Verilog Code

```
module phy_prog #(
    parameter DATA_WIDTH = 32  // cannot change
)(
    input                           clk,
    input                           rst,
    output reg                      o_cmd_ready,
    input                           i_cmd_valid,
    input  [15 : 0]                 i_cmd,
    input  [15 : 0]                 i_cmd_id,
    input  [39 : 0]                 i_addr,
    input  [63 : 0]                 i_data,
    input  [31 : 0]                 i_cmd_param,

    output                          o_wready,
    input                           i_wvalid,
    input  [DATA_WIDTH-1 : 0]       i_wdata,
    input                           i_wlast,

    input                           i_keep_wait,

    output reg [1:0]                o_status,

    output  reg                     o_res_valid,
    output  reg [15 : 0]            o_res_id,

    output  reg                     o_rd_st_req,
    output  reg [23 : 0]            o_rd_st_addr,
    output  reg [15 : 0]            o_rd_st_id,
    output  reg [ 2 : 0]            o_rd_st_type,

    output  reg                     io_busy,
    output  reg                     o_ce_n,
    input                           i_rb_n,
    output  reg                     o_we_n,
    output  reg                     o_cle,
    output  reg                     o_ale,
    output      [ 3 : 0]            o_re,
    output                          o_dqs_tri_en,
    input       [ 3 : 0]            i_dqs,
    output  reg [ 3 : 0]            o_dqs,
    output  reg                     o_dq_tri_en,
    output  reg [31 : 0]            o_dq
);
```

This is the declaration of the `phy_prog` module. It defines the module parameters and inputs/outputs:

- **`parameter DATA_WIDTH = 32`**: A parameter defining the width of the data bus. It's set to 32 bits and cannot be changed.

Inputs:

1. **`clk`**: Clock signal, driving the logic synchronously.
2. **`rst`**: Reset signal, when high, resets the entire FSM and all registers.
3. **`i_cmd_valid`**: Indicates if a valid command is present to be processed.
4. **`i_cmd`**: 16-bit input representing the command to be issued to the NAND flash. This could be specific operations like writing, erasing, etc.
5. **`i_cmd_id`**: 16-bit command identifier used to identify the specific command being processed.
6. **`i_addr`**: 40-bit address to specify where the data will be written in the NAND memory (row/column address).
7. **`i_data`**: 64-bit data input that holds the data to be written to the NAND flash.
8. **`i_cmd_param`**: 32-bit command parameter that contains additional information about the command:
    - **`[0]`**: Has second command cycle? (`1` - yes, `0` - no)
    - **`[3:1]`**: Number of addresses.
    - **`[15:4]`**: Busy time (wait time before proceeding).
    - **`[30:16]`**: Number of bytes of data.
    - **`[31]`**: Data source (0 - use `i_wdata`, 1 - use `i_data`).
9. **`i_wvalid`**: Indicates if the write data is valid for the current cycle.
10. **`i_wdata`**: 32-bit data input for write data.
11. **`i_wlast`**: Indicates the last piece of data in a burst write operation.
12. **`i_keep_wait`**: Keeps the controller in a WAIT state.
13. **`i_rb_n`**: Input signal that indicates the ready/busy status of the NAND chip (low when busy, high when ready).
14. **`i_dqs`**: Data strobe signal (used in DDR NAND interfaces).

Outputs:

1. **`o_cmd_ready`**: Indicates that the controller is ready to accept the next command.
2. **`o_wready`**: Indicates that the controller is ready to receive write data.
3. **`o_status`**: 2-bit status register that shows the current state of the controller (IDLE, BUSY, WAIT, READY).
4. **`o_res_valid`**: Output that signals when a result is valid (used after the operation is done).
5. **`o_res_id`**: Output that holds the ID of the command that was just executed.
6. **`o_rd_st_req`**: Read status request signal that indicates the controller is requesting a status read operation.
7. **`o_rd_st_addr`**: Address for the read status operation.
8. **`o_rd_st_id`**: ID of the read status request.
9. **`o_rd_st_type`**: Type of the status read request.
10. **`io_busy`**: Indicates whether the controller is busy (used for external monitoring).
11. **`o_ce_n`**: Output signal to control the NAND chip enable (`0` when active, `1` when inactive).
12. **`o_we_n`**: Output signal to control the NAND write enable (`0` for write).

13. **`o_cle`**: Command latch enable signal (used to latch command cycles).
14. **`o_ale`**: Address latch enable signal (used to latch addresses).
15. **`o_re`**: Read enable signal (used to control the NAND read operations).
16. **`o_dqs_tri_en`**: Data strobe output enable signal (used for tri-stating in DDR NAND).
17. **`o_dqs`**: Data strobe output signal.
18. **`o_dq_tri_en`**: Data bus tri-state enable signal (controls the direction of the data bus).
19. **`o_dq`**: 32-bit data bus for sending/receiving data to/from NAND.

Detailed Explanation of Signals

Command Handling

- **i_cmd_valid**: When asserted, this signal indicates that a new command is available. The FSM reads the command (`i_cmd`), address (`i_addr`), and data (`i_data`) when in the `IDLE` state.
- **o_cmd_ready**: The FSM asserts this signal when it is in the `IDLE` state, ready to accept a new command.

Address Handling

- **o_ale**: This signal enables the Address Latch when the FSM is sending an address to the NAND flash.
- **i_addr**: The address is provided as an input to the FSM and latched in the `ADDRESS` state.

Data Transfer

- **o_wready**: The FSM asserts this signal during the `DATA` state to indicate that it is ready to accept external data for a write operation.
- **i_wvalid**: This input indicates that the data on `i_wdata` is valid. The FSM processes this data during the `DATA` state.
- **i_wlast**: This input signal indicates the last data beat in a burst of data. When asserted, the FSM completes the data phase and transitions to the `WAIT` state.

NAND Control Signals

- **o_ce_n**: Chip Enable (active low), asserted when the FSM is communicating with the NAND flash.
- **o_we_n**: Write Enable (active low), asserted when the FSM is writing data to the NAND.
- **o_cle**: Command Latch Enable, asserted during the `COMMAND` state to send a command to the NAND.
- **o_ale**: Address Latch Enable, asserted during the `ADDRESS` state to send an address to the NAND.

Local Parameters and Registers

```
Verilog Code

localparam
    IDLE  = 10'b00_0000_0001,
    CMD1  = 10'b00_0000_0010,
    ADDR  = 10'b00_0000_0100,
    WPRE  = 10'b00_0000_1000,
    DATA  = 10'b00_0001_0000,
    WPST  = 10'b00_0010_0000,
    CMD2  = 10'b00_0100_0000,
    BUSY  = 10'b00_1000_0000,
    LOCK  = 10'b01_0000_0000,
    WAIT  = 10'b10_0000_0000;

localparam WARMUP_DATA_NUM = (PG_WARMUP << 1);
```

These are **local parameters** that define the FSM states:

- **`IDLE`**: The idle state when the controller is waiting for a new command.
- **`CMD1`**: The first command cycle (the first byte of a command is latched).
- **`ADDR`**: The state where the address is latched into the NAND flash.
- **`WPRE`**: Prepares for data write.
- **`DATA`**: Data write state where data is transferred.
- **`WPST`**: Post-data write state for timing requirements.
- **`CMD2`**: The second command cycle if applicable.
- **`BUSY`**: Waits for the NAND flash to complete the operation.
- **`LOCK`**: Locks the state machine while waiting for the NAND ready/busy signal (`i_rb_n`).
- **`WAIT`**: Waits for the `i_rb_n` signal to go high.

`WARMUP_DATA_NUM` defines a warm-up value for the number of data cycles required before actual data transfer begins.

Internal Registers and State Machine Logic

Next, let's discuss the internal state machine and the logic that makes the FSM work:

```
Verilog Code
    reg [3:0] state, next_state;
    reg [15:0] cmd_id;
    reg [39:0] addr;
    reg [63:0] data;
    reg [31:0] cmd_param;
    reg [31:0] wdata;
    reg wvalid, wlast;
    reg [1:0] status;
    reg res_valid;
    reg [15:0] res_id;
    reg rd_st_req;
    reg [23:0] rd_st_addr;
```

```
reg [15:0] rd_st_id;
reg ce_n, we_n, cle, ale;
reg [31:0] dq;
reg dq_tri_en, dqs_tri_en;
```

Internal Registers:

- **state, next_state [3:0]**: These represent the **current state** and the **next state** of the FSM. The FSM transitions from one state to another based on the inputs and the current state.
- **cmd_id [15:0]**: Holds the **command ID** currently being processed. This ID comes from `i_cmd_id` when `i_cmd_valid` is asserted.
- **addr [39:0]**: Stores the **address** received from `i_addr`. This is latched at the start of the operation when `i_cmd_valid` is high.
- **data [63:0]**: Holds the **data** to be written to the NAND flash. This comes from `i_data` during a write operation.
- **cmd_param [31:0]**: This stores additional **parameters** for the command, such as the number of addresses or data cycles, which are latched from `i_cmd_param`.
- **wdata [31:0]**: This is the **write data** received from `i_wdata`, used during write operations.
- **wvalid, wlast**: These hold the status of the **write data** validity and the **last data beat** during a write transaction. These signals mirror `i_wvalid` and `i_wlast`.
- **status [1:0]**: This register holds the current **status** of the FSM (`IDLE`, `BUSY`, `WAIT`, or `READY`).
- **res_valid**: This flag is set when the result of a command is ready to be returned to the external system.
- **res_id [15:0]**: Holds the **ID** of the command whose result is ready. This ID is latched from `i_cmd_id` during command execution.
- **rd_st_req**: This signal is asserted when the FSM needs to **read the status** from the NAND flash after executing a command.
- **rd_st_addr [23:0]**: The **address** associated with the read status request. Typically, this is the page/block address where the command was applied.
- **rd_st_id [15:0]**: The ID associated with the **read status request**, which helps to track which operation's status is being queried.
- **ce_n, we_n, cle, ale**: These are the internal control signals for the NAND flash memory. They control **chip enable**, **write enable**, **command latch enable**, and **address latch enable**.
- **dq [31:0]**: The data register used to drive the **DQ bus** during write operations or command/address transfers.
- **dq_tri_en, dqs_tri_en**: These signals control the tri-state buffers for the **DQ** and **DQS** lines, determining whether they are driven by the FSM or in high impedance mode.

This Verilog code represents a finite state machine (FSM) that transitions between various states depending on inputs and current conditions. Below is a line-by-line detailed explanation of the code and its process:

Sequential Logic Block (Triggered by Clock and Reset)

Verilog Code

```
always @(posedge i_clk or negedge i_nrst) begin
    if (!i_nrst) begin
        state <= IDLE;
    end else begin
        state <= next_state;
    end
end
```

- **`always @(posedge i_clk or negedge i_nrst)`**: This is a sequential block that gets triggered on the **rising edge** of the clock signal `i_clk` or when the reset signal `i_nrst` is **asserted low** (active-low reset).
    - **`posedge i_clk`**: Indicates the block is executed on the rising edge of the clock, ensuring the FSM is synchronized to the system clock.
    - **`negedge i_nrst`**: The FSM is also sensitive to the falling edge of `i_nrst`, meaning when `i_nrst` is low, the system will reset.
- **`if (!i_nrst)`**: This checks if the reset signal `i_nrst` is active (low).
    - **If reset is asserted**, the FSM sets the `state` to `IDLE`. This is the default or initial state the FSM enters after a reset to ensure the system starts from a known state.
- **`else`**: If `i_nrst` is not low (i.e., the system is not in reset), the FSM transitions to the `next_state`, which is determined by the combinational logic in the next block.

Combinational Logic Block (Next State Logic)

```
Verilog Code

always @(*) begin
    next_state = state;
    case (state)
        IDLE: begin
            if (i_cmd_valid) begin
                next_state = COMMAND;
            end
        end
```

- **`always @(*)`**: This is a combinational logic block that updates `next_state` based on the current state and the conditions inside the `case` statement.
    - The `(*)` sensitivity list ensures that the block is triggered when any of its inputs change, making it purely combinational.
- **`next_state = state;`**: This is a default assignment. The FSM assumes it will stay in the current state unless some specific condition changes (which is checked in the `case` statement).
- **`case (state)`**: This is a `case` statement that controls the behavior of the FSM based on its current `state`. The different states, like `IDLE`, `COMMAND`, `ADDRESS`, etc., represent different operational phases of the FSM.

IDLE State

```
Verilog Code

        IDLE: begin
            if (i_cmd_valid) begin
                next_state = COMMAND;
            end
        end
```

- **`IDLE:`**: This is the case where the FSM is in the `IDLE` state, which is the default state after reset or when the FSM is not processing any commands.
- **`if (i_cmd_valid)`**: This checks if the signal `i_cmd_valid` is high. The `i_cmd_valid` signal indicates that a new command is ready to be processed.
    - **If `i_cmd_valid` is asserted**, the FSM transitions to the `COMMAND` state. This signifies that the FSM has accepted the command and is ready to begin processing it.

COMMAND State

```
Verilog Code

        COMMAND: begin
            if (i_keep_wait) begin
                next_state = WAIT;
            end else begin
                next_state = ADDRESS;
            end
        end
```

- **`COMMAND:`**: This state is where the FSM processes the incoming command.
- **`if (i_keep_wait)`**: This checks if the `i_keep_wait` signal is high. `i_keep_wait` is typically used to signal that the FSM needs to pause or wait before proceeding. This may be required for synchronization or other timing constraints.
    - **If `i_keep_wait` is high**, the FSM transitions to the `WAIT` state, which indicates it will wait until the NAND flash is ready to proceed (e.g., after sending a command or address).
- **`else begin next_state = ADDRESS;`**: If `i_keep_wait` is not asserted, the FSM moves directly to the `ADDRESS` state, where it will begin sending the address for the command.

ADDRESS State

```
Verilog Code
        ADDRESS: begin
            if (/* address complete */) begin
                if (/* write operation */) begin
                    next_state = DATA;
                end else begin
                    next_state = WAIT;
                end
            end
```

```
    end
```

- **`ADDRESS:`**: In this state, the FSM is responsible for sending the address associated with the current command to the NAND flash. The address is typically needed for operations like read, write, and erase.
- **`if (/* address complete */)`**: This condition checks if the address has been fully sent to the NAND flash. The actual condition will depend on the address transmission mechanism and how many address cycles are required.
    - Once the address is sent, the FSM determines whether it needs to proceed with a write operation or wait.
- **`if (/* write operation */)`**: This condition checks if the current command is a write operation (as opposed to a read or other command). The actual condition here would check a bit in the command signal.
    - **If it's a write operation**, the FSM transitions to the `DATA` state to send the data.
    - **Else**, it transitions to the `WAIT` state, indicating that it needs to wait for the NAND flash to be ready.

DATA State

Verilog Code

```
        DATA: begin
            if (i_wlast) begin
                next_state = WAIT;
            end
        end
```

- **`DATA:`**: This state is used when the FSM needs to send or receive data (typically for write or read operations). In this state, data is transmitted or received over a bus to/from the NAND flash.
- **`if (i_wlast)`**: This condition checks if `i_wlast` is high. The `i_wlast` signal indicates that the last piece of data in the current transaction has been sent (in the case of a write) or received (in the case of a read).
    - **Once the last data has been transferred**, the FSM transitions to the `WAIT` state, indicating it is waiting for the NAND to finish the operation.

WAIT State

Verilog Code

```
        WAIT: begin
            if (i_rb_n) begin
                next_state = RESULT;
            end
        end
```

- **`WAIT:`**: In this state, the FSM is waiting for the NAND flash to be ready. This is common in NAND flash systems because many operations (like program, erase, or read) take time, and the FSM must wait for the NAND to signal it is ready to proceed.

- **`if (i_rb_n)`**: This condition checks the value of the `i_rb_n` signal. The `i_rb_n` signal (Ready/Busy) is typically used in NAND flash systems to indicate whether the flash memory is busy (low) or ready (high).
    - **If `i_rb_n` is high**, the FSM transitions to the `RESULT` state, meaning that the operation is complete, and the result can be processed.

RESULT State

```
Verilog Code

        RESULT: begin
            next_state = IDLE;
        end
```

- **`RESULT:`**: In this state, the FSM handles the result of the completed NAND operation. This may involve generating status signals, notifying the system that the operation is complete, and preparing for the next command.
- **`next_state = IDLE;`**: Once the result has been processed, the FSM transitions back to the `IDLE` state, waiting for the next command to be issued.

Default Case

```
Verilog Code

        default: begin
            next_state = IDLE;
        end
```

- **`default:`**: This is a safeguard to ensure that if the `state` somehow enters an undefined or invalid state, the FSM transitions back to `IDLE` to recover and wait for the next valid command.

Block 1: `flag` Register

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    flag <= 1'h0;
end else if((state == WPST) & has_cmd2)begin
    flag <= 1'h1;
end else if((state == CMD2) & is_latch_edge) begin
    flag <= 1'h0;
end
```

- **`always@(posedge clk or posedge rst)`**: This block is triggered by either the positive edge of the clock (`clk`) or the positive edge of the reset (`rst`).
- **`if (rst)`**: When `rst` (reset) is high, the `flag` is set to `1'h0` (reset state).

- **`else if((state == WPST) & has_cmd2)`**: If the current state is `WPST` (probably a Write Post state) and there is a second command (`has_cmd2` is true), the `flag` is set to `1'h1`.
- **`else if((state == CMD2) & is_latch_edge)`**: If the current state is `CMD2` (processing a second command) and `is_latch_edge` is true (probably indicating some timing edge has been reached), the `flag` is reset to `1'h0`.

Purpose:

The `flag` signal is used to track the presence of a second command (`has_cmd2`) and is reset when the second command processing (`CMD2`) is completed.

Block 2: `flag_r` Register

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    flag_r <= 1'h0;
end else begin
    flag_r <= flag;
end
```

- **`always@(posedge clk or posedge rst)`**: Another sequential block triggered by the clock or reset.
- **`if (rst)`**: When reset is asserted, the `flag_r` register is reset to `1'h0`.
- **`else`**: On every clock cycle, `flag_r` is updated with the value of `flag`.

Purpose:

`flag_r` is a delayed version of `flag`. It holds the previous value of `flag`, possibly for timing or synchronization purposes in the FSM.

Block 3: `io_busy` Signal

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    io_busy <= 1'h0;
end else if(((state == IDLE) & (~i_cmd_valid)) || (state == LOCK) || (state
== WAIT))begin
    io_busy <= 1'h0;
end else begin
    io_busy <= 1'h1;
end
```

- **`io_busy`**: This signal indicates whether the system is busy processing an operation.
- **`if (rst)`**: On reset, `io_busy` is cleared (set to `1'h0`).

- **`else if(...)`**: If the system is in the `IDLE`, `LOCK`, or `WAIT` state and no command is valid (`~i_cmd_valid`), `io_busy` remains low (not busy).
- **`else`**: If the conditions for being idle are not met, `io_busy` is set to `1'h1` (indicating that the system is busy).

Purpose:

`io_busy` is used to indicate whether the system is currently processing an operation or waiting for new commands.

Block 4: `o_we_n` Signal (Write Enable)

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    o_we_n <= 1'h1;
end else if(((state == CMD1) || (state == ADDR) || (state == CMD2)) &&
is_we_edge)begin
    o_we_n <= ~o_we_n;
end
```

- **`o_we_n`**: This signal controls the write enable (`WE`) for external devices, such as memory.
- **`if (rst)`**: On reset, `o_we_n` is set to `1'h1` (indicating no write operation).
- **`else if(...)`**: If the current state is `CMD1`, `ADDR`, or `CMD2` and the signal `is_we_edge` is true (likely indicating the correct timing edge for writing), `o_we_n` is toggled (`~o_we_n`).

Purpose:

`o_we_n` manages the write enable control signal for the memory. It toggles during the appropriate states (`CMD1`, `ADDR`, `CMD2`) when the FSM is ready to perform a write operation.

Block 5: `o_cle` Signal (Command Latch Enable)

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    o_cle <= 1'h0;
end else if((state == CMD1) || (state == CMD2) || flag || (i_cmd_valid &&
(i_cmd[7:0] == 8'h85)))begin
    o_cle <= 1'h1;
end else begin
    o_cle <= 1'h0;
end
```

- **`o_cle`**: This signal controls the command latch enable (`CLE`) for memory, which tells the memory when a command is being issued.
- **`if (rst)`**: On reset, `o_cle` is set to `1'h0`.

- **`else if(...)`**: The signal is set to `1'h1` (indicating a command is being latched) if the state is `CMD1` or `CMD2`, or if `flag` is set, or if `i_cmd_valid` is true and the command (`i_cmd[7:0]`) equals `8'h85`.

Purpose:

`o_cle` enables the memory command latch during specific states when a command needs to be sent to the memory (e.g., during `CMD1`, `CMD2`).

Block 6: `o_ale` Signal (Address Latch Enable)

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    o_ale <= 1'h0;
end else if(state == ADDR)begin
    o_ale <= 1'h1;
end else begin
    o_ale <= 1'h0;
end
```

- **`o_ale`**: This signal controls the address latch enable (`ALE`), which tells the memory when an address is being sent.
- **`if (rst)`**: On reset, `o_ale` is set to `1'h0`.
- **`else if (state == ADDR)`**: When the FSM is in the `ADDR` state (address phase), `o_ale` is set to `1'h1`.

Purpose:

`o_ale` activates during the address phase (`ADDR` state) to indicate that the address should be latched by the memory.

Block 7: `state_r` Register (State Register)

```
verilog
Copy code
always@(posedge clk or posedge rst)
if(rst) begin
    state_r <= 9'h0;
end else begin
    state_r <= state;
end
```

- **`state_r`**: This register stores the current state of the FSM with one clock cycle delay.
- **`if (rst)`**: On reset, `state_r` is cleared to `9'h0`.
- **`else`**: The `state_r` register captures the current state of the FSM (`state`) on every clock cycle.

Purpose:

`state_r` holds the previous state of the FSM. It is used to compare the current and previous states to track transitions or store state history.

Block 8: Write Enable (`o_we_n`) Control

Verilog Code

```
always@(posedge clk or posedge rst)
if(rst) begin
    o_we_n <= 1'h1;
end else if(((state == CMD1) || (state == ADDR) || (state == CMD2)) &&
is_we_edge) begin
    o_we_n <= ~o_we_n;
end
```

Purpose:

- This block manages the write enable signal `o_we_n`. The `o_we_n` signal is active-low, meaning that when it is `0`, a write operation occurs.

Detailed Explanation:

- **Reset Condition:** On reset (`rst`), `o_we_n` is set to `1'h1`, indicating that the write operation is disabled.
- **Write Control:** When the state is either `CMD1`, `ADDR`, or `CMD2`, and a write edge (`is_we_edge`) is detected, the signal `o_we_n` is toggled (i.e., flipped between `0` and `1`). This ensures that during specific command or address states, the write operation occurs at the appropriate time.

Block 9: Command Latch Enable (`o_cle`)

Verilog Code

```
always@(posedge clk or posedge rst)
if(rst) begin
    o_cle <= 1'h0;
end else if((state == CMD1) || (state == CMD2) || flag || (i_cmd_valid &&
(i_cmd[7:0] == 8'h85))) begin
    o_cle <= 1'h1;
end else begin
    o_cle <= 1'h0;
end
```

Purpose:

- The `o_cle` signal is used to indicate that a command is being latched (written to a command register). It becomes active when a command is present.

Detailed Explanation:

- **Reset Condition:** On reset, `o_cle` is set to `0`, meaning that the command latch is inactive.
- **Active Conditions:** `o_cle` is set to `1` in the following cases:
  - During the `CMD1` or `CMD2` states, indicating that the system is currently handling a command.
  - If the `flag` signal is set, which may indicate that another command is expected.
  - If a command is valid (`i_cmd_valid`) and its opcode (`i_cmd[7:0]`) is `8'h85`, which seems to be a specific case where the command latch needs to be enabled.
- **Default:** In other states, `o_cle` is deactivated (`0`).

Block 10: Address Latch Enable (`o_ale`)

```
Verilog Code

always@(posedge clk or posedge rst)
if(rst) begin
    o_ale <= 1'h0;
end else if(state == ADDR) begin
    o_ale <= 1'h1;
end else begin
    o_ale <= 1'h0;
end
```

Purpose:

- The `o_ale` signal is used to indicate that an address is being latched (written to an address register). It is only active during the address state.

Detailed Explanation:

- **Reset Condition:** On reset, `o_ale` is set to `0`, meaning that the address latch is inactive.
- **Active Condition:** If the state is `ADDR`, the system is handling an address, so `o_ale` is set to `1`.
- **Default:** In other states, the address latch is deactivated (`0`).

Block 11: State Register (`state_r`)

```
Verilog Code
always@(posedge clk or posedge rst)
if(rst) begin
    state_r <= 9'h0;
end else begin
    state_r <= state;
end
```

Purpose:

- The `state_r` register stores the previous state of the system.

Detailed Explanation:

- **Reset Condition:** On reset, `state_r` is cleared to `9'h0`, representing an idle or reset state.
- **State Latching:** On each clock cycle, `state_r` is updated to the current state (`state`), effectively keeping track of the previous state. This is useful for operations that need to reference both the current and previous state.

Block 12: Result Valid and Result ID (`o_res_valid`, `o_res_id`)

```
Verilog Code
always@(posedge clk or posedge rst)
if(rst) begin
    o_res_valid <= 1'h0;
    o_res_id    <= 16'h0;
end else if(((state_r == BUSY) || (state_r == WPST)) && (state == IDLE))
begin
    o_res_valid <= 1'h1;
    o_res_id    <= i_cmd_id;
end else begin
    o_res_valid <= 1'h0;
//    o_res_id    <= 16'h0;
end
```

Purpose:

- These signals handle the result validation and return the result ID after a command has completed.

Detailed Explanation:

- **Reset Condition:** On reset, `o_res_valid` and `o_res_id` are cleared, indicating no valid result and no result ID.
- **Result Generation:** If the previous state (`state_r`) was `BUSY` or `WPST` and the current state is `IDLE`, this means that a command has completed, so `o_res_valid` is set to `1` (indicating a valid result), and `o_res_id` is set to the current command ID (`i_cmd_id`).
- **Default:** In other cases, `o_res_valid` is cleared to `0`, signaling that no valid result is available at that time.

Block 13: Read Status Request (`o_rd_st_req`, `o_rd_st_type`, `o_rd_st_addr`, `o_rd_st_id`)

```
Verilog Code
always@(posedge clk or posedge rst)
if(rst) begin
```

```
    o_rd_st_req  <= 1'h0;
    o_rd_st_type <= 3'h0;
    o_rd_st_addr <= 24'h0;
    o_rd_st_id   <= 16'h0;
end else if((state == WAIT) && (i_rb_n == 1'h1) && (~i_keep_wait)) begin
    o_rd_st_req  <= 1'h1;
    o_rd_st_type <= {2'b0, busy_time[10]};
    o_rd_st_addr <= i_addr[39:16];
    o_rd_st_id   <= i_cmd_id;
end else begin
    o_rd_st_req  <= 1'h0;
    o_rd_st_type <= 3'h0;
    o_rd_st_addr <= 24'h0;
    o_rd_st_id   <= 16'h0;
end
```

Purpose:

- These signals handle the request to read status and provide relevant details like address and ID for the request.

Detailed Explanation:

- **Reset Condition:** On reset, all the request signals (`o_rd_st_req`, `o_rd_st_type`, `o_rd_st_addr`, and `o_rd_st_id`) are cleared.
- **Status Request Generation:** When the state is `WAIT`, the `i_rb_n` signal is asserted (`1'h1`), and `i_keep_wait` is not active, a read status request (`o_rd_st_req`) is generated. The type of request is encoded in `o_rd_st_type`, which uses the `busy_time[10]` signal. The request address is derived from the upper 24 bits of the `i_addr` signal, and the command ID is assigned to `o_rd_st_id`.
- **Default:** In all other states, no status request is generated, and all outputs are reset.

Block 14: Chip Enable (`o_ce_n`) Control

```
Verilog Code

always@(posedge clk or posedge rst)
if(rst) begin
    o_ce_n <= 1'h1;
end else if(((state == IDLE) & (~i_cmd_valid)) || (state == LOCK) || (state
== WAIT)) begin
    o_ce_n <= 1'h1;
end else begin
    o_ce_n <= 1'h0;
end
```

Purpose:

- This block manages the chip enable signal (`o_ce_n`), which is active-low. It controls when the chip is enabled or disabled based on the state.

Detailed Explanation:

- **Reset Condition:** On reset, `o_ce_n` is set to `1'h1`, disabling the chip.
- **Chip Enable Control:** If the state is `IDLE` and no command is valid (`~i_cmd_valid`), or the state is `LOCK` or `WAIT`, the chip is disabled (`1'h1`). Otherwise, the chip is enabled (`1'h0`), allowing operations to proceed.
- **Default:** The chip remains enabled in all other cases.

Block 16: Data Strobe (`o_dqs`)

```
Verilog Code

always@(posedge clk or posedge rst)
if(rst) begin
    o_dqs <= 4'hf;
end else if((state == DATA) & (~data_src) & (~i_wvalid)) begin
    o_dqs <= {4{o_dqs[3]}};
end else if(state == DATA) begin
    o_dqs <= 4'h5;
end else if((state_r == WPRE) || (state == WPST) || (state_r == WPST) ||
(flag | flag_r)) begin
    o_dqs <= 4'h0;
end else begin
    o_dqs <= 4'hf;
end
```

Purpose:

- The `o_dqs` signal controls the data strobe signal, which synchronizes the timing for data reads or writes. This logic ensures correct timing behavior for data transfers.

Detailed Explanation:

- **Reset Condition:** On reset, `o_dqs` is set to `4'hf`, which implies that the data strobe lines are disabled or inactive.
- **DATA State (With `~data_src` and `~i_wvalid`):** If the system is in the `DATA` state, and the `data_src` signal is not active (`~data_src`), and there is no valid write (`~i_wvalid`), the `o_dqs` is set to repeat the current `o_dqs[3]` value across all 4 bits. This ensures consistent strobe behavior during this condition.
- **DATA State (General):** When the system is in the `DATA` state (and neither of the above conditions applies), the `o_dqs` signal is set to `4'h5`, a specific pattern to indicate the data strobe is active for data transfer.
- **WPRE, WPST, and Flag Conditions:** In case the state is `WPRE` (Write Precharge), `WPST` (Write Post), or if the `flag` or `flag_r` is set, the data strobe is disabled by setting `o_dqs` to `4'h0`.

- **Default Case:** In all other cases, the data strobe is disabled and remains in its default inactive state (`4'hf`).

Block 17: Data Bus Tri-State Enable (`o_dq_tri_en`)

```
Verilog Code

always@(posedge clk or posedge rst)
if(rst) begin
    o_dq_tri_en <= 1'h1;
end else if( (state == CMD1) || (state == ADDR) || (state == CMD2) || (state
== DATA) ) begin
    o_dq_tri_en <= 1'h0;  // output
end else begin
    o_dq_tri_en <= 1'h1;
end
```

Purpose:

- This block controls whether the data bus (`o_dq`) is tri-stated or actively driven. In tri-state mode, the bus is effectively disconnected, allowing other devices to use it.

Detailed Explanation:

- **Reset Condition:** On reset, `o_dq_tri_en` is set to `1'h1`, meaning the data bus is in a tri-state mode (disconnected).
- **Active Output Conditions:** When the state is `CMD1`, `ADDR`, `CMD2`, or `DATA`, the data bus is actively driven (`o_dq_tri_en = 1'h0`), meaning the output data is valid and can be driven to external devices.
- **Default:** In other states, the data bus is tri-stated by setting `o_dq_tri_en` back to `1'h1`.

Block 18: Data Address Calculation (`dq_addr`) and Data Preparation (`dq_data`)

```
Verilog Code

wire [7:0] dq_addr;
assign dq_addr = (i_addr >> {addr_cnt, 3'h0}) & 8'hff;
reg  [31:0] dq_data;
always @*
if(data_cnt < WARMUP_DATA_NUM) begin
    dq_data = i_data[31:0];
end else begin
    dq_data = (i_data >> ({data_cnt, 3'h0} - (WARMUP_DATA_NUM << 3))) &
32'hffff_ffff;
end
```

Purpose:

- The `dq_addr` and `dq_data` signals are used to calculate the appropriate address and prepare the data to be sent on the data bus during the `DATA` state.

Detailed Explanation:

- **Data Address (`dq_addr`):** This 8-bit address is calculated by right-shifting the input address (`i_addr`) by the value `addr_cnt` (multiplied by 8 using `{addr_cnt, 3'h0}`), and then masking it with `8'hff` to ensure only 8 bits are taken. This address is used during the `ADDR` state to specify the memory address being accessed.
- **Data Preparation (`dq_data`):**
  - If the `data_cnt` (data counter) is less than the predefined `WARMUP_DATA_NUM`, the first 32 bits of `i_data` are selected and assigned to `dq_data`.
  - Otherwise, the `i_data` is shifted by the amount `(data_cnt * 8 - WARMUP_DATA_NUM * 8)` and masked with `32'hffff_ffff` to ensure that only the required portion of data is extracted for transmission.

This logic prepares the correct data based on the current data counter and how many cycles the data transfer has been ongoing.

Block 19: Data Output (`o_dq`)

```
Verilog Code

always@(posedge clk or posedge rst)
if(rst) begin
    o_dq <= 32'h0;
end else if(state == CMD1) begin
    o_dq <= {4{i_cmd[7:0]}};
end else if(state == CMD2) begin
    o_dq <= {4{i_cmd[15:8]}};
end else if(state == ADDR) begin
    o_dq <= {4{dq_addr}};
end else if((state == DATA) & data_src) begin
    o_dq <= dq_data;
end else if((state == DATA) & (~data_src) & (~i_wvalid)) begin
    o_dq <= {4{i_wdata[31:24]}};
end else if((state == DATA) & (~data_src)) begin
    o_dq <= i_wdata;
end else begin
    o_dq <= 32'h0;
end
```

Purpose:

- The `o_dq` signal represents the output data bus. This block manages what data gets placed onto the data bus depending on the current state and input conditions.

Detailed Explanation:

- **Reset Condition:** On reset, `o_dq` is cleared to `32'h0`, meaning no data is placed on the bus.
- **CMD1 and CMD2 States:**
  - During the `CMD1` state, the lower byte of the command (`i_cmd[7:0]`) is repeated 4 times and placed on the `o_dq` bus.
  - During the `CMD2` state, the upper byte of the command (`i_cmd[15:8]`) is repeated 4 times and placed on the bus.
- **ADDR State:** In the `ADDR` state, the calculated address (`dq_addr`, prepared in Block 18) is repeated 4 times and placed on the `o_dq` bus.
- **DATA State (With `data_src` Active):** If `data_src` is active during the `DATA` state, the prepared data (`dq_data`) is placed onto the data bus.
- **DATA State (Without `data_src`, Write Data Mode):** If `data_src` is not active, and there is no valid write (`~i_wvalid`), the upper byte of `i_wdata` is repeated 4 times and placed on the bus. If `i_wvalid` is true, the full `i_wdata` (32 bits) is placed on the bus.
- **Default Case:** In all other cases, the data bus is cleared (`32'h0`).

Block 20: Write Ready (`o_wready`)

```
Verilog Code

assign o_wready = (state == DATA) & (data_cnt >= WARMUP_DATA_NUM);
```

Purpose:

- The `o_wready` signal indicates when the system is ready to accept new write data.

Detailed Explanation:

- The system is ready for a write operation when the current state is `DATA` and the `data_cnt` is greater than or equal to the `WARMUP_DATA_NUM`. This ensures that the write pipeline is warmed up and ready to accept data for processing.