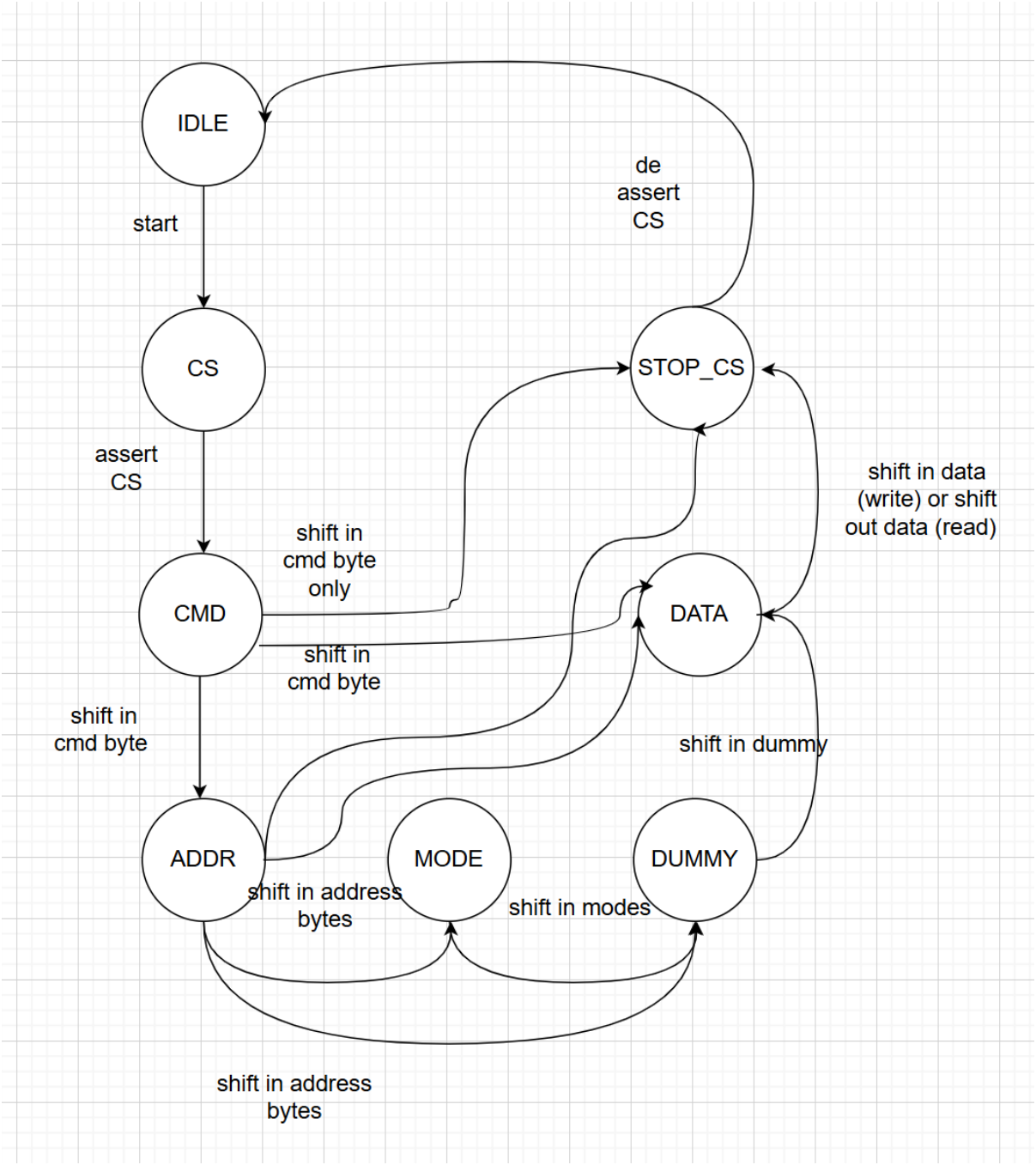

The states are:
IDLE, CS, CMD, ADDR, MODE, DUMMY, DATA, STOP_CS.

Inputs from CSR (for Command Mode, similar for XIP Mode):
- CMD_LANES (2 bits): 0=single, 1=dual, 2=quad (number of data lines for command phase)
- ADDR_LANES (2 bits): for address phase
- DATA_LANES (2 bits): for data phase
- ADDR_BYTES (2 bits): 0=0 bytes, 1=3 bytes, 2=4 bytes (address length)
- MODE_EN (1 bit): whether to send mode bits after address
- DUMMY_CYCLES (4 bits): number of dummy cycles (0-15)
- DIR (1 bit): 0=write (controller to flash), 1=read (flash to controller)

- OPCODE (8 bits): command opcode
- MODE_BITS (8 bits): mode bits (if MODE_EN is set)
- ADDR (32 bits): address (only lower 24 or 32 bits used as per ADDR_BYTES)
- LEN (32 bits): number of data bytes to transfer (for data phase)

Additionally, we have:
- start: from Command Engine (CE) to start the transaction.
- For data phase, we have FIFO interfaces (TX FIFO for write, RX FIFO for read).

FSM States and Operations:

1. IDLE:
- Wait for the start signal from CE.
- When start is asserted, move to CS state.

2. CS:
- Assert cs_n (active low) to select the flash device.
- Prepare to send the command opcode.
- Move immediately to CMD state (on the next clock edge).

3. CMD:
- Shift out the opcode on the configured number of lanes (1, 2, or 4).
- The opcode is 8 bits. The number of cycles required = ceil (8 / number_of_lanes). For example, with quad lanes (4), it takes 2 cycles.
- After sending the opcode, check the next phases:
    - If ADDR_BYTES > 0, go to ADDR state.
    - Else if MODE_EN is set, go to MODE state.
    - Else if DUMMY_CYCLES > 0, go to DUMMY state.
    - Else if LEN > 0 (data phase), go to DATA state.
    - Else, go to STOP_CS (no data, so end the transaction).

4. ADDR:
- Shift out the address bytes. The number of address bytes is 3 or 4 (as per ADDR_BYTES). The address is sent MSB first.
- The number of cycles = ceil((ADDR_BYTES * 8) / number_of_lanes (for address phase)).
- After sending the address, check:
    - If MODE_EN is set, go to MODE state.
    - Else if DUMMY_CYCLES > 0, go to DUMMY state.
    - Else if LEN > 0, go to DATA state.
    - Else, go to STOP_CS.

5. MODE:
- Shift out the mode bits (8 bits) on the same number of lanes as the address phase (as per spec: "Mode bits are sent with same number of lanes as in ADDR_LANES").
- Cycles = ceil(8 / ADDR_LANES).
- Then, check:
    - If DUMMY_CYCLES > 0, go to DUMMY state.
    - Else if LEN > 0, go to DATA state.
    - Else, go to STOP_CS.

6. DUMMY:

- Wait for the configured number of dummy cycles. During dummy cycles, the IO lines are typically driven to high-impedance (for read) or don't care (for write).
- We will count down the dummy cycles. On each clock cycle, we decrement the dummy counter.
- After the dummy cycles, if LEN>0, go to DATA state, else go to STOP_CS.

7. DATA:
- Transfer data bytes. The direction (read or write) is determined by the DIR bit.
- The number of lanes is configured by DATA_LANES.
    - **For write (DIR=0):** read data from TX FIFO and shift out on the data lanes. If FIFO becomes empty and we haven't sent all data, we might need to wait (but note: the FIFO should be filled by the DMA or CPU in advance). However, if underrun occurs, we need to handle error.
    - **For read (DIR=1):** shift in data from the IO lines and write to RX FIFO. If FIFO becomes full, we might need to wait (but in DMA mode, the DMA should read the FIFO quickly). If overrun, handle error.
- We must transfer LEN bytes. The number of cycles = ceil ((LEN * 8) / DATA_LANES). We transfer one beat per cycle (each beat is 1, 2, or 4 bits), we are transferring multiple bits per cycle, so we can accumulate to form bytes and write to FIFO in bytes.

However, note: the FIFOs are byte-oriented. So we need to accumulate the bits until we have a full byte (or more) to write to the FIFO. But the data transfer in the FSM is in terms of beats (each beat: 1,2,4 bits). We can use a shift register to accumulate the incoming bits until we have 8 bits, then write to the FIFO. Similarly, for write, we read a byte from the FIFO and then shift it out in multiple beats.

Alternatively, we can design the data phase to handle one byte at a time: for example, in quad mode, 2 cycles per byte (because 4 bits per cycle -> 2 cycles for 8 bits). The data phase is a continuous stream. We can use a state machine that handles one byte at a time, but that would be inefficient for long bursts.

Instead, we can use a larger shift register and a counter for the number of bits transferred. We'll design as follows:

- For write (TX):

We will read a word (32 bits) from the TX FIFO (if available) and then shift out the bits in groups of DATA_LANES until we have sent the required number of bytes (LEN). The TX FIFO is 32 bits, we can read one byte at a time from the FIFO to simplify. Since the FIFO is 32 bits, we can read 4 bytes at once and then shift out 4 bytes in quad mode in 8 cycles (because 4 bits per cycle * 8 cycles = 32 bits). But we must handle arbitrary LEN.

- For read (RX):

We will shift in groups of DATA_LANES bits and accumulate until we have a byte (or multiple bytes) and then write to the RX FIFO.

However, to keep the design simple and avoid complex buffering, we can process one byte at a time. The performance may not be optimal but it is acceptable for the base design.

Steps for data phase:

We'll break the data transfer into byte transfers. For each byte:

- Write (TX):
Read one byte from TX FIFO (if available, else wait and signal underrun if timeout).
Shift out the byte in ceil(8/DATA_LANES) cycles, LSB first or MSB first as configured (LSB_FIRST bit in CTRL register).
- Read (RX):
Shift in 8 bits in ceil(8/DATA_LANES) cycles, then write the byte to RX FIFO (if there is space, else wait and signal overrun if timeout).
The specification supports up to 4 lanes, so for quad mode, we can do 4 bits per cycle -> 2 cycles per byte.
We'll use a bit counter for the current byte (counts from 0 to 7) and a byte counter for the remaining bytes (LEN).
Alternatively, we can transfer multiple bytes at once by using a larger shift register. However, for simplicity, we do one byte at a time.
8. STOP_CS:
- Deassert cs_n (set to high).
- Signal done to the Command Engine (CE).
- Return to IDLE.
- Clock Phase (CPHA) and Polarity (CPOL): The FSM must generate the SCLK appropriately. The state machine transitions on the rising edge of the system clock, but the SCLK is generated by a clock divider and its edges must be aligned with the data. We will have a separate process for SCLK generation (based on CLK_DIV register) and then use the FSM to change data on the appropriate edge of SCLK. However, the specification of the FSM in the diagram does not detail the SCLK generation. We assume that the FSM controls the data and the SCLK is generated in a separate module that is synchronized with the FSM.
- Since the QSPI-FSM block handles serialization/deserialization, we need to generate the SCLK and control the data IOs. The SCLK is generated by dividing the system clock (as per CLK_DIV) and the phase and polarity are set by CPOL and CPHA.
- The FSM runs at the system clock, but the data is shifted on the edges of SCLK. We can generate the SCLK in a separate process and then have the FSM change data on the falling edge of SCLK (for mode 0) and sample on the rising edge (for mode 0).
But note: the FSM state transitions are on the system clock. We can run the FSM at the SCLK rate =>The FSM runs at the SCLK rate. We generate an enable signal (sclk_en) that is high for one system clock cycle at the desired SCLK rate. Then, the FSM transitions only when sclk_en is high.The FSM will have a condition: if sclk_en='1' then we do the next step in the state.
Generate sclk_en: We only need one pulse per SCLK cycle. We can generate sclk_en at the rate of SCLK (so frequency = system clock / (2*divider)). **But note:** the SCLK has two edges. We need to control the data on one edge and sample on the other. We'll generate two enables:
sclk_posedge_en: active when the SCLK is about to have a rising edge (so we sample input data on the next system clock rising edge if CPHA=0, or change data if CPHA=1)
sclk_negedge_en: active when the SCLK is about to have a falling edge (so we change output data if CPHA=0, or sample input data if CPHA=1)

However, we can simplify: the FSM will run at the system clock and the shifting will be controlled by the enables. The clock divider will generate the SCLK and the enables.
We design a clock divider module that:
- Takes clk, rst_n, and CLK_DIV value.
- Generates sclk_o (the actual SCLK output) and two enables:
    - **en_neg:** active for one system clock cycle at the negative edge of SCLK (for changing data on the next edge, which is rising for mode 0, or sampling for mode 1)
    - **en_pos:** active for one system clock cycle at the positive edge of SCLK (for sampling for mode 0, or changing data for mode 1)

The FSM will use these enables to shift data and sample data.
But note: the FSM states are not necessarily tied to the SCLK edges. For example, in the DUMMY state, we just count cycles (each dummy cycle is one SCLK cycle). So we can use the en_neg (or en_pos) to count one cycle per SCLK cycle.
We can use one of the enables to advance the state machine. Use en_neg for counting and shifting because in mode 0 (CPHA=0) the data is set on the falling edge and sampled on the rising edge. So we change data on the falling edge (using en_neg) and sample on the rising edge (using en_pos). But the FSM state machine must run on the system clock and use the enables to know when to shift and when to sample.
For the base design, we can assume the system clock is the same as the SCLK (CLK_DIV=0) and only support mode 0 (CPOL=0, CPHA=0). Then, we can generate the SCLK as the same as the system clock.
FSM runs at the system clock, but the data must be stable for the flash device on the rising and falling edges of SCLK. For the base design, we will run the FSM at the system clock and generate the SCLK as follows:
sclk <= (not state=IDLE) and (system_clock divided by 2) when CLK_DIV=0.
Actually, we have a clock divider module that generates SCLK.
We will design the FSM to be independent of the SCLK edges, but the shifting and sampling will be done by a separate process that is synchronized with the SCLK. This is beyond the scope of the FSM design.
Given the time, we will focus on the state transitions and assume that the shifting is done by a separate serializer/deserializer that is controlled by the FSM and uses the SCLK.
We will design the FSM to control the phases and the data flow, and the actual pin control (with SCLK) will be in a separate module.
Therefore, in the FSM, we will assume that we have:
- A shift register for output data (for command, address, mode, and write data) that is shifted by the number of lanes on each step.
- A shift register for input data (for read data) that captures the input on each step.
And we will have a counter for the number of steps in each state.
one step in the state machine corresponds to one SCLK cycle. But in one SCLK cycle, we shift one beat (1,2,4 bits). So the FSM will stay in the CMD state for a number of steps = ceil(8/CMD_LANES). Similarly for other states.
We design the FSM to count the steps and then move to the next state when the step counter reaches zero.

Implementation:
We will have:
state : IDLE, CS, CMD, ADDR, MODE, DUMMY, DATA, STOP_CS
step_counter : counts the number of steps (SCLK cycles) remaining in the current state for fixed-length states (CMD, ADDR, MODE). For DUMMY, it counts the dummy cycles. For DATA, we count the number of bytes remaining.
Steps:
- In IDLE:
step_counter = 0
wait for start
- In CS:
cs_n <= '0'
step_counter = 0
next state: CMD
- In CMD:
We load the opcode into a shift register. The number of steps = ceil(8 / cmd_lanes). For example, if cmd_lanes=4, steps=2.
On each step (each SCLK cycle), we shift out cmd_lanes bits (MSB first by default, unless LSB_FIRST is set).
After the steps, we move to the next state.
Similarly for ADDR and MODE.
- In DUMMY:
step_counter = dummy_cycles (from configuration)
On each step, we do nothing (just wait) and count down.
After step_counter reaches 0, move to next state.
- In DATA:
We are going to transfer LEN bytes. But we break it into steps: each step we transfer one beat (1,2,4 bits) but we need to form bytes. We will use a bit counter for the current byte and a byte counter for the remaining bytes.
Alternatively, we can count the total number of beats: total_beats = (LEN * 8) / data_lanes.
This becomes complex. We will transfer one byte at a time. For each byte:
- If write: we load one byte from TX FIFO into a shift register. Then we shift it out in ceil(8/data_lanes) steps.
- If read: we shift in data_lanes bits per step until we have 8 bits, then we write the byte to the RX FIFO.
But we must do this for LEN bytes.
We'll have:
byte_counter = LEN (number of bytes remaining)
bit_counter = 0 (for the current byte)
For write:
if byte_counter > 0 and bit_counter==0, then load a byte from TX FIFO (and decrement byte_counter, and set bit_counter=8)
Then, each step: shift out min(bit_counter, data_lanes) bits. Actually, we always shift data_lanes bits. But if bit_counter < data_lanes, then we shift the remaining bits and pad. But the standard requires that we shift a full beat. So we require that the

data_lanes divides 8. Actually, 8 must be divisible by data_lanes. Because 1,2,4 are divisors of 8. So we can do:
steps_per_byte = 8 / data_lanes
Then we can break each byte into steps_per_byte steps.
For read: similarly, we do steps_per_byte steps per byte.
Therefore, the total steps in DATA state = LEN * (8 / data_lanes)
Note: if data_lanes is 0? It shouldn't be because we are in data phase.
However, the state machine will be in DATA state for a fixed number of steps. We can precompute total_steps = LEN * (8 / data_lanes) and then count down steps.
The LEN can be large, and the step counter would be large. We can use a counter that counts the number of steps per byte and a byte counter.
We'll do:
total_bytes = LEN
steps_per_byte = 8 / data_lanes (must be integer: 8, 4, or 2)
Then we can have:
step_counter = total_bytes * steps_per_byte
But we also need to handle the data shifting and FIFO interaction.
Alternatively, we can have two counters: byte_counter and step_in_byte. We'll use:
byte_counter: counts down from LEN to 0.
step_in_byte: counts from 0 to steps_per_byte-1.
Then, for each step in the DATA state:
if write:
if step_in_byte==0, then load a byte from TX FIFO into a shift register (and assert FIFO read enable)
shift out data_lanes bits (MSB or LSB first) from the shift register.
step_in_byte = step_in_byte + 1
if step_in_byte = steps_per_byte, then step_in_byte=0 and byte_counter = byte_counter - 1
if read:
shift in data_lanes bits into a shift register.
step_in_byte = step_in_byte + 1
if step_in_byte = steps_per_byte, then we have a full byte: write the shift register to RX FIFO (and reset the shift register, and byte_counter = byte_counter - 1)
Then when byte_counter becomes 0, we exit the DATA state.
This method is more efficient because it does not require a large step counter.
We'll use this approach.
Therefore, the FSM for the DATA state will have:
- A shift register (8 bits) for the current byte.
- A counter for step_in_byte (0 to steps_per_byte-1)
- A counter for byte_counter (from LEN down to 0)
And we also need to generate FIFO read/write enables.
However, note: the FIFOs are 32 bits, but we are reading/writing one byte at a time. We must use a FIFO interface that supports byte access. The FIFO_TX register in the CSR is a 32-bit register, but it is written by the CPU one byte at a time. Actually, the FIFO is byte-addressable. We design the FIFO as a synchronous FIFO with 32-bit width and 4-byte depth. But we are reading one byte at a time for the data phase. So

we need to read one byte from the FIFO at a time. This means the FIFO must be able to provide one byte per step. And similarly for RX, we write one byte per steps_per_byte steps.
We can design the FIFO to be 32 bits wide and then use a separate buffer to break the 32-bit word into bytes. But for simplicity, we design the FIFO to be 8 bits wide and depth = FIFO_DEPTH (in bytes). Then, the TX FIFO is 8-bit wide, and we read one byte per steps_per_byte steps. Similarly, RX FIFO is 8-bit wide.
But the specification says FIFO_DEPTH is in bytes. So we can implement the FIFOs as 8-bit wide and FIFO_DEPTH deep.
We'll do that.
Therefore, the FSM in DATA state:

- ***For write:***

if step_in_byte = 0 then
if TX FIFO is not empty, read one byte into the shift register, else wait (and eventually timeout for underrun)
end if
shift out data_lanes bits (from the MSB or LSB) of the shift register.
shift the register by data_lanes bits (in the direction of shifting).
step_in_byte = step_in_byte + 1
if step_in_byte = steps_per_byte then
step_in_byte <= 0;
byte_counter <= byte_counter - 1;
end if

- ***For read:***

shift in data_lanes bits into the shift register (in the appropriate order).
step_in_byte = step_in_byte + 1
if step_in_byte = steps_per_byte then
if RX FIFO is not full, write the shift register to the FIFO, else wait (and eventually overrun)
byte_counter <= byte_counter - 1;
step_in_byte <= 0;
end if
But note: the order of bits (LSB first or MSB first) is controlled by the LSB_FIRST bit in the CTRL register.
We must also handle the first byte: at the beginning of DATA state, we reset step_in_byte to 0 and for write, we immediately request a byte from the FIFO.
This design for DATA state will work.
Note: the above FSM does not handle the LSB_FIRST option. We would need to reverse the bit order if LSB_FIRST is set.

The Verilog file

```
// QSPI Controller - Finite State Machine (FSM)
// Author: Sang Nguyen
// Date: 2025-08-18

module qspi_fsm (
    // System Interface
    input             clk,          // System clock (100-400MHz)
    input             rst_n,        // Active-low reset

    // Control and Status Interface
    input             start,        // Start transaction signal
    output reg        done,         // Transaction completion signal
    output reg        error,        // Error indicator
    output reg [1:0]    error_type,   // Error type: 00=timeout, 01=underrun, 10=overrun,
11=config

    // Configuration Interface
    input [1:0]       cfg_cmd_lanes, // Command lanes: 00=1, 01=2, 10=4
    input [1:0]       cfg_addr_lanes,// Address lanes: 00=1, 01=2, 10=4
    input [1:0]       cfg_data_lanes,// Data lanes: 00=1, 01=2, 10=4
    input [1:0]       cfg_addr_bytes,// Address bytes: 00=0, 01=3, 10=4
    input             cfg_mode_en,   // Mode bits enable
    input [3:0]       cfg_dummy_cycles, // Dummy cycles (0-15)
    input             dir,          // Direction: 0=write, 1=read
    input [31:0]       data_len,      // Data length in bytes
    input [7:0]       opcode,       // Command opcode
    input [7:0]       mode_bits,     // Mode bits (if enabled)
    input [31:0]       addr,         // Flash address
    input [1:0]       cpol_cpha,     // SPI mode: {CPOL, CPHA}

    // FIFO Interface
    input [31:0]       tx_data,       // TX data from FIFO
    input             tx_valid,      // TX data valid
    output reg        tx_ready,      // TX ready for data
    output reg [31:0]   rx_data,       // RX data to FIFO
    output reg        rx_valid,      // RX data valid

    // QSPI Physical Interface
    output reg        sclk,         // Serial clock
    output reg        cs_n,         // Chip select (active low)
    output reg        io0_out,       // IO0 output data
    output reg        io1_out,       // IO1 output data
    output reg        io2_out,       // IO2 output data
    output reg        io3_out,       // IO3 output data
    output reg        io0_en,        // IO0 output enable
```

```
    output reg          io1_en,       // IO1 output enable
    output reg          io2_en,       // IO2 output enable
    output reg          io3_en,       // IO3 output enable
    input               io0_in,       // IO0 input data
    input               io1_in,       // IO1 input data
    input               io2_in,       // IO2 input data
    input               io3_in        // IO3 input data
);

// Parameters and State Definitions

// FSM States
typedef enum logic [2:0] {
    IDLE    = 3'b000,
    CS      = 3'b001,
    CMD     = 3'b010,
    ADDR    = 3'b011,
    MODE    = 3'b100,
    DUMMY   = 3'b101,
    DATA    = 3'b110,
    STOP_CS = 3'b111
} state_t;

// Configuration constants
parameter SINGLE = 2'b00;
parameter DUAL   = 2'b01;
parameter QUAD   = 2'b10;

// Error types
parameter ERR_TIMEOUT  = 2'b00;
parameter ERR_UNDERRUN = 2'b01;
parameter ERR_OVERRUN  = 2'b10;
parameter ERR_CONFIG   = 2'b11;

-------------------------------------
// Internal Signals and Registers
-------------------------------------

// FSM Control
reg [2:0] current_state, next_state;
reg [31:0] state_counter;      // State timeout counter
parameter STATE_TIMEOUT = 32'd10_000; // 10,000 cycles timeout

// Configuration Latches
reg [1:0]  latched_cmd_lanes;
reg [1:0]  latched_addr_lanes;
```

```
reg [1:0]  latched_data_lanes;
reg [1:0]  latched_addr_bytes;
reg        latched_mode_en;
reg [3:0]  latched_dummy_cycles;
reg        latched_dir;
reg [7:0]  latched_opcode;
reg [7:0]  latched_mode_bits;
reg [31:0] latched_addr;
reg [31:0] latched_data_len;
reg [1:0]  latched_cpol_cpha;

// Shift Registers and Counters
reg [31:0] tx_shift_reg;       // TX shift register
reg [31:0] rx_shift_reg;       // RX shift register
reg [31:0] addr_shift_reg;     // Address shift register
reg [7:0]  mode_shift_reg;     // Mode shift register

reg [5:0]  bit_counter;        // Bit counter for current phase
reg [3:0]  dummy_counter;      // Dummy cycle counter
reg [31:0] data_counter;       // Data byte counter
reg [31:0] bytes_transferred;  // Bytes transferred in DATA phase

// Clock Generation
reg [7:0]  sclk_counter;       // SCLK divider counter
reg [7:0]  clk_div_value;      // Clock divider value
reg        sclk_prev;          // Previous SCLK value
wire       sclk_rising;        // SCLK rising edge
wire       sclk_falling;       // SCLK falling edge

// Clock Divider and Edge Detection


// Calculate clock divider value (2^(clk_div+1))
always_comb begin
   case (latched_cpol_cpha[1:0])
      2'b00: clk_div_value = 8'd1;  // SCLK = clk/2
      2'b01: clk_div_value = 8'd3;  // SCLK = clk/4
      2'b10: clk_div_value = 8'd7;  // SCLK = clk/8
      2'b11: clk_div_value = 8'd15; // SCLK = clk/16
      default: clk_div_value = 8'd1;
   endcase
end

// SCLK generation
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
```

```
        sclk_counter <= 0;
        sclk <= latched_cpol_cpha[1]; // CPOL determines idle state
    end
    else if (current_state != IDLE && current_state != STOP_CS) begin
        if (sclk_counter == clk_div_value) begin
            sclk <= ~sclk;
            sclk_counter <= 0;
        end
        else begin
            sclk_counter <= sclk_counter + 1;
        end
    end
    else begin
        sclk <= latched_cpol_cpha[1]; // Return to idle state
        sclk_counter <= 0;
    end
end

// Edge detection
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        sclk_prev <= latched_cpol_cpha[1];
    end
    else begin
        sclk_prev <= sclk;
    end
end

assign sclk_rising  = (sclk_prev == 0) && (sclk == 1);
assign sclk_falling = (sclk_prev == 1) && (sclk == 0);

// Main FSM State Transition

// State transition
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        current_state <= IDLE;
    end
    else begin
        current_state <= next_state;
    end
end

// Next state logic
always_comb begin
    next_state = current_state;
```

```
case (current_state)
    IDLE: begin
        if (start) next_state = CS;
    end

    CS: begin
        next_state = CMD;
    end

    CMD: begin
        if (bit_counter == 0) begin
            if (latched_addr_bytes != 0) begin
                next_state = ADDR;
            end
            else if (latched_mode_en) begin
                next_state = MODE;
            end
            else if (latched_dummy_cycles != 0) begin
                next_state = DUMMY;
            end
            else if (latched_data_len != 0) begin
                next_state = DATA;
            end
            else begin
                next_state = STOP_CS;
            end
        end
    end

    ADDR: begin
        if (bit_counter == 0) begin
            if (latched_mode_en) begin
                next_state = MODE;
            end
            else if (latched_dummy_cycles != 0) begin
                next_state = DUMMY;
            end
            else if (latched_data_len != 0) begin
                next_state = DATA;
            end
            else begin
                next_state = STOP_CS;
            end
        end
    end
```

```
      MODE: begin
         if (bit_counter == 0) begin
            if (latched_dummy_cycles != 0) begin
               next_state = DUMMY;
            end
            else if (latched_data_len != 0) begin
               next_state = DATA;
            end
            else begin
               next_state = STOP_CS;
            end
         end
      end

      DUMMY: begin
         if (dummy_counter == 0) begin
            if (latched_data_len != 0) begin
               next_state = DATA;
            end
            else begin
               next_state = STOP_CS;
            end
         end
      end

      DATA: begin
         if (data_counter == 0) begin
            next_state = STOP_CS;
         end
         else if (state_counter > STATE_TIMEOUT) begin
            next_state = STOP_CS; // Timeout
         end
      end

      STOP_CS: begin
         next_state = IDLE;
      end

      default: next_state = IDLE;
   endcase
end

// FSM Control Logic

// Latch configuration at start
```

```
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      latched_cmd_lanes    <= SINGLE;
      latched_addr_lanes   <= SINGLE;
      latched_data_lanes   <= SINGLE;
      latched_addr_bytes   <= 0;
      latched_mode_en      <= 0;
      latched_dummy_cycles <= 0;
      latched_dir          <= 0;
      latched_opcode       <= 0;
      latched_mode_bits    <= 0;
      latched_addr         <= 0;
      latched_data_len     <= 0;
      latched_cpol_cpha    <= 0;
   end
   else if (current_state == IDLE && start) begin
      latched_cmd_lanes    <= cfg_cmd_lanes;
      latched_addr_lanes   <= cfg_addr_lanes;
      latched_data_lanes   <= cfg_data_lanes;
      latched_addr_bytes   <= cfg_addr_bytes;
      latched_mode_en      <= cfg_mode_en;
      latched_dummy_cycles <= cfg_dummy_cycles;
      latched_dir          <= dir;
      latched_opcode       <= opcode;
      latched_mode_bits    <= mode_bits;
      latched_addr         <= addr;
      latched_data_len     <= data_len;
      latched_cpol_cpha    <= cpol_cpha;
   end
end

// State counter for timeout detection
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      state_counter <= 0;
   end
   else begin
      if (current_state != next_state) begin
         state_counter <= 0;
      end
      else if (current_state != IDLE) begin
         state_counter <= state_counter + 1;
      end
   end
end
```

```
// Command Phase (CMD)

// CMD phase control
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      tx_shift_reg <= 0;
      bit_counter <= 0;
   end
   else if (current_state == CS) begin
      // Initialize CMD phase
      tx_shift_reg <= {24'h0, latched_opcode};

      // Calculate bits to transmit based on lanes
      case (latched_cmd_lanes)
         SINGLE: bit_counter <= 8;  // 8 bits
         DUAL:   bit_counter <= 4;  // 4 cycles (2 bits/cycle)
         QUAD:   bit_counter <= 2;  // 2 cycles (4 bits/cycle)
         default: bit_counter <= 8;
      endcase
   end
   else if (current_state == CMD) begin
      // Shift data on appropriate clock edge
      if ((latched_cpol_cpha[0] == 0 && sclk_falling) ||
         (latched_cpol_cpha[0] == 1 && sclk_rising)) begin

         if (bit_counter > 0) begin
            // Shift data based on lane configuration
            case (latched_cmd_lanes)
               SINGLE: begin
                  io0_out <= tx_shift_reg[7];
                  tx_shift_reg <= tx_shift_reg << 1;
               end
               DUAL: begin
                  io0_out <= tx_shift_reg[7];
                  io1_out <= tx_shift_reg[6];
                  tx_shift_reg <= tx_shift_reg << 2;
               end
               QUAD: begin
                  io0_out <= tx_shift_reg[7];
                  io1_out <= tx_shift_reg[6];
                  io2_out <= tx_shift_reg[5];
                  io3_out <= tx_shift_reg[4];
                  tx_shift_reg <= tx_shift_reg << 4;
               end
            endcase
```

```
            bit_counter <= bit_counter - 1;
        end
    end
  end
end

// Address Phase (ADDR)

// ADDR phase control
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    addr_shift_reg <= 0;
    bit_counter <= 0;
  end
  else if (current_state == CMD && next_state == ADDR) begin
    // Initialize ADDR phase
    case (latched_addr_bytes)
      2'b01: addr_shift_reg <= {8'h0, latched_addr[23:0]}; // 24-bit address
      2'b10: addr_shift_reg <= latched_addr;               // 32-bit address
      default: addr_shift_reg <= 0;
    endcase

    // Calculate bits to transmit based on lanes and address size
    case (latched_addr_lanes)
      SINGLE: bit_counter <= (latched_addr_bytes == 2'b01) ? 24 : 32;
      DUAL:   bit_counter <= (latched_addr_bytes == 2'b01) ? 12 : 16;
      QUAD:   bit_counter <= (latched_addr_bytes == 2'b01) ? 6 : 8;
      default: bit_counter <= 24;
    endcase
  end
  else if (current_state == ADDR) begin
    // Shift data on appropriate clock edge
    if ((latched_cpol_cpha[0] == 0 && sclk_falling) ||
      (latched_cpol_cpha[0] == 1 && sclk_rising)) begin

      if (bit_counter > 0) begin
        // Shift data based on lane configuration
        case (latched_addr_lanes)
          SINGLE: begin
            io0_out <= addr_shift_reg[31];
            addr_shift_reg <= addr_shift_reg << 1;
          end
          DUAL: begin
            io0_out <= addr_shift_reg[31];
            io1_out <= addr_shift_reg[30];
            addr_shift_reg <= addr_shift_reg << 2;
```

```
                end
                QUAD: begin
                    io0_out <= addr_shift_reg[31];
                    io1_out <= addr_shift_reg[30];
                    io2_out <= addr_shift_reg[29];
                    io3_out <= addr_shift_reg[28];
                    addr_shift_reg <= addr_shift_reg << 4;
                end
            endcase

            bit_counter <= bit_counter - 1;
        end
    end
  end
end

// Mode Phase (MODE)

// MODE phase control
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    mode_shift_reg <= 0;
    bit_counter <= 0;
  end
  else if ((current_state == ADDR || current_state == CMD) &&
           next_state == MODE) begin
    // Initialize MODE phase
    mode_shift_reg <= latched_mode_bits;

    // Calculate bits to transmit based on lanes
    case (latched_addr_lanes) // Use same lanes as address
      SINGLE: bit_counter <= 8;
      DUAL:   bit_counter <= 4;
      QUAD:   bit_counter <= 2;
      default: bit_counter <= 8;
    endcase
  end
  else if (current_state == MODE) begin
    // Shift data on appropriate clock edge
    if ((latched_cpol_cpha[0] == 0 && sclk_falling) ||
      (latched_cpol_cpha[0] == 1 && sclk_rising)) begin

      if (bit_counter > 0) begin
        // Shift data based on lane configuration
        case (latched_addr_lanes)
          SINGLE: begin
```

```
                io0_out <= mode_shift_reg[7];
                mode_shift_reg <= mode_shift_reg << 1;
            end
            DUAL: begin
                io0_out <= mode_shift_reg[7];
                io1_out <= mode_shift_reg[6];
                mode_shift_reg <= mode_shift_reg << 2;
            end
            QUAD: begin
                io0_out <= mode_shift_reg[7];
                io1_out <= mode_shift_reg[6];
                io2_out <= mode_shift_reg[5];
                io3_out <= mode_shift_reg[4];
                mode_shift_reg <= mode_shift_reg << 4;
            end
        endcase

        bit_counter <= bit_counter - 1;
      end
    end
  end
end

// Dummy Phase (DUMMY)

// DUMMY phase control
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    dummy_counter <= 0;
  end
  else if ((current_state == MODE || current_state == ADDR || current_state ==
CMD) &&
           next_state == DUMMY) begin
    // Initialize DUMMY phase
    dummy_counter <= latched_dummy_cycles;

    // Prepare for data phase
    if (latched_dir) begin
      // For read operations, tri-state IOs
      io0_en <= 0;
      io1_en <= 0;
      io2_en <= 0;
      io3_en <= 0;
    end
  end
  else if (current_state == DUMMY) begin
```

```
        // Count down dummy cycles
        if ((latched_cpol_cpha[0] == 0 && sclk_rising) ||
            (latched_cpol_cpha[0] == 1 && sclk_falling)) begin
            if (dummy_counter > 0) begin
                dummy_counter <= dummy_counter - 1;
            end
        end
    end
end

// Data Phase (DATA)

// DATA phase control
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        tx_shift_reg    <= 0;
        rx_shift_reg    <= 0;
        data_counter    <= 0;
        bytes_transferred <= 0;
        tx_ready        <= 0;
        rx_valid        <= 0;
        rx_data         <= 0;
    end
    else if (current_state == DUMMY && next_state == DATA) begin
        // Initialize DATA phase
        data_counter <= latched_data_len;
        bytes_transferred <= 0;

        if (latched_dir) begin
            // Read operation - tri-state IOs
            io0_en <= 0;
            io1_en <= 0;
            io2_en <= 0;
            io3_en <= 0;
        end
        else begin
            // Write operation - drive IOs
            io0_en <= 1;
            io1_en <= (latched_data_lanes >= DUAL);
            io2_en <= (latched_data_lanes >= QUAD);
            io3_en <= (latched_data_lanes >= QUAD);

            // Load first word if available
            if (tx_valid) begin
                tx_shift_reg <= tx_data;
                tx_ready <= 1;
```

```
        end
    end
end
else if (current_state == DATA) begin
    rx_valid <= 0;

    // Handle data transfer on appropriate clock edge
    if (latched_dir) begin
        // READ operation
        if ((latched_cpol_cpha[0] == 0 && sclk_rising) ||
            (latched_cpol_cpha[0] == 1 && sclk_falling)) begin

            // Sample input data based on lane configuration
            case (latched_data_lanes)
                SINGLE: begin
                    rx_shift_reg <= {rx_shift_reg[30:0], io0_in};
                end
                DUAL: begin
                    rx_shift_reg <= {rx_shift_reg[29:0], io1_in, io0_in};
                end
                QUAD: begin
                    rx_shift_reg <= {rx_shift_reg[27:0], io3_in, io2_in, io1_in, io0_in};
                end
            endcase

            // Update counters
            case (latched_data_lanes)
                SINGLE: bytes_transferred <= bytes_transferred + 1;
                DUAL:   bytes_transferred <= bytes_transferred + 2;
                QUAD:   bytes_transferred <= bytes_transferred + 4;
            endcase

            // Check if we have a complete word
            if ((bytes_transferred % 4) == 0) begin
                rx_data <= rx_shift_reg;
                rx_valid <= 1;
            end

            // Update data counter
            case (latched_data_lanes)
                SINGLE: data_counter <= data_counter - 1;
                DUAL:   data_counter <= data_counter - 2;
                QUAD:   data_counter <= data_counter - 4;
            endcase
        end
    end
```

```
else begin
   // WRITE operation
   if ((latched_cpol_cpha[0] == 0 && sclk_falling) ||
      (latched_cpol_cpha[0] == 1 && sclk_rising)) begin

      // Output data based on lane configuration
      case (latched_data_lanes)
         SINGLE: begin
            io0_out <= tx_shift_reg[31];
            tx_shift_reg <= tx_shift_reg << 1;
         end
         DUAL: begin
            io0_out <= tx_shift_reg[31];
            io1_out <= tx_shift_reg[30];
            tx_shift_reg <= tx_shift_reg << 2;
         end
         QUAD: begin
            io0_out <= tx_shift_reg[31];
            io1_out <= tx_shift_reg[30];
            io2_out <= tx_shift_reg[29];
            io3_out <= tx_shift_reg[28];
            tx_shift_reg <= tx_shift_reg << 4;
         end
      endcase

      // Update counters
      case (latched_data_lanes)
         SINGLE: bytes_transferred <= bytes_transferred + 1;
         DUAL:   bytes_transferred <= bytes_transferred + 2;
         QUAD:   bytes_transferred <= bytes_transferred + 4;
      endcase

      // Update data counter
      case (latched_data_lanes)
         SINGLE: data_counter <= data_counter - 1;
         DUAL:   data_counter <= data_counter - 2;
         QUAD:   data_counter <= data_counter - 4;
      endcase

      // Request next word if needed
      if (bytes_transferred % 4 == 0 && data_counter > 0) begin
         tx_ready <= 1;
      end
   end

   // Load next word when ready
```

```
            if (tx_ready && tx_valid) begin
                tx_shift_reg <= tx_data;
                tx_ready <= 0;
            end
        end
    end
    else begin
        tx_ready <= 0;
        rx_valid <= 0;
    end
end

// Chip Select and I/O Control

// Chip Select control
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cs_n <= 1'b1; // Deasserted
        done <= 0;
        error <= 0;
        error_type <= 0;
    end
    else begin
        case (current_state)
            IDLE: begin
                cs_n <= 1'b1;
                done <= 0;
                error <= 0;
            end

            CS: begin
                cs_n <= 1'b0; // Assert
            end

            STOP_CS: begin
                cs_n <= 1'b1; // Deassert
                done <= 1;

                // Check for errors
                if (state_counter > STATE_TIMEOUT) begin
                    error <= 1;
                    error_type <= ERR_TIMEOUT;
                end
                else if (!latched_dir && tx_ready && !tx_valid) begin
                    error <= 1;
                    error_type <= ERR_UNDERRUN;
```

```
            end
        end

        default: begin
            done <= 0;
        end
    endcase
  end
end

// I/O enable control
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    io0_en <= 0;
    io1_en <= 0;
    io2_en <= 0;
    io3_en <= 0;
  end
  else begin
    case (current_state)
      IDLE, STOP_CS: begin
        io0_en <= 0;
        io1_en <= 0;
        io2_en <= 0;
        io3_en <= 0;
      end

      CMD, ADDR, MODE: begin
        io0_en <= 1;
        io1_en <= (latched_cmd_lanes >= DUAL);
        io2_en <= (latched_cmd_lanes >= QUAD);
        io3_en <= (latched_cmd_lanes >= QUAD);
      end

      DATA: begin
        if (latched_dir) begin
          // Read - tri-state
          io0_en <= 0;
          io1_en <= 0;
          io2_en <= 0;
          io3_en <= 0;
        end
        else begin
          // Write - drive
          io0_en <= 1;
          io1_en <= (latched_data_lanes >= DUAL);
```

```
                io2_en <= (latched_data_lanes >= QUAD);
                io3_en <= (latched_data_lanes >= QUAD);
            end
        end

        default: begin
            // DUMMY phase uses previous configuration
        end
    endcase
  end
end

// Output Initialization

// Initialize outputs
initial begin
  sclk = 0;
  cs_n = 1;
  io0_out = 0;
  io1_out = 0;
  io2_out = 0;
  io3_out = 0;
  io0_en = 0;
  io1_en = 0;
  io2_en = 0;
  io3_en = 0;
  done = 0;
  error = 0;
  error_type = 0;
  tx_ready = 0;
  rx_valid = 0;
  rx_data = 0;
end

endmodule
```