EE405(B), Fall 2022

Electronics Design Lab. <Advanced Digital System Design>

- Student Names: Hyemin Lee
- Student IDs: 20190533
- LAB Project Number: Lab3

1. Goal / Specification

A. Goal of this lab is to learn about the role of FIFO and DMA, and to implement a simple DMA and verify its functionality with an FPGA.

B. [FIFO] First In, First Out (FIFO) is a method for organizing the manipulation of a data structure where the oldest entry, or "head" of the queue, is processed first.

C. [DMA] DMA is a separate engine to offload data movement between memory from CPU. DMA operates like this: First, CPU describes the transfer information (source address of memory, destination address of memory, transfer size of data to move, tec.) to the DMA. Second, when the CPU sends a start signal, DMA performs memory read/writes without help of CPU. Third, DMA finishing its works, DMA completes transfer by sending interrupt to CPU.

2. Architecture / Design

A. Description

i. Host-Kernel Communication (APB_master, APB_slave, and RTL kernel, DUT)

1. The host or testbench generates writing or reading instruction using tasks made in APB_master interface. Write instruction transfers source address, destination address, size of the data to APB_slave through APB bus. PADDR indicates the type of data, and PWDATA contains the value.

2. APB_slave stores data received from APB_master and deliver them to DMA. Also, it transfers interrupt and led output signal from DMA to APB_master. APB_slave and DMA are connected via DUT, and above that, RTL kernel (fgpa_top) connects DUT with two memories. Additionally, DUT assigns the memory whether it's source or destination memory.

ii. DMA

1. Input: source address, destination address, transfer size, mode, and read data from source and destination memories

2. Output: done flag, output leds, and memory related signals for source and destination memories (enable, write enable, address, write data)

3. States of DMA can be briefly divided into IDLE, OPERATION_READ, OPERATION_WRITE, and VERIFICATION. Since mode 1 means normal operation and mode 2 means verification, if the input mode is 1, the state

changes from IDLE to OPERATION_READ, OPERATION_WRITE, and IDLE, sequentially. Similarly, if the input mode is 2, the state changes from IDLE to VERIFICATION, and IDLE, sequentially. Several ENABLE and INPUT_ADDRESS sub-states are added to properly generate enable, write enable, and address signal to memory.

4. Operation Mode

   A. [Read from source memory] Since memory stores four individual data in a single line, DMA should generate the proper line address (*out_s_addr*) and send it to the memory to actually read the data requested from APB. After reading the whole line (*in_s_rddata*), DMA separates four data and push the targeted one to FIFO (*push_data*). By increasing *count_data_ff*, DMA reads data from 'source address' to 'source address + transfer size' and save them into FIFO sequentially.

   B. [Write to destination memory] DMA pops data from FIFO (*pop_data*) and write it on the destination memory (*out_d_wrdata*). Similar to the reading process, DMA generates proper line address (*out_d_addr*) and the order of data inside the line (*out_d_we*). The transfer size number of data will be popped and be written to the memory.

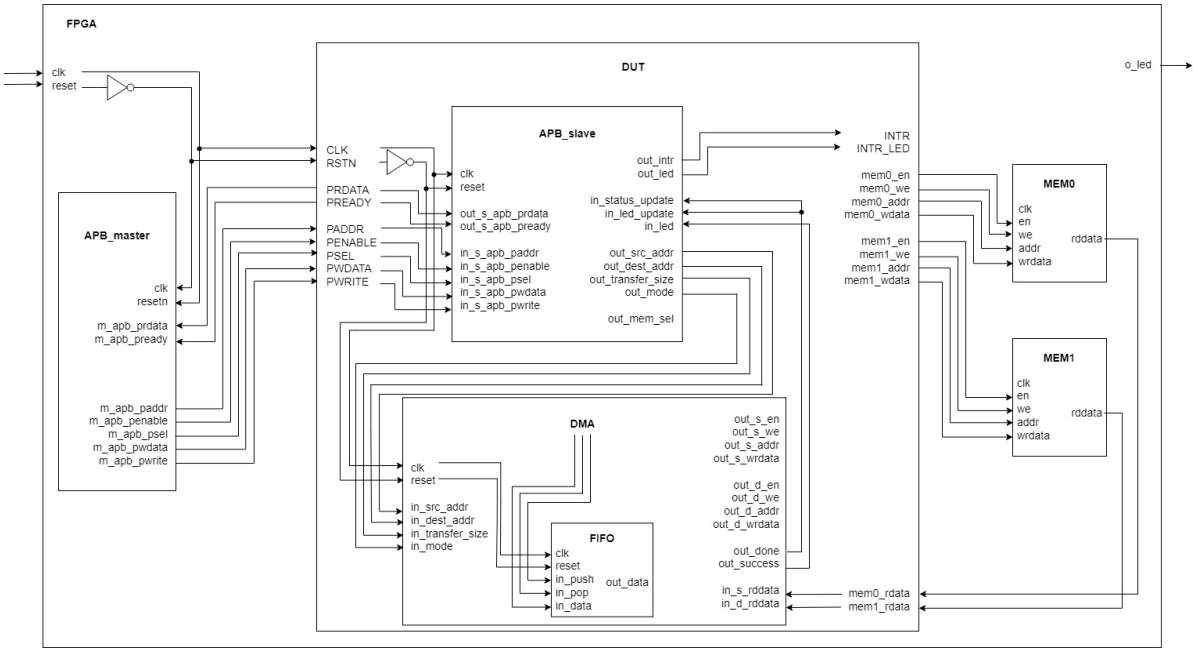
5. Verification mode

   A. DMA compares the data from source and destination memories and returns if they match (*out_success*). DMA generates the proper line address to read data both from source and destination memory (*out_s_addr, out_d_addr*). According to the order of the targeted data inside the line, the read data (*in_s_rddata, in_d_rddata*) is separated (*src_data, dest_data*) and compared with each other. If at least one data is different, 2 was returned to indicate fail. If every data was the same, 1 was returned indicating success.

iii. FIFO

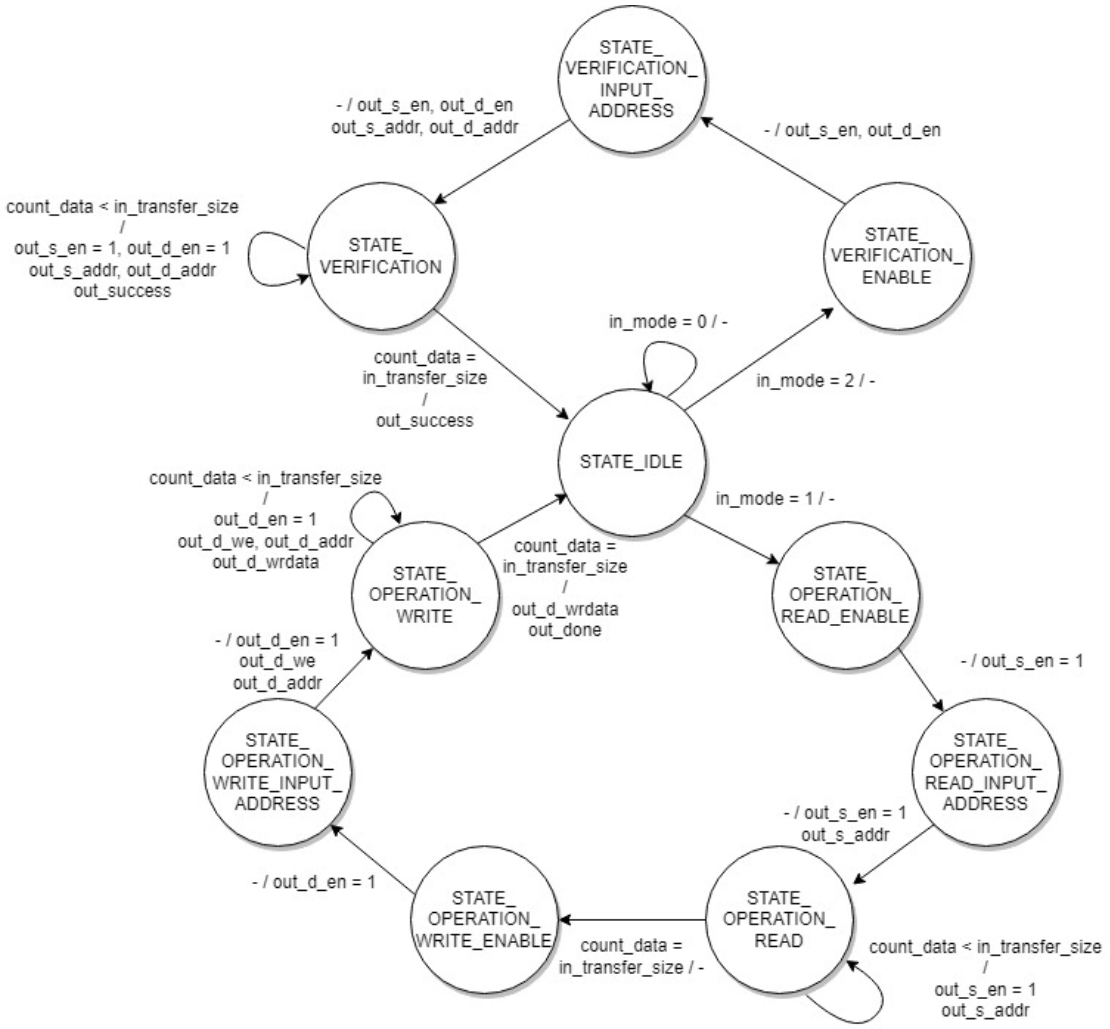
1. FIFO has two terminals. From the one side, data is being pushed (*in_push, in_data*) and saved inside the FIFO sequentially. From the other side, data is popped (*in_pop, out_data*) and removed from FIFO. *out_empty, out_almost_empty, out_full, out_almost_full* indicated the status of current FIFO, meaning that additional pushing popping is available.

B. Hardware Block Diagram



C. FSM

i. Description can be referred from 2. A. ii.



3. Experiment Results

A. Wave form of Problem 3A. FIFO

i. The figure above shows the waveform of FIFO design. It follows the following scenario, while the FIFO_AWIDTH = 13.

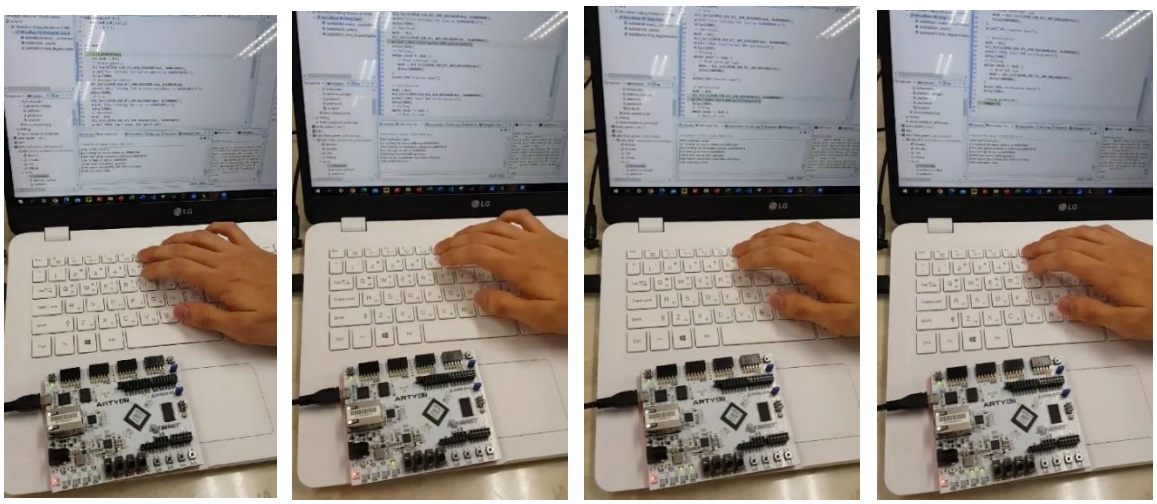
1. During 11 cycles, FIFO writes 11 times (0 to 10)
2. During one cycle, FIFO writes data (11), then ‘almost full’ is turned on.
3. During two cycles, FIFO writes once (12), then ‘almost full’ and ‘full’ are turned on.
4. During 8 cycles, read once and read/write 7 times simultaneously. (almost full)
5. During 10 cycles, read ten times.
6. During one cycle, read one time. (almost empty)
7. During last two cycles, read once, then ‘almost empty’ and ‘empty’ are turned on.

B. Wave form of Problem 3B. DMA (SRAM)

i. The figure below shows the testbench of DMA operation (SRAM).

C. Demo Picture of Problem 3C. DMA on FPGA

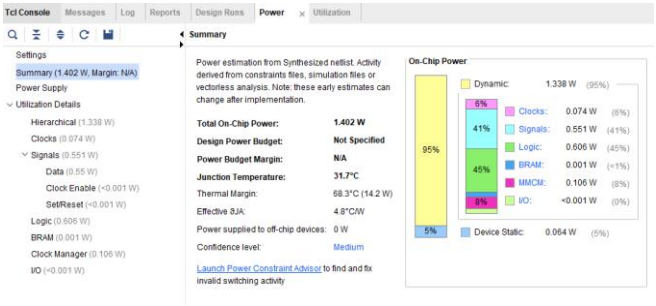
i. With the successful implementation, the LED signal is turned on like the figures above. During DMA transfer, LSB LED is turned on, and during DMA verification, LED[2] and LED[0] are turned on.



D. FPGA report

i. Utilization and power reports are showed like below. The utilization report analyzed the number of components used in the synthesis model, and the power report analyzed that of power consumption.

Resource	Utilization	Available	Utilization %
LUT	86381	20800	415.29
LUTRAM	65726	9600	684.65
FF	3337	41600	8.02
BRAM	2	50	4.00
IO	9	210	4.29
MMCM	1	5	20.00



4. Discussion

A. Question 1

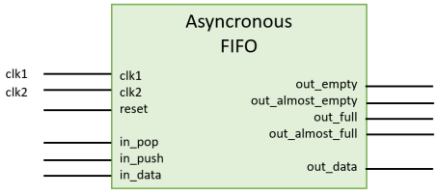
i. Description of APB_master, APB_slave, fpga_top, and DUT can be referred to 2. Architecture / Design > A. Description.

ii. [DMA, Operation Mode] If *in_mode = 1*, FSM state changes from STATE_IDEL to STATE_OPERATION_READ_~. First it enables source memory (*out_s_en_nxt = 1*) and insert the address to read the target data (*out_s_addr= (in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)/4)*. Since we should access memory as line and the line contains 4 data, the address was divided by 4. *count_data_ff* is increased until reaches *in_transfer_size*, saving the read data into FIFO (*push = 1, push_data = in_s_rddata[~:~]*). After reading all the data from source memory, the FSM state changes to STATE_OPERATION_WRITE_~. It enables destination memory (*out_d_en_nxt = 1*) with the proper write enabling signal (*out_d_we_nxt =* ~) which determines one of the 4 places where to write the data (*pop = 1, pop_data, out_d_wrdata*) inside the designated line (*out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff -1)/4*). In addition, *out_d_wrdata* was generated using *pop_data* with proper number of 0 biases (e.g. *{8'b0, pop_data, 16'b0}*) to match the position of the data inside the line.

iii. [DMA, Verification mode] Both source and destination memories are enabled and the line address was inserted to read from the both memories (*out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4; out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4;*). While increasing *count_data_ff* until *in_transfer_size*, individual data was compared to determine they are equivalent (*src_data = in_s_rddata[~:~], dest_data = in_d_rddata[~:~], if in_s_rddata != in_d_rddata*). If at least one data is different, 2 was returned (*out_success* = 2'b10) to indicate fail. If every data was the same, 1 was returned indicating success (*out_success = 2'b01*).

iv. [FIFO] If *in_push* is 1, *wr_pt_ff* increased by 1 and *in_data* was saved in the *wr_pt_ff%FIFO_AWIDTH* index of *FIFO_nxt*. If *in_pop* is 1, *rd_pt_ff* increased by 1 while always returning *FIFO_ff[rd_pt_ff%FIFO_AWIDTH]* as *out_data*. *out_empty, out_almost_empty, out_full,* and *out_almost_full* was determined by *nData = wr_pt_ff – rd_pt_ff*.

B. Question 2



i. [General case] Reading and writing in asynchronous FIFO is not a big deal. It can just be implemented as in the synchronous FIFO. However, returning empty and full sign should be modified. Generally, in synchronous FIFO, a counter is used to count the number of data inside the FIFO. The counter is increased with push, and decreased with pop instruction. In asynchronous FIFO, however, since push and pop happen in two different clock domains, counter cannot be used. So subtraction between write pointer and read pointer can be used. Here, since subtraction of two pointers are both 0 in empty and full case, additional bit should be added on the MSB side to both pointers. If wr_pt[n-1:0] == rd_pt[n-1:0] and wr_pt[n] == rd_pt[n], it means empty, and if wr_pt[n-1:0] == rd_pt[n-1:0] and wr_pt[n] != rd_pt[n], it means full.

ii. [My case] In my case, I already used subtraction of pointers to return empty and full signals of synchronous FIFO. But I didn't used additional MSB bit but provided sufficiently large bits to my pointers so that they can increase continuously. However, this method will cause error eventually since overflow will must happen after a large number of push and pop. So as explained in general case, my pointers should also be modified to only have one additional MSB bit and compare MSB and the other bits to determine empty and full condition.

C. Question 3

i. Since my DMA supports byte-enable operation, block diagram can be substituted by the 2. Architecture / Design > B. Hardware Block Diagram. Byte-enabled DMA reads data by line (*out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)/4*) but access the target data by indexing the read data (e.g. *target_data = in_s_rddata[15:8];*). When writing a data, set address by line (*out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff - 1)/4*) and set the proper write enable value according to the order of the target inside the line (e.g. *out_d_we_nxt = 4'b0010*). Writing data should be also properly shifted to match the position inside the line (e.g. *out_d_wrdata = {16'b0, pop_data, 8'b0}*).

5. Appendix: Include your code here

A. fpga_top

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ----------------------------------------------------------------
// Filename      : fpga_top.sv
// Author        : Castlab
//             JaeUk Kim    <kju5789@kaist.ac.kr >
//             Donghyuk Kim <kar02040@kaist.ac.kr>
// ----------------------------------------------------------------
// Description    : This is the top module of the fpga.
//             The block design shell and your DUT design
//             declared here.
// -FHDR----------------------------------------------------------------
`timescale 1ns / 1ps

import dma_pkg::*;
module fpga_top
(
   input  logic                        diff_clock_rtl_clk_n,
   input  logic                        diff_clock_rtl_clk_p,
   input  logic [3:0]                     axi_gpio_tri_i,
   input  logic                        reset,
   input  logic                        usb_uart_rxd,
   output logic                         usb_uart_txd,
   output logic [3:0]                     o_led
);

/* TO DO: Logic declaration */
// Internal Clock, Reset Declaration
logic CLK;
logic RSTN;

// APB Bus Signal Declaration
logic                         psel;
logic                         penable;
logic                         pready;
logic                         pwrite;
logic [REG_ADDR_WIDTH -1:0]     paddr;
logic [REG_DATA_WIDTH -1:0]     pwdata;
logic [REG_DATA_WIDTH -1:0]     prdata;
logic                         pstrb;
logic                         pprot;
logic                         pslverr;

// Memory 0 Signal Declaration
logic                           mem0_en;
logic [MEM_STRB_WIDTH -1:0]       mem0_we;
logic [MEM_ADDR_WIDTH -1:0]       mem0_addr;
logic [MEM_DATA_WIDTH -1:0]       mem0_wdata;
logic [MEM_DATA_WIDTH -1:0]       mem0_rdata;

// Memory 1 Signal Declaration
logic                           mem1_en;
logic [MEM_STRB_WIDTH -1:0]       mem1_we;
logic [MEM_ADDR_WIDTH -1:0]       mem1_addr;
logic [MEM_DATA_WIDTH -1:0]       mem1_wdata;
logic [MEM_DATA_WIDTH -1:0]       mem1_rdata;

// Shell instantiation
/* TO DO: Declare your wrapper instance here. */
design_1_wrapper wrapper (
    .diff_clock_rtl_clk_n                      ( diff_clock_rtl_clk_n     ),
    .diff_clock_rtl_clk_p                      ( diff_clock_rtl_clk_p    ),
    .led_4bits_tri_o                         ( axi_gpio_tri_i       ),
    .reset                                  ( reset              ),
    .usb_uart_rxd                            ( usb_uart_rxd          ),
    .usb_uart_txd                            ( usb_uart_txd          ),
    .usr_rtl_apb_paddr                        ( paddr              ),
```

```
    .usr_rtl_apb_penable                        ( penable            ),
    .usr_rtl_apb_pprot                          (                      ),
    .usr_rtl_apb_prdata                         ( prdata            ),
    .usr_rtl_apb_pready                         ( pready            ),
    .usr_rtl_apb_psel                           ( psel              ),
    .usr_rtl_apb_pslverr                        (                      ),
    .usr_rtl_apb_pstrb                          (                      ),
    .usr_rtl_apb_pwdata                         ( pwdata            ),
    .usr_rtl_apb_pwrite                         ( pwrite            ),
    .usr_rtl_clk                                ( CLK              ),
    .usr_rtl_rst                                ( RSTN              )
);


// DUT module instantiation
DUT u_DUT (
   .CLK                             ( CLK              ),     // i
    .RSTN                            ( RSTN                ),     // i
    .INTR                            ( o_led[0]            ),     // o
   .INTR_LED                          ( o_led[2:1]          ),     // o

    .PSEL                            ( psel               ),     // i
    .PENABLE                         ( penable             ),     // i
    .PREADY                           ( pready             ),     // o
    .PWRITE                           ( pwrite             ),     // i
    .PADDR                           ( paddr               ),     // i
    .PWDATA                           ( pwdata             ),     // i
    .PRDATA                           ( prdata             ),     // o

   .mem0_en                          ( mem0_en            ),     // o
   .mem0_we                          ( mem0_we            ),     // o
   .mem0_addr                          ( mem0_addr            ),     // o
   .mem0_wdata                          ( mem0_wdata          ),     // o
   .mem0_rdata                          ( mem0_rdata          ),     // i

   .mem1_en                          ( mem1_en            ),     // o
   .mem1_we                          ( mem1_we            ),     // o
   .mem1_addr                          ( mem1_addr            ),     // o
   .mem1_wdata                          ( mem1_wdata          ),     // o
   .mem1_rdata                          ( mem1_rdata          )     // i
);

// SRAM for your code
// Alternate this code with your bram instance after bram initialization.
// Note that you should use BRAM after simulation. (Synthesis...)
//Sram #(
//   .AWIDTH                           ( MEM_ADDR_WIDTH       ),
//   .DWIDTH                           ( MEM_DATA_WIDTH       ),
//   .WSTRB                           ( MEM_STRB_WIDTH       ),
//   .INIT_FILE                         ( "mem0_init.txt"       )
//) u_mem0 (
//   .clk                             ( CLK              ),
//   .en                               ( mem0_en             ),
//   .we                               ( mem0_we             ),
//   .addr                            ( mem0_addr            ),
//   .wrdata                           ( mem0_wdata          ),
//   .rddata                           ( mem0_rdata          )
//);

//Sram #(
//   .AWIDTH                           ( MEM_ADDR_WIDTH       ),
//   .DWIDTH                           ( MEM_DATA_WIDTH       ),
//   .WSTRB                           ( MEM_STRB_WIDTH       ),
//   .INIT_FILE                         ( "mem1_init.txt"       )
//) u_mem1 (
//   .clk                             ( CLK              ),
//   .en                               ( mem1_en             ),
//   .we                               ( mem1_we             ),
//   .addr                            ( mem1_addr               ),
//   .wrdata                           ( mem1_wdata          ),
//   .rddata                           ( mem1_rdata          )
//);
```

```
// BRAM instantiation
/* TO DO: Instantiate your BRAM for synthesis
       Note that you could use BRAM for simulation result */
blk_mem_gen_0 u_mem0
(
    .clka         (CLK        ),
    .ena          (mem0_en     ),
    .wea          (mem0_we     ),
    .addra        (mem0_addr   ),
    .dina         (mem0_wdata  ),
    .douta        (mem0_rdata  )
);

blk_mem_gen_1 u_mem1
(
    .clka         (CLK        ),
    .ena          (mem1_en     ),
    .wea          (mem1_we     ),
    .addra        (mem1_addr   ),
    .dina         (mem1_wdata  ),
    .douta        (mem1_rdata  )
);

Endmodule
```

B. DUT

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : DUT.sv
// Author       : Castlab
//             JaeUk Kim    < kju5789@kaist.ac.kr >
//             Donghyuk Kim < kar02040@kaist.ac.kr>
// ---------------------------------------------------------------------
// Description: DMA Top Design
//          In this module, you have to declare APB slave
//          and DMA module.
//          Luckily, APB slave module is provided... :>
// -FHDR----------------------------------------------------------------
`timescale 1ns / 1ps

import dma_pkg::*;
module DUT
(
   // Global
    input  logic                       CLK,
    input  logic                       RSTN,

   // 1: Done, 0: Not Done
   output logic                        INTR,

   // 00: Idle, 10: Success, 01: Fail
   output logic [2           -1:0]          INTR_LED,

   // APB Bus Signal
   input  logic                       PSEL,
   input  logic                       PENABLE,
   output logic                       PREADY,
   input  logic                       PWRITE,
   input  logic [REG_ADDR_WIDTH -1:0]          PADDR,
   input  logic [REG_DATA_WIDTH -1:0]          PWDATA,
   output logic [REG_DATA_WIDTH -1:0]          PRDATA,

   // Memory 0 Signal
   output logic                        mem0_en,
   output logic [MEM_STRB_WIDTH -1:0]          mem0_we,
   output logic [MEM_ADDR_WIDTH -1:0]          mem0_addr,
   output logic [MEM_DATA_WIDTH -1:0]          mem0_wdata,
   input  logic [MEM_DATA_WIDTH -1:0]          mem0_rdata,

   // Memory 1 Signal
```

```
   output logic                          mem1_en,
   output logic [MEM_STRB_WIDTH -1:0]          mem1_we,
   output logic [MEM_ADDR_WIDTH -1:0]          mem1_addr,
   output logic [MEM_DATA_WIDTH -1:0]          mem1_wdata,
   input  logic [MEM_DATA_WIDTH -1:0]          mem1_rdata

);

// DMA signal declaration
logic [REG_DATA_WIDTH   -1:0]               w_src_addr;          // DMA Source Address
logic [REG_DATA_WIDTH   -1:0]               w_dest_addr;        // DMA Destination Address
logic [REG_DATA_WIDTH   -1:0]               w_transfer_size;     // DMA Transfer Size
logic [MODE           -1:0]                w_mode;              // DMA Operation Mode, 00: Idle,
01: Normal Mode Start, 10: Test Mode Start
logic                                 w_status_update;     // DMA done
logic [2          -1:0]                  w_led;

// Memory signal declaration
logic                                  w_mem_sel;           // 0: Mem0-Source, Mem1-Destination, 1:
Mem0-Source, Mem1-Destination

// Source memory
logic                                 s_en;
logic [MEM_STRB_WIDTH   -1:0]               s_we;
logic [MEM_ADDR_WIDTH   -1:0]               s_addr;
logic [MEM_DATA_WIDTH   -1:0]               s_wrdata;
logic [MEM_DATA_WIDTH   -1:0]               s_rddata;

// Destination memory
logic                                 d_en;
logic [MEM_STRB_WIDTH   -1:0]               d_we;
logic [MEM_ADDR_WIDTH   -1:0]               d_addr;
logic [MEM_DATA_WIDTH   -1:0]               d_wrdata;
logic [MEM_DATA_WIDTH   -1:0]               d_rddata;

// Input, Output Signal Interconnect
/* TO DO: Connect mem0 and mem1 signal according to the w_mem_sel signal.
       Note that there exists 2 possible cases (source, dest) = (mem0, mem1), (mem1, mem0). */
always_comb begin
   if (w_mem_sel==1'b0) begin               // Mem0: Source, Mem1: Destination
      /* TO DO: Fill out the code */
      mem0_en = s_en;
      mem0_we = s_we;
      mem0_addr = s_addr;
      mem0_wdata = s_wrdata;
      s_rddata = mem0_rdata;
      mem1_en = d_en;
      mem1_we = d_we;
      mem1_addr = d_addr;
      mem1_wdata = d_wrdata;
      d_rddata = mem1_rdata;
   end
   else begin                            // Mem0: Destination, Mem1: Source
      /* TO DO: Fill out the code */
      mem1_en = s_en;
      mem1_we = s_we;
      mem1_addr = s_addr;
      mem1_wdata = s_wrdata;
      s_rddata = mem1_rdata;
      mem0_en = d_en;
      mem0_we = d_we;
      mem0_addr = d_addr;
      mem0_wdata = d_wrdata;
      d_rddata = mem0_rdata;
   end
end

// APB Slave Instantiation
APB_slave u_APB_slave (
   .clk                              ( CLK               ),
   .reset                              ( ~RSTN               ),

   .out_intr                            ( INTR              ),
```

```
   .out_led                          ( INTR_LED            ),

   .in_s_apb_psel                     ( PSEL               ),
   .in_s_apb_penable                  ( PENABLE            ),
   .out_s_apb_pready                  ( PREADY             ),
   .in_s_apb_pwrite                   ( PWRITE             ),
   .in_s_apb_paddr                     ( PADDR               ),
   .in_s_apb_pwdata                   ( PWDATA             ),
   .out_s_apb_prdata                   ( PRDATA             ),

   .out_src_addr                      ( w_src_addr          ),
   .out_dest_addr                      ( w_dest_addr          ),
   .out_transfer_size                   ( w_transfer_size       ),
   .out_mode                         ( w_mode             ),
   .in_status_update                   ( w_status_update       ),
   .in_led_update                     ( w_status_update       ),
    .in_led                           ( w_led               ),

   .out_mem_sel                       ( w_mem_sel            )
);

/* TO DO: Connect the port of DMA. */
// DMA Instantiation
DMA u_DMA (
   .clk                            ( CLK               ),
   .reset                            ( ~RSTN               ),

   .in_src_addr                       ( w_src_addr          ),
   .in_dest_addr                       ( w_dest_addr          ),
   .in_transfer_size                    ( w_transfer_size       ),
   .in_mode                          ( w_mode             ),
   .out_done                          ( w_status_update       ),

   .out_s_en                          ( s_en               ),
   .out_s_we                          ( s_we               ),
   .out_s_addr                          ( s_addr             ),
   .out_s_wrdata                       ( s_wrdata            ),
   .in_s_rddata                        ( s_rddata            ),

   .out_d_en                          ( d_en               ),
   .out_d_we                          ( d_we               ),
   .out_d_addr                          ( d_addr             ),
   .out_d_wrdata                       ( d_wrdata            ),
   .in_d_rddata                        ( d_rddata            ),

   .out_success                       ( w_led               )
);

Endmodule
```

C. DMA

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : DMA.sv
// Author       : Castlab
//                                  JaeUk Kim      < kju5789@kaist.ac.kr >
//                                  Donghyuk Kim < kar02040@kaist.ac.kr>
// ---------------------------------------------------------------------
// Description: DMA Design
//            This module is one of the main module that you have to design.
//            Design your own DMA on this file.
//            You could write your code from the scratch, but make sure to
//            match the specification.
// -FHDR----------------------------------------------------------------

`timescale 1ns / 1ps

import dma_pkg::*;
module DMA
(
```

```
    input  logic                                                                clk,
    input  logic                                                                reset,

         // APB Slave
    input  logic [REG_DATA_WIDTH -1:0]                  in_src_addr,
    input  logic [REG_DATA_WIDTH -1:0]                  in_dest_addr,
    input  logic [REG_DATA_WIDTH -1:0]                  in_transfer_size,
    input  logic [MODE          -1:0]                   in_mode,

         output logic                                                                   out_done,

         // Memory
         output logic                                                                   out_s_en,
         output logic [MEM_STRB_WIDTH -1:0]             out_s_we,
         output logic [MEM_ADDR_WIDTH -1:0]             out_s_addr,
         output logic [MEM_DATA_WIDTH -1:0]             out_s_wrdata,
         input  logic [MEM_DATA_WIDTH -1:0]             in_s_rddata,

         output logic                                                                   out_d_en,
         output logic [MEM_STRB_WIDTH -1:0]             out_d_we,
         output logic [MEM_ADDR_WIDTH -1:0]             out_d_addr,
         output logic [MEM_DATA_WIDTH -1:0]             out_d_wrdata,
         input  logic [MEM_DATA_WIDTH -1:0]             in_d_rddata,

         output logic [2                               -1:0]              out_success
);

/* TO DO: Declare your logic here. */
// Declare Registers
reg [REG_DATA_WIDTH -1:0]
in_src_addr_ff,in_dest_addr_ff,in_src_addr_nxt,in_dest_addr_nxt;
reg [REG_DATA_WIDTH -1:0]             in_transfer_size_ff,in_transfer_size_nxt;
reg                                                                  out_done_ff,
out_done_nxt;
reg [2                             -1:0]              out_success_ff, out_success_nxt;
reg                                                                  out_s_en_ff,
out_s_en_nxt;
reg                                                                  out_d_en_ff,
out_d_en_nxt;
reg [MEM_STRB_WIDTH -1:0]             out_d_we_ff, out_d_we_nxt;

reg [MEM_ADDR_WIDTH -1:0]             count_data_ff, count_data_nxt;
reg                             push, pop, is_empty, is_almost_empty, is_full, is_almost_full;
reg [8 -1:0]                  push_data, pop_data, src_data, dest_data;



// Assign output
assign out_done = out_done_ff;
assign out_success = out_success_ff;
assign out_s_en = out_s_en_ff;
assign out_s_we = 0;
assign out_s_wrdata = 0;
assign out_d_en = out_d_en_ff;
assign out_d_we = out_d_we_ff;

// Status
typedef enum logic [3:0] {
        STATE_IDLE,
        STATE_OPERATION_READ_ENABLE,
        STATE_OPERATION_READ_INPUT_ADDRESS,
        STATE_OPERATION_READ,
        STATE_OPERATION_WRITE_ENABLE,
        STATE_OPERATION_WRITE_INPUT_ADDRESS,
        STATE_OPERATION_WRITE,
        STATE_VERIFICATION_ENABLE,
        STATE_VERIFICATION_INPUT_ADDRESS,
        STATE_VERIFICATION
} StatusType;
StatusType
        state_ff, state_nxt;
```

```
// Verification memory
/* TO DO: Design verification mode of DMA using this memory. */
logic [8                          -1:0]                                          Mem[0:(1<<4)-1];
logic [4                          -1:0]                                          mem_read_ptr;
logic [4                          -1:0]                                          mem_write_ptr;
logic
        read;
logic
        rst;

/* TO DO: Write sequential code for your DMA here. */
always_ff @(posedge clk) begin
   state_ff <= state_nxt;
   out_done_ff <= out_done_nxt;
   out_success_ff <= out_success_nxt;
   out_s_en_ff <= out_s_en_nxt;
   out_d_en_ff <= out_d_en_nxt;
   out_d_we_ff <= out_d_we_nxt;
   count_data_ff <= count_data_nxt;
   in_src_addr_ff <= in_src_addr_nxt;
   in_dest_addr_ff <= in_dest_addr_nxt;
   in_transfer_size_ff <= in_transfer_size_nxt;
end

/* TO DO: Write combinational code for your DMA here. */
always_comb begin
   if (reset) begin
       state_nxt = STATE_IDLE;
       out_done_nxt = 0;
       out_success_nxt = 0;
       out_s_en_nxt = 0;
       out_d_en_nxt = 0;
       out_d_we_nxt = 0;
       count_data_nxt = 0;
       in_src_addr_nxt = 0;
       in_dest_addr_nxt = 0;
       in_transfer_size_nxt = 0;
       push = 0;
       pop = 0;
       push_data = 0;
   end else begin
       state_nxt = state_ff;
       out_done_nxt = 0;
       out_success_nxt = 0;
       out_s_en_nxt = 0;
       out_d_en_nxt = 0;
       out_d_we_nxt = 0;
       count_data_nxt = 0;
       push = 0;
       pop = 0;
       push_data = 0;

       case (state_ff)
           STATE_IDLE: begin
               if (in_mode == 1) begin
                   state_nxt = STATE_OPERATION_READ_ENABLE;
               end
               else if (in_mode == 2 ) begin
                   state_nxt = STATE_VERIFICATION_ENABLE;
               end
           end

           STATE_OPERATION_READ_ENABLE: begin
               out_s_en_nxt = 1;
               state_nxt = STATE_OPERATION_READ_INPUT_ADDRESS;
           end

           STATE_OPERATION_READ_INPUT_ADDRESS: begin
               state_nxt = STATE_OPERATION_READ;
               out_s_en_nxt = 1;
               out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)/4;  // push the read
data in the NEXT cycle
```

```
            count_data_nxt = count_data_ff + 1;
        end

        STATE_OPERATION_READ: begin
            if (count_data_ff < in_transfer_size) begin
                out_s_en_nxt = 1;
                count_data_nxt = count_data_ff + 1;
                out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)/4;   // push the
read data in the NEXT cycle
                push = 1;
                case ((in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff-1)%4)
                    0: push_data = in_s_rddata[7:0];
                    1: push_data = in_s_rddata[15:8];
                    2: push_data = in_s_rddata[23:16];
                    3: push_data = in_s_rddata[31:24];
                endcase
            end else begin
                count_data_nxt = 0;
                push = 1;
                case ((in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff-1)%4)
                    0: push_data = in_s_rddata[7:0];
                    1: push_data = in_s_rddata[15:8];
                    2: push_data = in_s_rddata[23:16];
                    3: push_data = in_s_rddata[31:24];
                endcase
                state_nxt = STATE_OPERATION_WRITE_ENABLE;
            end
        end

        STATE_OPERATION_WRITE_ENABLE: begin
            out_d_en_nxt = 1;
            state_nxt = STATE_OPERATION_WRITE_INPUT_ADDRESS;
        end
        STATE_OPERATION_WRITE_INPUT_ADDRESS: begin
            out_d_en_nxt = 1;
            case ((in_dest_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)%4)
                0: out_d_we_nxt = 4'b0001;
                1: out_d_we_nxt = 4'b0010;
                2: out_d_we_nxt = 4'b0100;
                3: out_d_we_nxt = 4'b1000;
            endcase
            count_data_nxt = count_data_ff + 1;
            state_nxt = STATE_OPERATION_WRITE;
        end

        STATE_OPERATION_WRITE: begin
            if (count_data_ff < in_transfer_size) begin
                out_d_en_nxt = 1;
                case ((in_dest_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff)%4)
                    0: out_d_we_nxt = 4'b0001;
                    1: out_d_we_nxt = 4'b0010;
                    2: out_d_we_nxt = 4'b0100;
                    3: out_d_we_nxt = 4'b1000;
                endcase
                count_data_nxt = count_data_ff + 1;
                out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff - 1)/4;  // write
the popped data in the SAME cycle

                pop = 1;
                case ((in_dest_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff - 1)%4)
                    0: out_d_wrdata = {24'b0, pop_data};
                    1: out_d_wrdata = {16'b0, pop_data, 8'b0};
                    2: out_d_wrdata = {8'b0, pop_data, 16'b0};
                    3: out_d_wrdata = {pop_data, 24'b0};
                endcase
            end else begin
                count_data_nxt = 0;
                out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff - 1)/4;  // write
the popped data in the SAME cycle
                pop = 1;
                case ((in_dest_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff - 1)%4)
                    0: out_d_wrdata = {24'b0, pop_data};
                    1: out_d_wrdata = {16'b0, pop_data, 8'b0};
```

```
                2: out_d_wrdata = {8'b0, pop_data, 16'b0};
                3: out_d_wrdata = {pop_data, 24'b0};
            endcase
            state_nxt = STATE_IDLE;
            out_done_nxt = 1;
        end
    end

    // Verification
    STATE_VERIFICATION_ENABLE: begin
        out_s_en_nxt = 1;
        out_d_en_nxt = 1;
        state_nxt = STATE_VERIFICATION_INPUT_ADDRESS;
    end

    STATE_VERIFICATION_INPUT_ADDRESS: begin
        state_nxt = STATE_VERIFICATION;
        out_s_en_nxt = 1;
        out_d_en_nxt = 1;
        count_data_nxt = count_data_ff + 1;
        out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4;
        out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4;
    end

    STATE_VERIFICATION: begin
        if (count_data_ff < in_transfer_size) begin
            out_s_en_nxt = 1;
            out_d_en_nxt = 1;
            count_data_nxt = count_data_ff + 1;  // add one line

            case ((in_src_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff-1)%4)
                0: src_data = in_s_rddata[7:0];
                1: src_data = in_s_rddata[15:8];
                2: src_data = in_s_rddata[23:16];
                3: src_data = in_s_rddata[31:24];
            endcase
            case ((in_dest_addr[MEM_ADDR_WIDTH -1:0]+count_data_ff-1)%4)
                0: dest_data = in_d_rddata[7:0];
                1: dest_data = in_d_rddata[15:8];
                2: dest_data = in_d_rddata[23:16];
                3: dest_data = in_d_rddata[31:24];
            endcase

            out_s_addr = (in_src_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4;
            out_d_addr = (in_dest_addr[MEM_ADDR_WIDTH -1:0] + count_data_ff)/4;
            if (src_data != dest_data) begin
                count_data_nxt = 0;
                state_nxt = STATE_IDLE;
                out_done_nxt = 1;
                out_success_nxt = 2'b10;    // Fail
            end
        end else begin
            if (in_s_rddata != in_d_rddata) begin
                out_success_nxt = 2'b10;    // Fail
            end else begin
                out_success_nxt = 2'b01;       // Success
            end
            count_data_nxt = 0;
            state_nxt = STATE_IDLE;
            out_done_nxt = 1;

        end
    end
  endcase
 end
end

/* TO DO: Instantiate your FIFO design here. */
FIFO
#(
    .FIFO_AWIDTH                            ( 32                                ),       //  Maximum
2^32 data
```

```
    .FIFO_DWIDTH                            ( 8                                                 )
// Byte
)
fifo
(
    .clk                                   ( clk                                                ),
    .reset                                 ( reset                                              ),
    .in_push                               ( push                                    ),
    .in_pop                                ( pop                          ),
    .in_data                               ( push_data                                    ),
    .out_empty                             ( is_empty                                  ),
    .out_almost_empty                       ( is_almost_empty         ),
    .out_full                              ( is_full                                            ),
    .out_almost_full                        ( is_almost_full          ),
    .out_data                              ( pop_data                                     )
);

endmodule
```

D. FIFO

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : FIFO.sv
// Author       : Castlab
//                                  JaeUk Kim      < kju5789@kaist.ac.kr >
//                                  Donghyuk Kim < kar02040@kaist.ac.kr>
// ---------------------------------------------------------------------
// Description: FIFO module
//                               Design your own FIFO here.
// -FHDR----------------------------------------------------------------
`timescale 1ns / 1ps
module FIFO
#(
    parameter FIFO_AWIDTH                       = 13,
    parameter FIFO_DWIDTH                       = 32
)
(
    input  logic                                clk,
    input  logic                                reset,
    input  logic                                in_push,
    input  logic                                in_pop,
    input  logic [FIFO_DWIDTH-1:0]                 in_data,
    output logic                                out_empty,
    output logic                                out_almost_empty,
    output logic                                out_full,
    output logic                                out_almost_full,
    output logic [FIFO_DWIDTH-1:0]                 out_data
);

        /* TO DO: Design your FIFO here.
                          Your FIFO implementation should follow provided method.
                          Note that, in ASIC design, we usually use FIFO that
                          could directly use data, then, pop to prepare next data.
                          In other word, if you want d0 at cycle c0, just use out_data
                          of FIFO, then, pop so that at cycle c1, out_data=d1
                          (next data). */

    reg [FIFO_DWIDTH-1:0] FIFO_ff [FIFO_AWIDTH-1:0];
    reg [FIFO_DWIDTH-1:0] FIFO_nxt [FIFO_AWIDTH-1:0];
    reg [FIFO_AWIDTH-1:0] rd_pt_ff, wr_pt_ff, nData;
    reg [FIFO_AWIDTH-1:0] rd_pt_nxt, wr_pt_nxt;
    reg [FIFO_AWIDTH-1:0] i;

    assign nData = wr_pt_ff - rd_pt_ff;
    assign out_empty = (nData == 0) ? 1 : 0;
    assign out_almost_empty = (nData <= 1) ? 1 : 0;
    assign out_full = (nData == FIFO_AWIDTH) ? 1 : 0;
    assign out_almost_full = (nData >= (FIFO_AWIDTH-1)) ? 1 : 0;
    assign out_data = FIFO_ff[rd_pt_ff%FIFO_AWIDTH];
```

```
    always_ff @ (posedge clk) begin
        rd_pt_ff <= rd_pt_nxt;
        wr_pt_ff <= wr_pt_nxt;
        FIFO_ff <= FIFO_nxt;
    end

    always_comb begin
        if (reset) begin
            rd_pt_nxt = 0;
            wr_pt_nxt = 0;
        end else begin
            // Write
            if (in_push) begin
                if   (wr_pt_ff   >   0)   FIFO_nxt[(wr_pt_ff-1)%FIFO_AWIDTH]   =   FIFO_ff[(wr_pt_ff-
1)%FIFO_AWIDTH];
                FIFO_nxt[wr_pt_ff%FIFO_AWIDTH] = in_data;

                wr_pt_nxt = wr_pt_ff + 1;
            end

            // Read
            if (in_pop) begin
                rd_pt_nxt = rd_pt_ff + 1;
            end
        end
    end

endmodule
```

E. dma_pkg

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename      : dma_pkg.sv
// Author       : Castlab
//                                  JaeUk Kim      < kju5789@kaist.ac.kr >
//                                  Donghyuk Kim < kar02040@kaist.ac.kr>
// ---------------------------------------------------------------------
// Description: Declare your parameter here.
// -FHDR----------------------------------------------------------------

`timescale 1ns/1ps

package dma_pkg;
    parameter REG_ADDR_WIDTH                                                    = 32;
    parameter REG_DATA_WIDTH                                                    = 32;
          parameter MEM_ADDR_WIDTH                                              = 16;
          parameter MEM_DATA_WIDTH                                              = 32;
          parameter MEM_STRB_WIDTH                                              = MEM_DATA_WIDTH/8;
          parameter MODE                                                                                =
2;
          parameter MAX_TRANS_SIZE                                              = 5;
          // Maximum Transfer Size = 16 (Represent in 5byte)

          /* TO DO: Don't forget to modify this Base Address before synthesis.
                               If you forget, just spending 1hr for acknowledging your fault would be
lucky case. */
          parameter DMA_BASE_ADDR                                                                       =
32'h44A10000;

          /* TO DO: Declare your global parameter.
                               This is optional that depends on your RTL design. */

endpackage
```