. Esto se deduce del comportamiento: cuando R/W=1, el esclavo activa `write_enable` para escribir `data_out` en la memoria, y no carga ningún dato en `tx_shift` (porque el maestro estaba enviando datos, no esperando recibirlos). En cambio, cuando R/W=0, el esclavo no activa `write_enable` (no escribe nada en memoria en ese ciclo) sino que carga el dato de `data_in` en `tx_shift` para transmitirlo de vuelta – justamente el comportamiento de una lectura.

Por lo tanto, nuestro **formato de trama** queda definido así:
- Bit 15: R/W (0 = Read, 1 = Write).
- Bits 14-8: Dirección del registro al que se desea leer o escribir (7 bits permiten 128 direcciones, 0x00 a 0x7F).
- Bits 7-0: Datos a escribir (si R/W=1) o valor dummy (si R/W=0, este byte se ignora y será reemplazado por la respuesta del esclavo simultáneamente).

El módulo entrega hacia arriba: `addr_out` = dirección de 7 bits recibida, `data_out` = byte de datos recibido (válido solo si fue comando de escritura), y `write_enable` = 1 en caso de comando de escritura completado. Si fue comando de lectura, en lugar de `write_enable` activo, lo que ocurre es que el esclavo habrá tomado `data_in` (la supuesta lectura del registro solicitado) y lo habrá enviado por MISO durante la segunda mitad de la trama.

Adicionalmente, la señal `done` indica que **la trama completa (los 16 bits)** ha sido recibida y procesada. Esto puede ser útil para fines de sincronización, por ejemplo para que la lógica superior sepa que puede tomar `addr_out` / `data_out` o que la respuesta en una lectura ha concluido.

3. Módulo `top_spi.v` – Integración con el banco de registros

El archivo `top_spi.v` define el módulo de nivel superior que instancia al esclavo SPI (`spi_slave_param_mode0`) y lo conecta con un **banco de registros** interno de 128x8 (128 registros de 8 bits) que representan la memoria de registros a la cual se van a realizar lecturas/escrituras. Este módulo se encarga de mapear las señales de lectura/escritura de la capa SPI al almacenamiento real.

Dentro de `top_spi_regs` (nombre del módulo top), se declara un arreglo `regfile` de 128 posiciones de 8 bits, que constituye el banco de registros accesibles vía SPI. La idea es que los 7 bits de dirección (`addr_out`) indexan este arreglo (0 a 127).

Para las **lecturas**, se implementa una lectura combinacional: `assign data_in = regfile[addr_out]`. Esto significa que en todo momento, `data_in` (que alimenta al módulo SPI_slave) reflejará el contenido del registro apuntado por `addr_out`. En la práctica, cuando se esté llevando a cabo una operación de lectura SPI, en cuanto los primeros 8 bits (R/W=0 + dirección) sean recibidos y latcheados, la señal `addr_out` del esclavo se actualizará con la dirección. Inmediatamente (combinacionalmente) `data_in` tomará el valor almacenado en `regfile[esa_dirección]`. Luego, como vimos, el esclavo cargará este `data_in` en su registro de transmisión para enviarlo de vuelta al maestro. De esta manera, el **dato correcto se coloca a tiempo en MISO** durante la segunda parte de la transacción de lectura.

Para las **escrituras**, la llegada de la trama completa con R/W=1 activará `write_enable` por un pulso. Este pulso indica que `data_out` (salida del esclavo) tiene un byte válido que debe ser escrito en la dirección `addr_out`. Sin embargo, hay que tener cuidado con la sincronización: las señales `addr_out`, `data_out` y `write_enable` se actualizan en el flanco de subida final de `SCLK` (al terminar de recibirse el bit 15). Si uno intentara escribir inmediatamente en ese mismo instante, podría haber una carrera de tiempos. Por ello, en `top_spi.v` se realiza la escritura en el **flanco de bajada de** `SCLK` siguiente, cuando las señales ya están estabilizadas. Esto se implementa con un bloque `always @(negedge sclk or negedge rst_n)` donde, si el sistema no está en reset y el esclavo está activo, **y** `write_enable` está en 1, entonces se actualiza el registro: `regfile[addr_out] <= data_out`. La condición `!ss_n && write_enable` asegura que solo cuando el esclavo esté seleccionado y efectivamente se haya indicado una escritura, se ejecute la asignación. Así, en la práctica, al final del ciclo de reloj en que llegó el último bit, durante el flanco descendente, se escribirá el dato en la memoria. Esto ocurre antes de que el maestro eventualmente levante `SS` (lo cual en nuestro testbench sucede un poco después). La inserción de esta escritura en negedge previene problemas de timing (carrera) entre las señales provenientes del esclavo y la escritura en el registro, tal como lo comenta el código fuente.

El resto del módulo top simplemente conecta las señales entre la instancia del esclavo SPI (`u_spi`) y los wires declarados. Por ejemplo, `addr_out`, `data_out`, etc., en `top_spi_regs` se conectan directamente a los puertos homónimos del submódulo `spi_slave_param_mode0`. Nótese que al instanciar, se pasan los parámetros para configurar el submódulo: 16 bits de frame, 7 de dirección, 8 de datos, etc., concordando con nuestro protocolo.

En síntesis, `top_spi.v` actúa como **envoltorio** que integra la lógica SPI con un pequeño banco de registros interno:
- Si el maestro SPI escribe (RW=1) en una dirección, el esclavo captura el dato y activa `write_enable`, y el top module en el próximo negedge escribe ese dato en `regfile` en la posición dada.
- Si el maestro SPI lee (RW=0) una dirección, el top module ya estará proporcionando el valor almacenado en esa dirección al esclavo vía `data_in`, de modo que el esclavo lo envíe por MISO. No se modifica nada en `regfile` en este caso.

Cabe mencionar que en este diseño las señales `done` y `rx_frame` provenientes del esclavo no se usan activamente en `top_spi.v` (no están conectadas a ninguna otra lógica o salidas del módulo). Podrían ser útiles para depuración o para expandir funcionalidad (por ejemplo, generar una interrupción al finalizar una operación SPI, etc.), pero por ahora son internas.

4. Banco de pruebas `tb_slave.v` – Simulando el maestro SPI

El testbench `tb_slave.v` provee un entorno de simulación donde un proceso actúa como maestro SPI para validar el correcto funcionamiento del esclavo (`top_spi_regs` instanciado como DUT, Device Under Test). A grandes rasgos, el testbench genera las señales de reloj y datos del maestro (`sclk` , `mosi` , `ss_n`) con la temporización adecuada para el modo 0, envía tramas de ejemplo (una de escritura y luego una de lectura) y verifica que el esclavo responda correctamente (almacenando o devolviendo el dato esperado).

4.1 Inicialización y configuración temporal

Al inicio (`initial begin`), se fijan los estados iniciales: se coloca el reset en activo (`rst_n = 0`), `ss_n = 1` (es decir, sin seleccionar el esclavo aún), `sclk = 0` (reloj en bajo, coherente con CPOL=0) y `mosi = 0` . Tras unos nanosegundos de espera (#50), se libera el reset (`rst_n = 1`) y se espera un poco antes de comenzar las operaciones.

El testbench define `T_HALF = 10` como medio periodo de reloj, implicando que un ciclo completo de `sclk` será de 20 unidades de tiempo en la simulación. Esto se usará para temporizar correctamente los flancos.

4.2 Tareas para generar la señal SPI (bit banging)

En lugar de usar un generador de reloj continuo, este testbench implementa el envío de datos SPI mediante **tareas** (tasks) que realizan *bit banging*, controlando manualmente cada transición para mayor flexibilidad:

- `task spi_begin` : Simula el inicio de una comunicación SPI. Simplemente garantiza que `sclk` esté en 0, espera un pequeño tiempo (#5) y luego baja `ss_n = 0` para seleccionar al esclavo, esperando un poco más (#5). Esto corresponde a la acción de un maestro tirando la línea SS a bajo mientras el reloj está estable en bajo (importante para CPHA=0: no debe haber flancos de reloj mientras SS estaba alto justo hasta que bajó, evitando bits fantasmas).
- `task spi_end` : Finaliza la comunicación. Pone `sclk = 0` (asegurando que termine en estado bajo) y tras unos ns sube `ss_n = 1` . Luego espera #20 como margen. Esto efectúa la deselección del esclavo al final de la trama.
- `task spi_clock_bit(input mosi_bit, output miso_bit)` : Esta es la tarea fundamental que envía un bit por MOSI y genera un pulso de reloj, capturando a la vez el bit que el esclavo coloca en MISO. Su comportamiento sigue exactamente el modo 0:
- Asigna el valor `mosi_bit` al señal `mosi` (colocando el bit en la línea MOSI).
- Espera `T_HALF` (10) unidades de tiempo con `sclk` en 0, así manteniendo estable MOSI durante la primera mitad del periodo.
- Sube el reloj: `sclk = 1` simula el flanco de subida. Después de 1 unidad de tiempo, lee la línea `miso` y guarda ese valor en la variable local `miso_bit` . Este paso imita al maestro muestreando el bit de MISO en el medio del pulso de reloj (ligeramente retrasado del flanco para capturar estable).
- Espera otro `T_HALF` con el reloj en alto, luego baja `sclk = 0` (flanco de bajada). Tras otro pequeño delay de 1, termina la tarea.

Esta secuencia garantiza que en cada invocación se genera un ciclo completo de reloj de 20 unidades (10 low, subir y samplear, 10 high, bajar) y se sincroniza adecuadamente MOSI/MISO. Notemos cómo antes de cada flanco de subida, `mosi` ya contiene el siguiente bit a enviar, y la lectura de `miso` se

hace justo después del flanco de subida – coherente con CPHA=0, donde el maestro lee en flanco ascendente. - `task spi_transfer16(input [15:0] w_mosi, output [15:0] w_miso)`: encapsula el envío de una palabra de 16 bits por SPI. Coloca inicialmente `w_miso = 16'h0000` (limpia el registro que almacenará lo recibido). Luego realiza un bucle `for (k = 15; k >= 0; k = k - 1)` que recorre de 15 a 0 todos los bits de la palabra de entrada `w_mosi`. En cada iteración llama a `spi_clock_bit(w_mosi[k], mb)` donde `mb` es un bit temporal para MISO. Esto asegura que se transmite el bit más significativo primero (porque comienza en k=15). Tras cada bit, guarda el bit recibido en la posición correspondiente de `w_miso`: `w_miso[k] = mb`. Al final de este loop, `w_miso` contendrá la palabra de 16 bits leída desde MISO durante la transferencia. Es decir, esta tarea implementa una transacción SPI completa de 16 bits, manejando tanto envío como recepción en paralelo.

4.3 Secuencia de prueba: escritura seguida de lectura

Con las tareas definidas, el testbench realiza dos operaciones para probar el esclavo: primero una **escritura** a un registro, luego una **lectura** del mismo registro para verificar que el valor fue almacenado correctamente.

- **Prueba de escritura:** Se prepara un vector de 16 bits `mosi_w` con un comando de escritura. En el código se construye como `{1'b1, 7'h12, 8'h3A}`. Desglosemos esto:
- `1'b1` corresponde al bit R/W = 1 (escritura).
- `7'h12` es la dirección 0x12 (18 decimal) a la cual queremos escribir. Son 7 bits que representan la dirección; 0x12 en hexadecimal es 00010010 en 8 bits, pero en 7 bits sigue siendo el valor decimal 18 (0x12).
- `8'h3A` es el dato de 8 bits que escribiremos (0x3A en hex, que es 58 decimal). Concatenados, forman la palabra de 16 bits `16'h923A` (el testbench lo anota como comentario). Efectivamente 0x92 = 1001 0010_2 contiene R/W=1 y dirección=0x12; luego 0x3A son los datos.

A continuación, se llama a `spi_begin(); spi_transfer16(mosi_w, miso_w); spi_end();`. Esto baja `SS`, envía los 16 bits almacenando lo recibido en `miso_w`, y luego sube `SS`. Durante esta transacción: - `mosi_w` = 0x923A sale bit a bit por MOSI. El esclavo debería interpretar esto como “escribe valor 0x3A en la dirección 0x12”. - Simultáneamente, el testbench recoge en `miso_w` los bits enviados por el esclavo en MISO. Dado que esta es una operación de escritura, esperamos que el esclavo no tenga datos útiles que devolver; por diseño debería haber enviado 16 bits en cero (0x0000) o algo irrelevante. `miso_w` no se verifica explícitamente tras la escritura, porque lo importante es el efecto de lado en la memoria.

Después de un pequeño delay (#50) para separar transacciones, el testbench verifica la escritura: comprueba `if (dut.regfile[7'h12] !== 8'h3A)`. Es decir, lee directamente el valor del registro 0x12 dentro del DUT (`dut.regfile` refiere al arreglo `regfile` del módulo top instanciado) y lo compara con 0x3A. Si no coincide, imprime un mensaje de error "FAIL WRITE..." y detiene la simulación (`$stop`); si coincide, imprime "PASS WRITE: regfile[0x12]=3A". Esto comprueba que la lógica del esclavo efectivamente escribió el valor en la posición correcta del banco de registros.

- **Prueba de lectura:** Ahora se prepara un vector `mosi_r` para leer el registro 0x12. Según nuestro protocolo, una lectura se codifica con R/W = 0. Por tanto, `mosi_r = {1'b0, 7'h12, 8'h00}`. Aquí:
- `1'b0` es el bit R/W = 0 (lectura).
- `7'h12` de nuevo la dirección 0x12.

- `8'h00` es un byte dummy (en lectura, el maestro suele mandar un byte sin importancia sólo para generar los ciclos de reloj necesarios para recibir la respuesta). Se usa 0x00. Esta palabra completa es `16'h1200` (como comenta el testbench).

Luego se ejecuta de forma análoga: `spi_begin(); spi_transfer16(mosi_r, miso_r); spi_end();`. Durante esta transacción: - El maestro envía 0x1200 por MOSI, indicando "quiero leer la dirección 0x12" y proporcionando 8 pulsos de reloj adicionales (los bits 7:0) durante los cuales espera recibir la respuesta del esclavo. - El esclavo, al reconocer R/W=0 en los primeros bits, debería haber cargado el valor del registro 0x12 (que ahora es 0x3A tras la escritura previa) y enviar ese valor por MISO en el segundo byte. - El testbench almacena lo recibido en `miso_r`. Al final de `spi_transfer16`, `miso_r` debería contener la respuesta completa del esclavo. Esperamos que los bits 15:8 de `miso_r` quizás sean 0x?? (podrían ser ceros o indefinidos durante la cabecera, dado que el esclavo no transmite nada significativo en los primeros 8 bits), y los bits 7:0 de `miso_r` deberían ser 0x3A, el dato leído.

Finalmente se verifica la lectura: `if (miso_r[7:0] !== 8'h3A)`. Es decir, compara el byte menos significativo de lo recibido (que corresponde a la segunda mitad de la trama, supuesta respuesta) con 0x3A. Si no coincide, se imprime "FAIL READ..." mostrando el valor recibido y el esperado, y se detiene la simulación; si coincide, imprime "PASS READ: miso_r[7:0]=3A ...". En el mensaje de éxito también imprime `full=%04h` con `miso_r` completo, para ver los 16 bits recibidos (por curiosidad, debería ser algo como 0x003A probablemente).

- Si ambas pruebas pasan, se imprime "ALL TESTS PASSED." y se termina la simulación (`$finish`).

4.4 Interpretación de las señales simuladas

Durante la simulación, el **testbench** imita perfectamente el comportamiento de un maestro SPI en modo 0: - `ss_n` baja al inicio de cada transacción y sube al final, delimitando las tramas de 16 bits. - `sclk` se maneja manualmente pero de forma que resulta en un tren de 16 pulsos durante `SS` activo, empezando siempre en 0 (idle low) y oscilando 0→1→0 con periodo constante. El primer flanco de subida ocurre poco después de `SS` bajar, tal como en una situación real. - `mosi` se coloca estable antes de cada flanco de subida, asegurando tiempos de setup adecuados, y se cambia de valor sólo en los flancos de bajada (dentro de `spi_clock_bit` primero se sube `sclk` luego se baja al final, y el siguiente ciclo pondrá un nuevo mosi_bit antes de volver a subir). - La lectura de `miso` se hace justo después del flanco de subida, que es cuando el maestro leería el bus.

El testbench también nos permite observar internamente el estado del DUT (`dut`), leyendo directamente `dut.regfile[...]` en las comprobaciones. En una simulación completa se podrían monitorizar señales internas del esclavo (como `write_enable`, `addr_out`, etc.) para asegurarse de su comportamiento, pero en este caso las verificaciones directas del efecto (contenido de memoria y valor recibido) son suficientes.

5. Resultados de la simulación y comportamiento esperado

A continuación, relacionamos lo que ocurre en la simulación con el comportamiento teórico del protocolo SPI modo 0 y las expectativas de diseño. En la figura se muestra una captura de la simulación para ambas transacciones (escritura y lectura), incluyendo las señales SPI y algunas internas relevantes del esclavo:

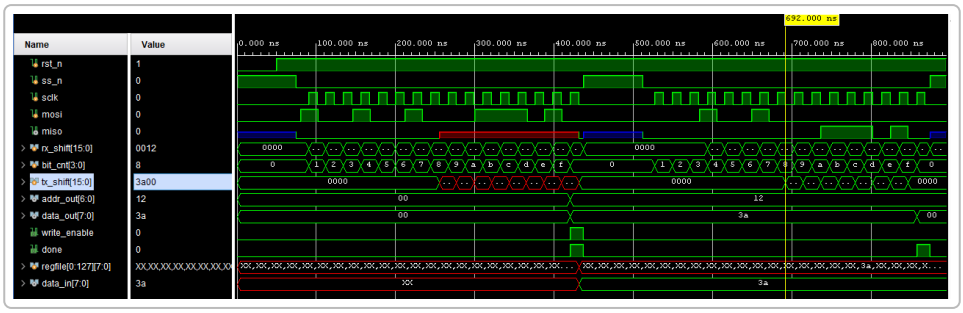

Figura 1: Captura de la simulación del esclavo SPI. Se visualizan, de arriba a abajo: la señal de reset (`rst_n`), chip select (`ss_n`), reloj serial (`sclk`), datos maestro->esclavo (`mosi`), datos esclavo->maestro (`miso`), el registro de desplazamiento de recepción `rx_shift[15:0]`, el registro de desplazamiento de transmisión `tx_shift[15:0]`, la dirección latcheada `addr_out[6:0]`, el dato capturado `data_out[7:0]`, la señal de escritura `write_enable`, la señal de fin `done`, el bus del regfile (se destaca el cambio en la dirección 0x12) y la salida combinacional `data_in` que alimenta el esclavo.

En la **primera transacción (escritura a 0x12)**, `ss_n` pasa a 0 y el maestro envía la palabra MOSI = 0x923A bit a bit (R/W=1, ADDR=0x12, DATA=0x3A). Podemos observar en la simulación:

- `mosi` (línea verde) lleva los bits **9,2,3,A** en hex a lo largo de 16 ciclos de reloj. El primer bit enviado es `1` (R/W), luego los 7 bits de la dirección 0x12, luego el byte 0x3A.
- `sclk` (amarillo) inicia en bajo y presenta 16 flancos de subida (marcados con líneas verticales) con sus correspondientes flancos de bajada. En cada flanco de subida, vemos cómo el valor en `mosi` es amostrado por el esclavo (internamente, `rx_shift` captura ese bit).
- `miso` (rojo) durante esta primera mitad permanece en 0 en cada flanco ascendente – esto corresponde a que el esclavo no tiene datos útiles aún (está enviando 0x00 por defecto).

Mirando las señales internas del esclavo:

- `rx_shift[15:0]` (traza azul claro) comienza en 0x0000 y va desplazando bits entrantes. Después de 8 bits (cuando se ha recibido 0x92), podemos apreciar que `addr_out` (traza rosa) se actualiza a 0x12 y `rw_latched` (no mostrada directamente pero implícita) es 1. En ese momento de la simulación, el esclavo reconoce que es una **escritura**, por lo que *no* carga nada en `tx_shift` (permanece en 0).
- Efectivamente, `tx_shift[15:0]` (traza verde agua) se ve que permanece en 0x0000 durante toda la transacción de escritura, indicando que el esclavo está transmitiendo solo ceros.
- Cuando llegan los últimos 8 bits (0x3A por MOSI), `rx_shift` termina con el valor completo 0x923A, que también se refleja en `rx_frame` y dividido en `addr_out=0x12`, `data_out=0x3A`. Vemos que al finalizar el bit 15, la señal `write_enable` (línea naranja) pulsa brevemente a 1, señalando que hay un dato para escribir. Al mismo tiempo, la señal `done` (no destacada en la figura pero estaría presente) también pulsa indicando fin de trama.
- Inmediatamente después (en el flanco de bajada final, antes de `SS` subir), el testbench escribe el valor 0x3A en `regfile[0x12]`. En la figura, en la línea del **regfile** (gris), se observa que la celda de dirección 0x12 cambia de valor (de un valor desconocido/indefinido `XX` a `3A`). Este cambio ocurre sincronizado con `write_enable`.

Cuando `ss_n` vuelve a 1 (desactivación del esclavo), el esclavo resetea sus registros internos (podemos ver que `addr_out`, `data_out` se limpian a 0, etc., preparándose para la siguiente operación).

En la **segunda transacción (lectura de 0x12)**, el maestro baja `ss_n` nuevamente y envía MOSI = 0x1200 (R/W=0, ADDR=0x12, dummy=0x00). Observamos en la simulación:

- `mosi` ahora muestra los bits de **1,2,0,0** (0x12 y luego 0x00). El primer bit esta vez es 0 (R/W=0 para lectura).
- `sclk` genera otro bloque de 16 pulsos separados en el tiempo de la primera transacción.
- `miso` durante los primeros 8 bits permanece en 0, pero tan pronto como el esclavo identifica la lectura y carga el dato, vemos cambiar `miso`. En la figura, a partir del bit index 8, la línea `miso` comienza a reflejar el valor **0x3A** que el esclavo envía: los bits de `miso` (rojo) durante los ciclos 8 a 15 corresponden justamente al patrón `0x3A` en binario (0011 1010). Por ejemplo, el bit 7:0 de `miso_r` resultó ser 0x3A, como confirmó la verificación.

Internamente:

- Al recibir los primeros 8 bits (0x12), el esclavo latchea `rw_latched=0` y `addr_latched=0x12`. Vemos `addr_out` actualizarse a 0x12 nuevamente tras el bit 7. Dado que es una lectura, en el flanco de bajada inmediatamente siguiente (bit_cnt==8) el esclavo carga `tx_shift` con el valor presente en `data_in`. Aquí `data_in` ya valía 0x3A porque el regfile en la dirección 0x12 fue actualizado en la operación anterior. En la figura, notamos que `data_in` (última línea azul oscuro) está en 0x3A durante esta transacción. Así, `tx_shift` cambia de 0x0000 a **0x3A00** en ese instante (colocando 0x3A en la parte alta de los 16 bits).
- Consecuentemente, en los siguientes flancos de bajada, `tx_shift` se va desplazando a la izquierda. Vemos su valor: 0x3A00, luego 0x7400, 0xE800, etc., básicamente sacando bit por bit de 0x3A por el MSB. Esto coincide con lo esperado: MISO transmite 0x3A.
- `rx_shift`, mientras tanto, termina con el valor 0x1200 (podemos ver que al final de los 16 bits de la lectura, `rx_shift` tiene 0x1200, lo cual coincide con lo enviado por el maestro). `data_out` queda con 0x00 (los últimos 8 bits recibidos del maestro son dummy y no generan acción). `write_enable` esta vez nunca se activa, ya que R/W=0 implica que no hay escritura al regfile.

Finalmente, el maestro sube `ss_n`, el esclavo se reinicia (limpia registros) y el testbench comprueba que la lectura fue correcta, habiendo recibido 0x3A en `miso_r`. El mensaje "PASS READ" confirma que `miso_r[7:0]` fue 3A, tal como se observa también en la figura donde la línea `miso` durante la segunda mitad de la segunda transacción representa ese valor.

En resumen, la simulación muestra que el esclavo SPI diseñado cumple con el **comportamiento esperado según el protocolo SPI modo 0**: sincroniza la recepción de `MOSI` con flancos ascendentes de `SCLK`, sincroniza la salida `MISO` con flancos descendentes, interpreta correctamente el bit de R/W y la dirección, realiza la escritura en memoria cuando corresponde y envía de regreso el dato almacenado en una lectura. Las señales internas (`rx_shift`, `tx_shift`, `write_enable`, etc.) evolucionan exactamente de acuerdo a la lógica descrita: - El **registro de desplazamiento de entrada** acumuló los bits seriales y al final contenía la trama completa recibida (ej. 0x923A y luego 0x1200). - El **registro de desplazamiento de salida** se cargó con el dato correcto solo en la operación de lectura y fue desplazándolo para transmitir bit por bit. - La memoria interna (**regfile**) fue actualizada con el valor correcto tras la escritura y se mantuvo para ser leída después. - La **temporización** de `MISO` concuerda con CPHA=0: en los flancos de subida, el maestro lee bits que el esclavo puso en el flanco anterior. Esto se evidencia en la tarea de testbench que lee `miso` 1 ns después del flanco de subida y obtuvo los bits correctos.

Con todo esto, los resultados de la simulación validan el diseño del esclavo SPI modo 0 en Verilog, mostrando que responde adecuadamente a comandos de un maestro SPI real, cumpliendo el protocolo

y modificando el banco de registros tal como se esperaba. Todas las pruebas pasan exitosamente, indicando que tanto la escritura como la lectura funcionan según lo planeado.
