Documentation for AXI DMA Master Design

Design Choices

This module implements a Direct Memory Access (DMA) controller using the AXI protocol. The design choices include:

1. **State Machine-Based Control:** Two separate state machines manage **read and write** operations to ensure data integrity and proper handshaking with the AXI bus.
2. **FIFO for Intermediate Storage:** A synchronous FIFO buffers the data between AXI read and write transactions to decouple the two operations.
3. **AXI-Lite Protocol:** The design uses AXI-Lite handshaking for reliable data transfers, ensuring compatibility with standard AXI interfaces**.**
4. **Configurable Transfer Length:** The length of the data transfer is controlled via an input signal (length), making the module flexible.
5. **Trigger-Based Execution:** The DMA transaction begins upon receiving a trigger signal, allowing external control of transfers.
6. **Synchronous Reset: T**he module supports a synchronous reset mechanism to ensure a clean start for the read and write operations.
7. **Byte-Address Calculation**: Since AXI expects byte addresses, word addresses must be converted before sending to the slave memory.

Byte Address Conversion

AXI uses byte addressing, while memory operations might be based on word addressing. To ensure proper access, the word address must be converted to a byte address before being sent to the AXI bus. A dedicated function is implemented in Verilog to perform this conversion:

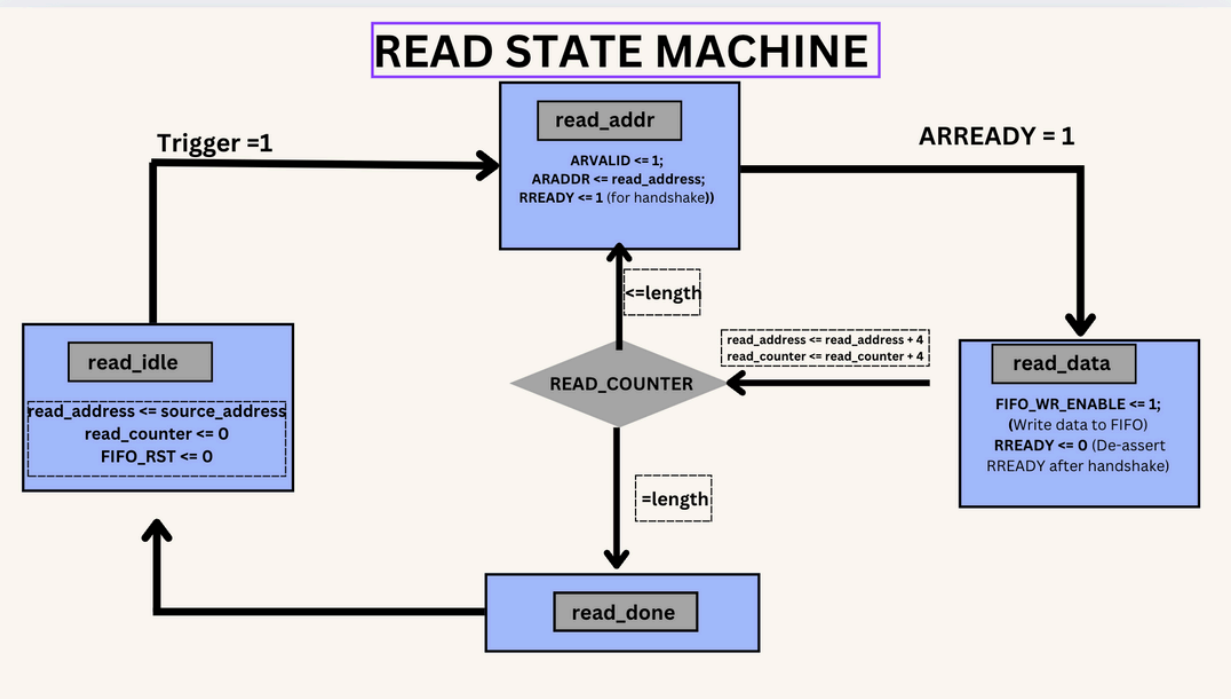
```
function [31:0] convert_to_byte_address;
    input [31:0] word_address;
    input [3:0] word_size; // Typically 4 bytes for 32-bit
begin
    convert_to_byte_address = word_address * word_size;
end
endfunction
```

State Machine Logic

Read State Machine

The read state machine controls fetching data from the source address via AXI.

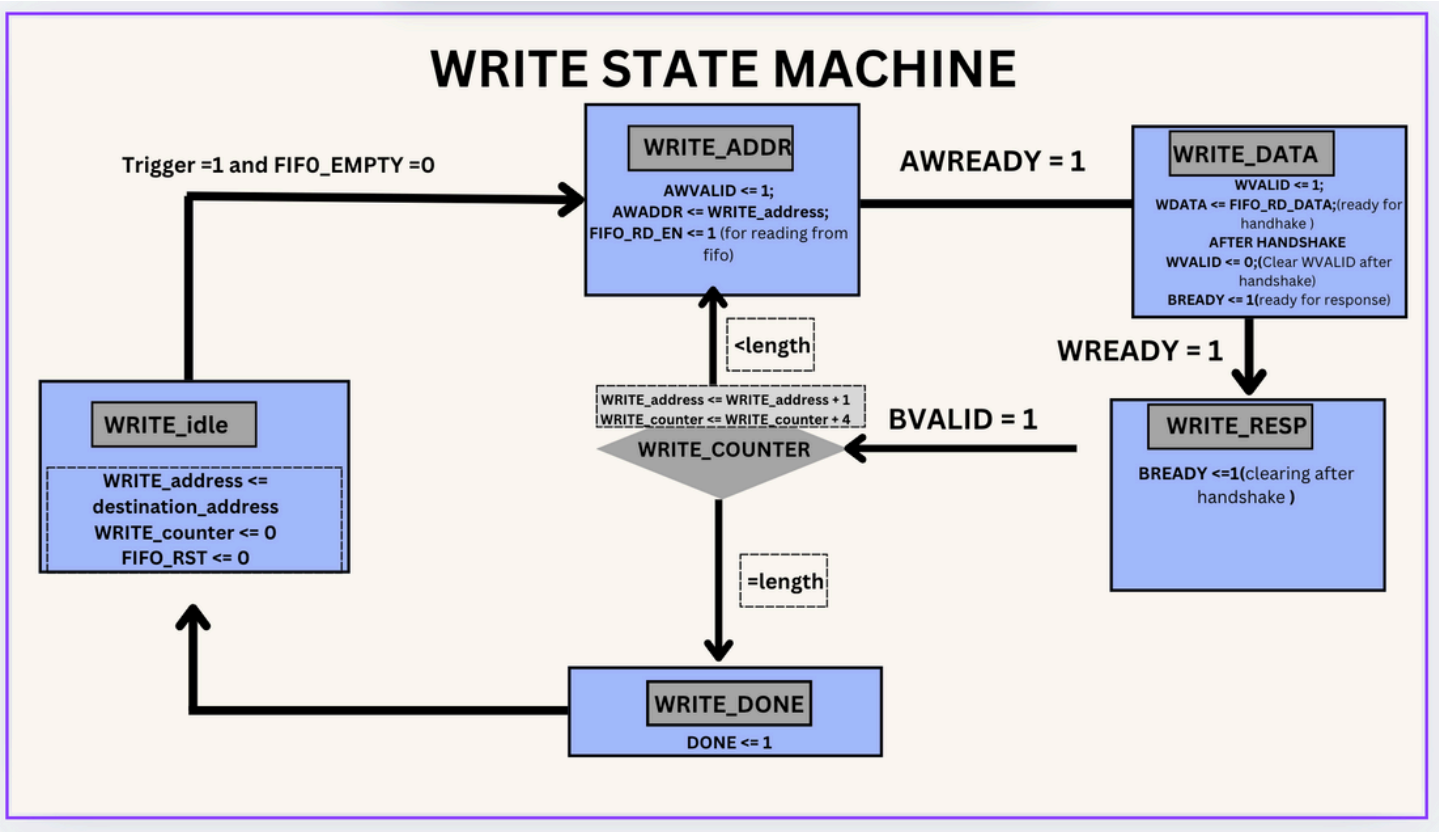
State	Description
READ_IDLE	Waits for the trigger signal to start the transaction.
READ_ADDR	Sends the read address to the AXI bus and waits for address acceptance.
READ_DATA	Waits for the data to be available from the AXI bus and writes it to the FIFO.
READ_DONE	Completes the read operation and transitions to READ_IDLE.



Write State Machine

The write state machine manages storing data into the destination address via AXI.

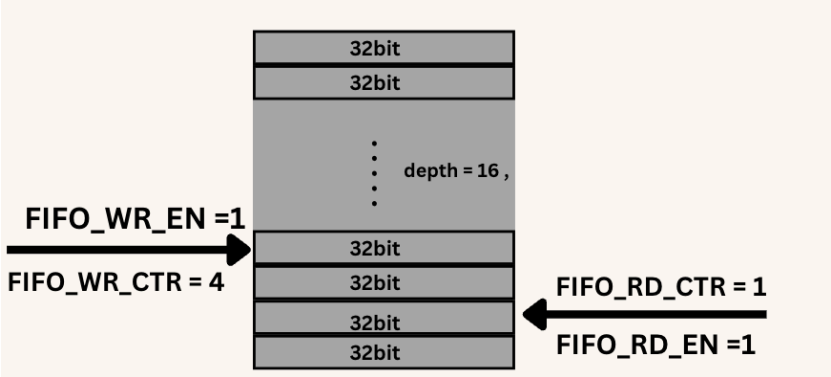
State	Description
WRITE_IDLE	Waits for data in the FIFO and a trigger signal to start the transaction.
WRITE_ADDR	Sends the write address to the AXI bus and waits for address acceptance.
WRITE_DATA	Reads data from the FIFO and writes it to the AXI bus.
WRITE_RESP	Waits for write response from the AXI bus**.**
WRITE_DONE	Completes the write operation and transitions to WRITE_IDLE.



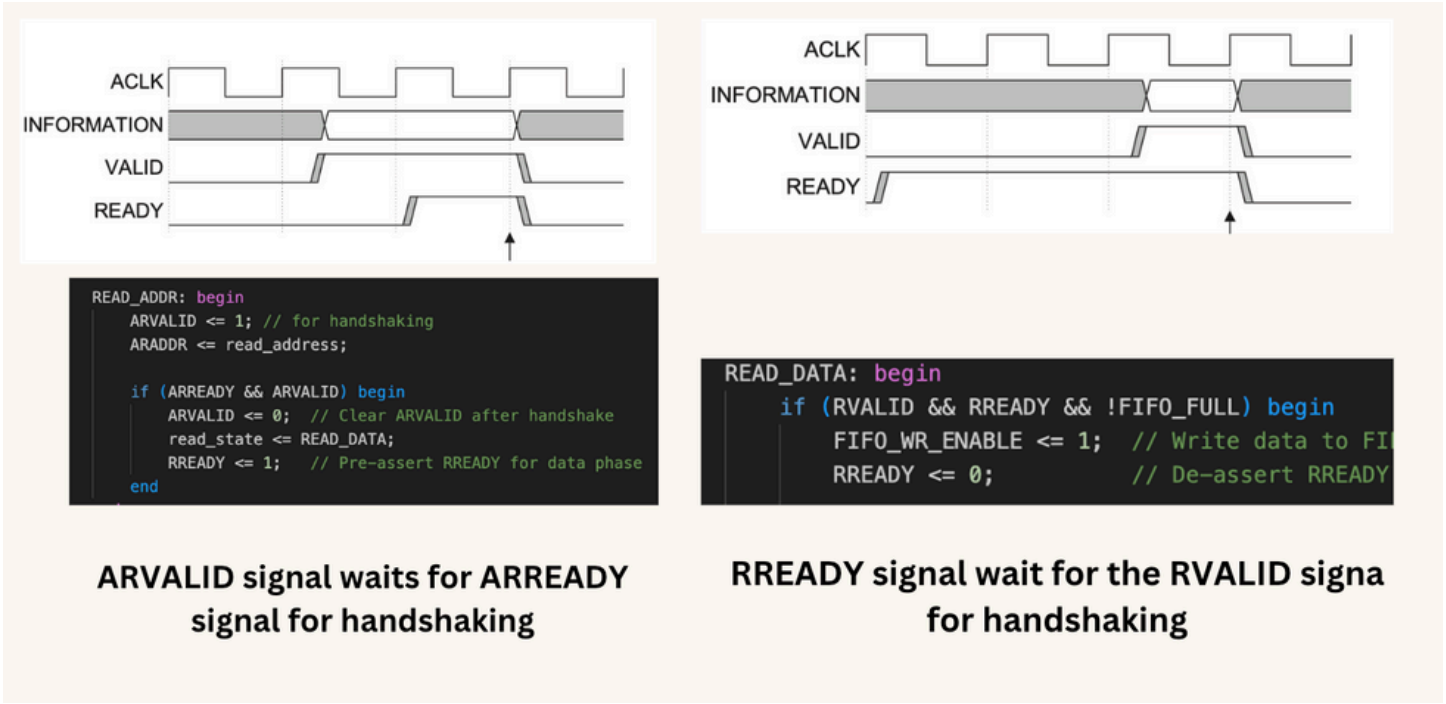
FIFO Usage

A 16-depth synchronous FIFO is used to buffer data between read and write transactions. The FIFO allows read and write operations to proceed independently, improving efficiency. The FIFO maintains:

- **FIFO_WR_ENABLE:** Signals when data should be written into the FIFO**.**
- **FIFO_RD_EN:** Signals when data should be read from the FIFO.
- **FIFO_EMPTY & FIFO_FULL:** Flags indicating FIFO status to prevent overflow or underflow**.**
- **FIFO_WR_PTR & FIFO_RD_PTR:** Pointers for write and read operations to manage FIFO entries.



AXI-Lite Handshaking Methodology



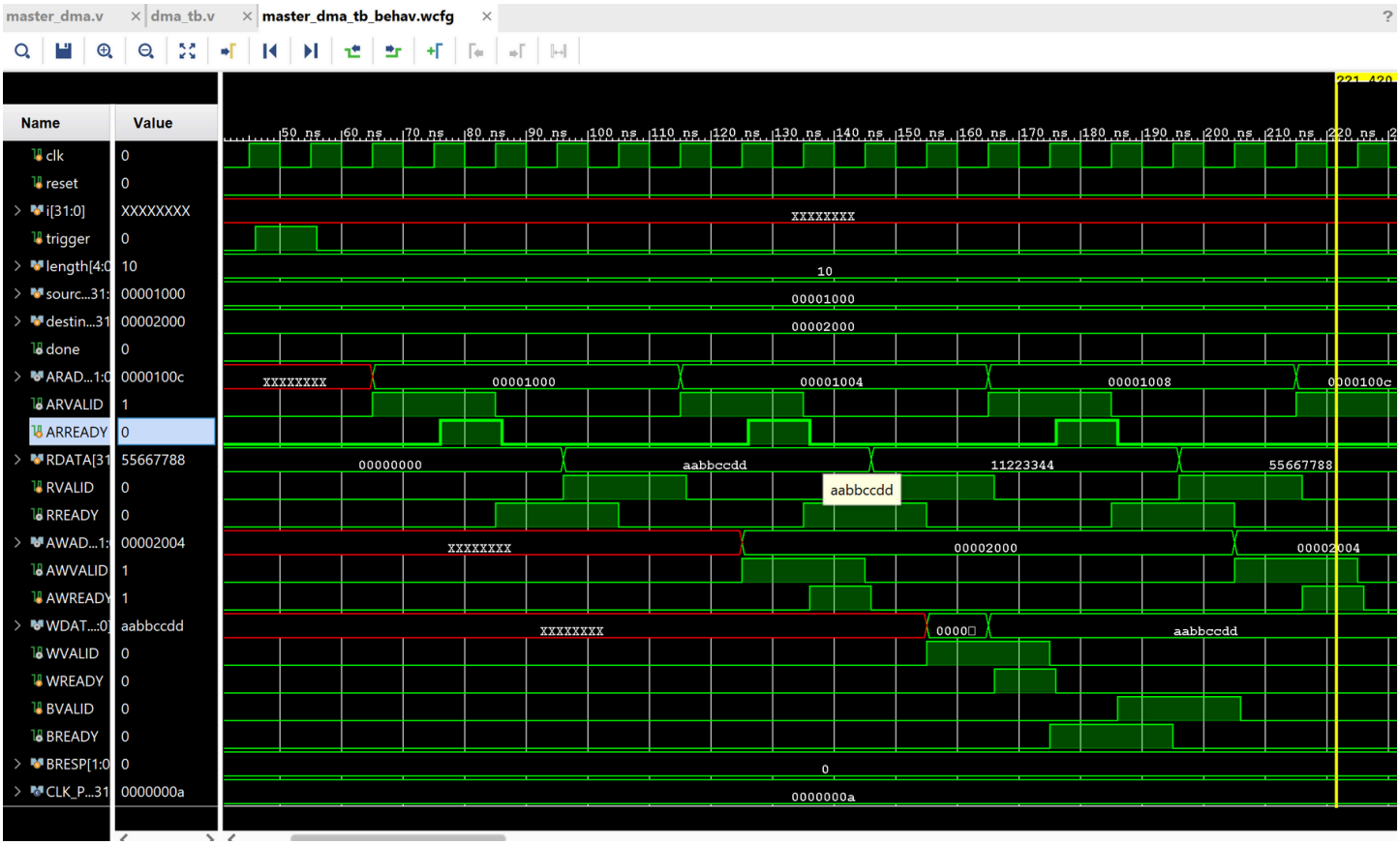
The module follows the AXI-Lite protocol for reliable data transfers:

1. **Read Address Channel (ARADDR, ARVALID, ARREADY)**
   - The module asserts ARVALID with the read address (ARADDR).
   - It waits for ARREADY from the AXI slave before proceeding to the read data phase.
2. **Read Data Channel (RDATA, RVALID, RREADY)**
   - The slave asserts RVALID when data is ready.
   - The module asserts RREADY to receive data and stores it in FIFO if not full.
3. **Write Address Channel (AWADDR, AWVALID, AWREADY)**
   - The module asserts AWVALID with the write address (AWADDR).
   - It waits for AWREADY before proceeding to the data phase.

4. **Write Data Channel (WDATA, WVALID, WREADY)**
   - The module asserts WVALID with data (WDATA) read from the FIFO.
   - It waits for WREADY before proceeding to the write response phase.
5. **Write Response Channel (BVALID, BREADY, BRESP)**
   - The slave asserts BVALID with a response.
   - The module asserts BREADY to acknowledge the response and transition the state machine.

This handshaking ensures that data transfers occur correctly without collisions or data loss.

SIMULATION:



PS2

UNALIGNED DMA TRANSFER

1. Overview

A Verilog-based AXI-Lite DMA controller that transfers data between arbitrarily aligned source and destination addresses. Handles unaligned reads/writes and arbitrary byte lengths using a 16-entry FIFO.

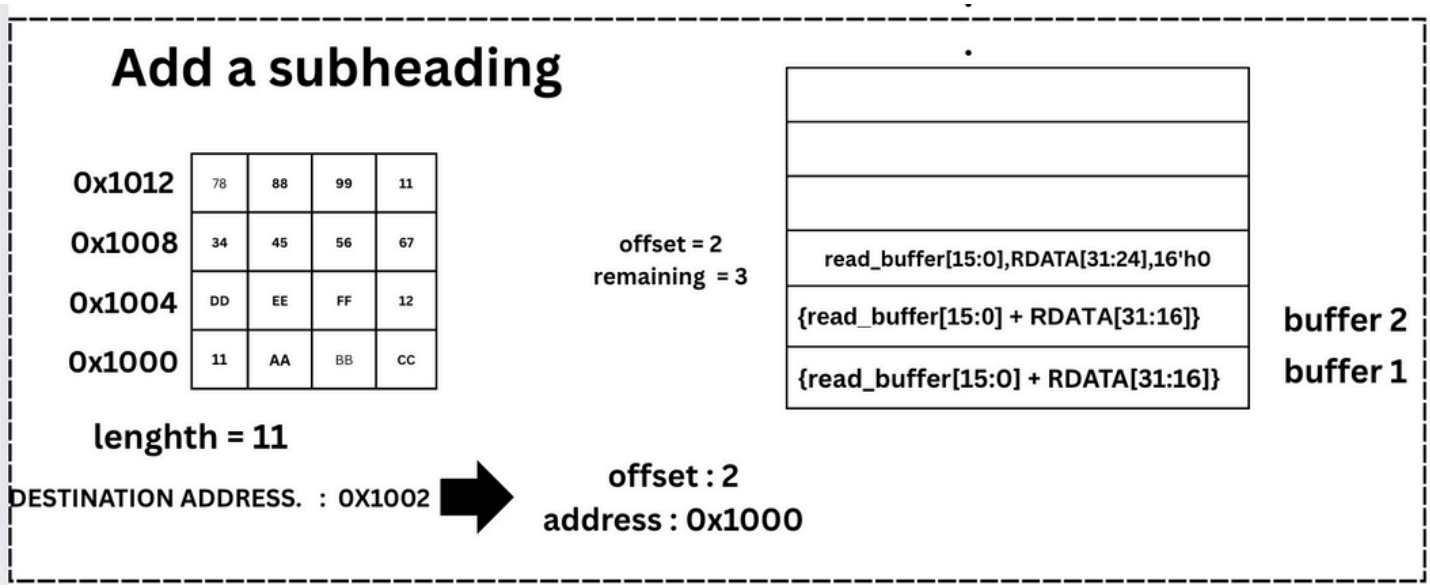
Introducing READ_PACK and WRITE_PACK states

- **READ_PACK** : if the transfering length is not a multiple fo r4 then it packs the unaligned length in fifo as **FIFO** requires 32 bit data.

```
case (offset)
    2'b00: begin
        if (remaining == 1) packed_word = {buffer[7:0], 24'h0};
        else if (remaining == 2) packed_word = {buffer[15:0], 16'h0};
        else if (remaining == 3) packed_word = {buffer[23:0], 8'h0};
        else packed_word = buffer;
    end
    2'b01: begin
        if (remaining == 1) packed_word = {buffer[15:8], 24'h0};
        else if (remaining == 2) packed_word = {buffer[23:8], 16'h0};
        else if (remaining == 3) packed_word = {buffer[31:8], 8'h0};
        else packed_word = {buffer[31:8], 8'h0};
    end
    2'b10: begin
        if (remaining == 1) packed_word = {buffer[23:16], 24'h0};
        else if (remaining == 2) packed_word = {buffer[31:16], 16'h0};
        else packed_word = {buffer[31:16], 16'h0};
    end
    2'b11: begin
        if (remaining == 1) packed_word = {buffer[31:24], 24'h0};
        else packed_word = {buffer[31:24], 24'h0};
    end
endcase
```

here remaning is the number of bytes that are left after all the bytes that are backet in 4Bytes(32) bit

for eg : for 11 Bytes and offset of 2



- **WRITE_PACK :** similarly write pack will handle packing and **write strobing** for last word transfer .

2. Key Innovations

- **Dual Offset Handling:** Supports both src_offset and dst_offset for unaligned source/destination addresses.
- **Dynamic WSTRB Generation:** Computes byte masks for partial writes at unaligned destinations.
- **Data Packing/Unpacking:** Combines unaligned reads into FIFO words and splits them for unaligned writes.

3. Core Features

Feature	Description
Unaligned Reads	Reads from addresses like 0x1002 by merging words (e.g., 0x3344AABB).
Unaligned Writes	Writes to addresses like 0x2003 using WSTRB masking (e.g., 4'b0001).
AXI-Lite Compliance	Implements handshaking (ARVALID/ARREADY, WVALID/WREADY).
FIFO Buffering	Decouples read/write operations for parallel transfers.

4. Design Breakdown

Helper Functions

- **align_to_word(addr)**: Strips lower 2 bits of addr to align to 4-byte boundaries.
  - Example: 0x1003 → 0x1000
- **get_offset(addr)**: Returns addr[1:0] to compute byte offset (0–3).
- **packed_word()**: Merges partial bytes into a 32-bit word based on offset and remaining bytes.

Read FSM

- **States**: IDLE → ADDR → DATA → PACK → DONE.
- **Logic**:
  - Reads aligned words (e.g., 0x1000 for src_addr=0x1002).
  - Combines partial data using src_offset (see table below).
  - Pushes aligned 32-bit words to FIFO.

src_offset	Data Merging Example
2'b01	{prev_word[23:0], new_word[31:24]}
2'b10	{prev_word[15:0], new_word[31:16]}

Write FSM

- **States**: IDLE → ADDR → DATA → PACK → RESP → DONE.
- **Logic**:
  - Writes to aligned addresses but shifts data using dst_offset.
  - Generates WSTRB dynamically for partial writes.

dst_offset	Data Shifting & WSTRB Example
2'b01	WDATA = {8'h0, FIFO_data[31:8]}; WSTRB=4'b0111
2'b11	WDATA = {24'h0, FIFO_data[31:24]}; WSTRB=4'b0001

FIFO

- **16-entry buffer** to decouple read/write operations , so that reading and writing canhappen at different instants.
- **Synchronous read/write** with full/empty flags.

5. Example: Unaligned Source & Destination

Transfer: src=0x1002, dest=0x2003, length=10 bytes.

Read Operations:

1. Read 0x1000 → Extract bytes 2–3: 0x3344.
2. Read 0x1004 → Merge with previous: 0x3344AABB (FIFO[0]).

Write Operations:

1. **First Write** (dest=0x2000):
   - Shift 0x3344AABB by dst_offset=3 → 0xBB000000.
   - WSTRB=4'b0001 (write only byte 3).
2. **Second Write** (dest=0x2004):
   - Shift remaining data → 0xAABBCCDD with WSTRB=4'b1110.

6. Testing Strategy

Corner Cases:

- src_offset/dst_offset = 1, 2, 3.
- length = 1, 4, 5, 17 bytes.

Simulation:

- Verify WSTRB for partial writes (e.g., 3 bytes at dst_offset=2).

This design efficiently handles real-world unaligned data (network packets, sensor outputs) with minimal CPU overhead.

The packed_word function uses case statements to modularize alignment logic, making it reusable across read and write FSMs.

Case Statements vs. Arithmetic/Shift Logic

Criteria	Case Statements	Arithmetic/Shift Logic
Readability	Clear and self-explanatory.	Needs manual bit-width calculations.
Maintainability	Easy to tweak for offsets.	Error prone
Hardware Efficiency	Uses multiplexers (low area).	Might need barrel shifters (high area).