HW#4 AXI4-Lite Device Bus Protocol

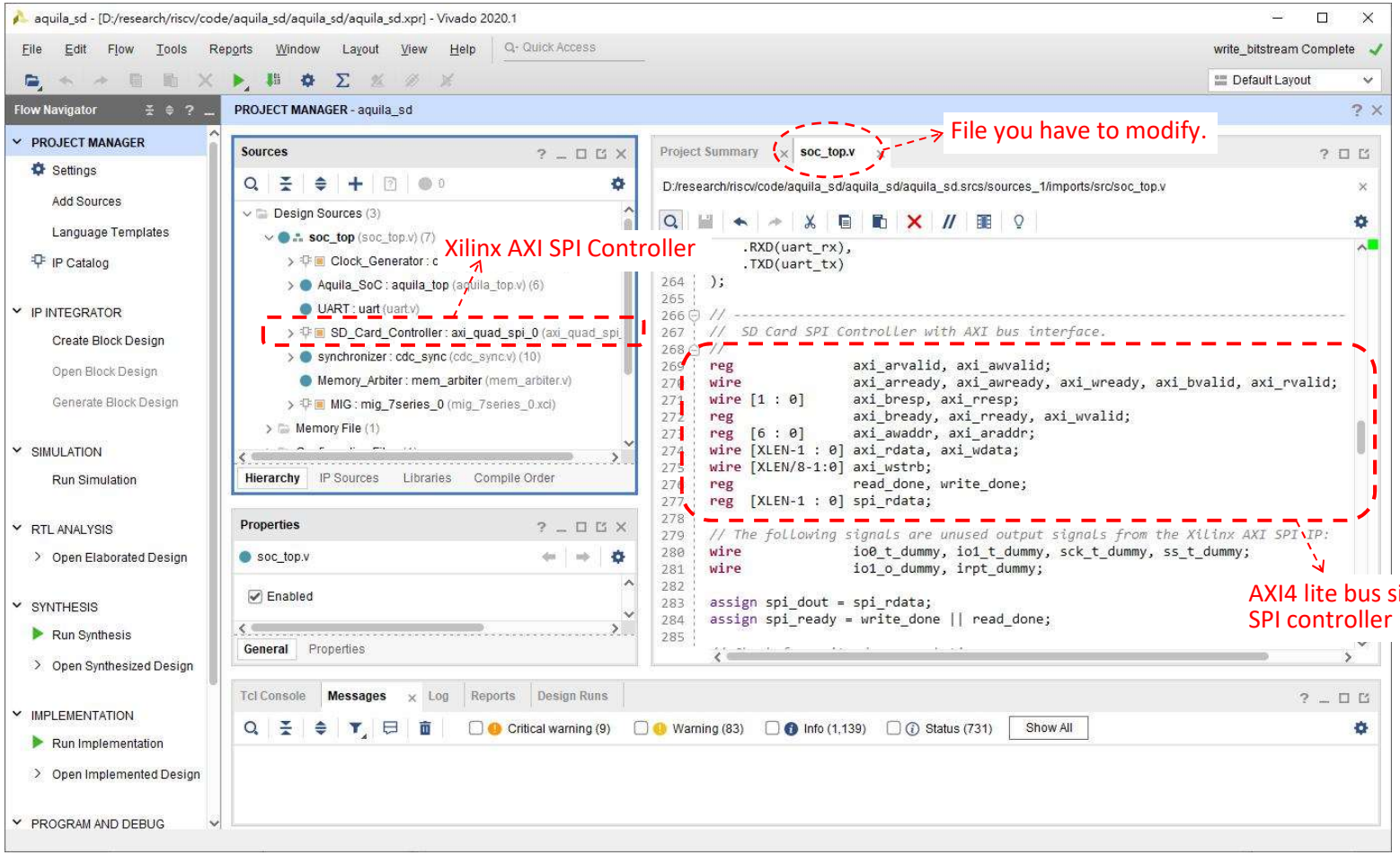
Chun-Jen Tsai
National Chiao Tung University
11/30/2021

Homework Goal

- Implement the AXI4-lite bus interface between Aquila and the SPI controller so that you can read a text file from the SD card
  - An Aquila workspace, `aquila_sd`, with the SPI controller can be downloaded from the E3 website
  - A software application, `read_text`, that reads a text file from an FAT32-formatted SD card is also available on E3
  - You must study the AXI4-Lite bus protocol to complete the task
- Upload your modified code to E3 by 12/13, 17:00 pm. Note that you do not have to write report for this HW.

The Aquila with SD Workspace

- Create the workspace using the TCL script, you see:



Prepare an SD Card for This HW

- The SD card that we use follows the SDHC standard, formatted with the FAT32 file system
  - Typically, SD card with capacity from 4G ~ 32G should be OK.
  - The logical structure is composed of 512-byte blocks, starting at block number 0
- You can put a ASCII text file, `test.txt`, on your card, in the root directory

Running `read_text.ebf`

- If you run the `read_text.ebf`, Aquila will stuck because the SD card cannot be initialized

```
================================================================
Copyright (c) 2019-2021, EISL@NCTU, Hsinchu, Taiwan.
The Aquila SoC is ready to go.
Waiting for an ELF/EBF file to be sent from the UART ...

Program entry point at 0x80000000, size = 0x4A28.
----------------------------------------------------------------
============================================================
 Try to display the text file 'test.txt' on the SD card...
============================================================
init SPI
```

- You must modify soc_top.v and properly setup the three AXI4 write bus channels to make it work!

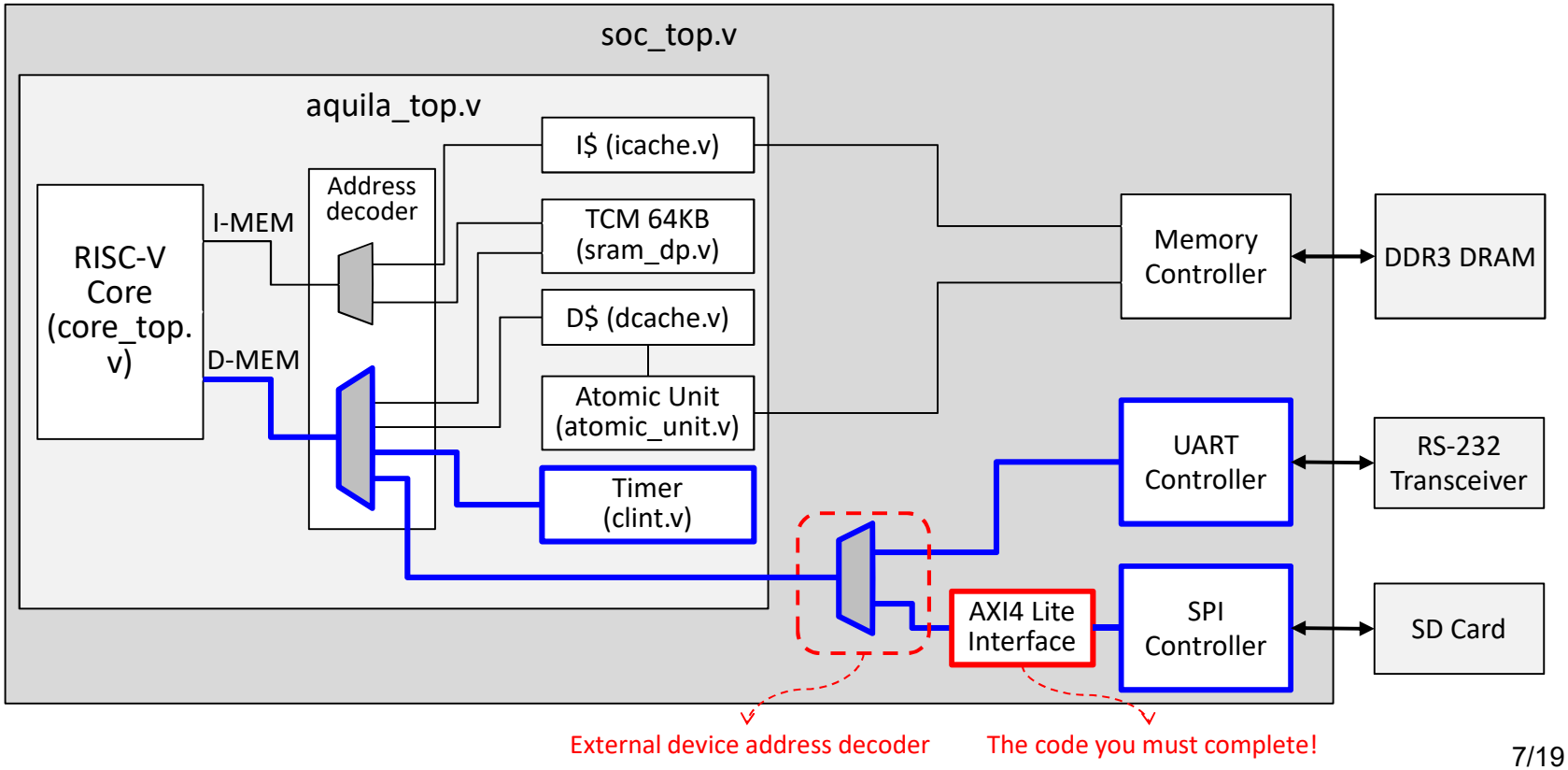
A Sample Bit File

- There is a sample bit file on E3 that has all the AXI4 bus channels properly connected to the SPI controller
  - Configure the FPGA with it, run `read_text.ebf` and you see:

```
============================================================
 Try to display the text file 'test.txt' on the SD card...
============================================================
init SPI
status: 0x25
status: 0x25
SPI initialized!
initializing SD...
SD command cmd0          response : 1
SD command cmd55         response : 1
SD command acmd41        response : 1
. . . . .
SD command cmd55         response : 1
SD command acmd41        response :
SD card initialized!

The texts in test.txt are as follows:
------------------------------------------------------------------------

The Hitch Hiker's Guide to the Galaxy
. . . . .
It begins with a house.

------------------------------------------------------------------------
Press <reset> on the FPGA board to reboot the cpu ...
```

Keys To the HW

- For this HW, you must understand how Aquila sends a read/write requests to a specific I/O device, and gets proper responses



I/O Address Decoder of Aquila

- A device I/O request will be directed to the right device by the I/O address decoder (in `soc_top.v`)
  - Note that the bus outputs, `dev_din`, `dev_we`, and `dev_be`, from Aquila to devices are driven by Aquila and feed to all I/O devices simultaneously
  - The I/O address decoder selects one of the devices and return its output, `dev_dout`, to Aquila

```
// ----------------------------------------------------------------------------
//  Device address decoder.
//
//       [0] 0xC000_0000 - 0xC0FF_FFFF : UART device
//       [1] 0xC200_0000 - 0xC2FF_FFFF : SPI device

assign uart_sel  = (dev_addr[XLEN-1:XLEN-8] == 8'hC0);
assign spi_sel   = (dev_addr[XLEN-1:XLEN-8] == 8'hC2);
assign dev_dout  = (uart_sel)? uart_dout : (spi_sel)? spi_dout : {XLEN{1'b0}};
assign dev_ready = (uart_sel)? uart_ready : (spi_sel)? spi_ready : {XLEN{1'b0}};
```

The AXI4 Lite Interface

- An AXI bus has five channels:
  - Read address bus (axi_araddr, axi_arvalid, axi_arready)
  - Read data bus (axi_rdata, axi_rvalid, axi_rready, axi_rresp)
  - Write address bus (axi_awaddr, axi_awvalid, axi_awready)
  - Write data bus (axi_wdata, axi_wvalid, axi_wready, axi_wstrb)
  - Write response bus (axi_bvalid, axi_bresp, axi_bready)
- In `soc_top.v`, the first two channels have been implemented, you must complete the write channels
  - That is, you must write code to control the red signals.

Code Template in `soc_top.v`

- Code template you need to fill in:

```
// ------------------------
//  Write Address Channel (INCOMPLETE)
// ------------------------
always @(posedge clk)
begin
    axi_awvalid <= 0;
end

always @(posedge clk) // Write Addresses
begin
    axi_awaddr <= 0;
end

// --------------------
//  Write Data Channel (INCOMPLETE)
// --------------------
always @(posedge clk)
begin
    axi_wvalid <= 0;
end

assign axi_wdata = {XLEN{1'b0}};     // write data from Aquila.
assign axi_wstrb = {(XLEN/8){1'b0}}; // byte-select for write operations.

// ---------------------------
//  Write Response (B) Channel (INCOMPLETE)
// ---------------------------
always @(posedge clk)
begin
    axi_bready <= 0;
end
```

AXI Bus Specification

- For your references, an AXI4 bus specification will be uploaded to E3:
  - ARM, *AMBA® AXI™ and ACE™ Protocol Specification*, ARM IHI 0022D, 2011.
- Other articles on the web might be useful for you too
  - Common AXI Themes on Xilinx's Forum (https://zipcpu.com/blog/2021/03/20/xilinx-forums.html)
  - The most common AXI mistake (https://zipcpu.com/formal/2019/04/16/axi-mistakes.html)

Accessing the SD Card

- SD cards support at least two I/O interfaces, SDIO and SPI. Here, we use the SPI interface to read data
- In addition to integrating a hardware controller, you also needs a software library to support reading a file from an FAT32 file system
  - The library contains three files:
    - `fat32.c`: handles FAT 32 file system structure
    - `sd.c`: issues SD controller commands to read data blocks
    - `spi.c`: low-level SPI I/O routines

About the Test Program

- The application `read_text.c` is as follows:
  - You can modify the linker script and code to avoid using DRAM
  - A circuit without DRAM synthesizes much faster

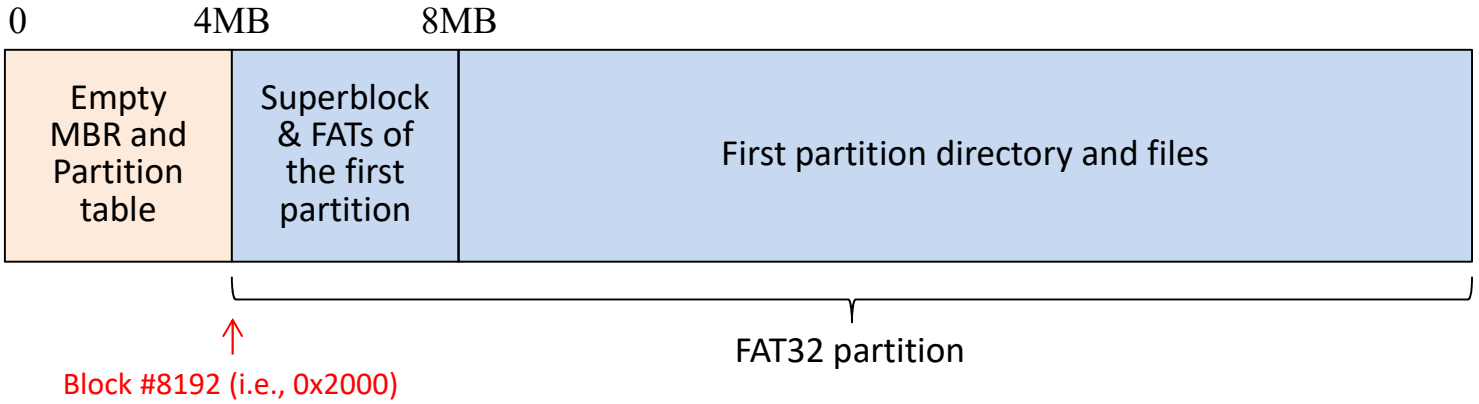
```
int main(void)
{
    uint32_t size;
    char *fname = "test.txt";
    uint8_t *fbuf  = (uint8_t *) 0x82000000L;

    printf("============================================================\n");
    printf(" Try to display the text file '%s' on the SD card...\n", fname);
    printf("============================================================\n");

    if ((size = read_file(fname, fbuf)) == 0)
    {
        printf("Cannot read the file '%s' on the SD card.\n", fname);
        return -1;
    }
    else
    {
        printf("\nThe texts in %s are as follows:\n", fname);
        printf("------------------------------------------------------------\n");
        for (int idx = 0; idx < size; idx++) putchar((char) fbuf[idx]);
        printf("------------------------------------------------------------\n");
        printf("Press <reset> on the FPGA board to reboot the cpu ...\n\n");
    }

    while (1);
}
```

Physical Structure and File Systems

- The logical structure of an SD card is simply composed of a sequence of 512-byte blocks
  - Physically, SD cards are divided into 4KB ~ 32KB clusters
- To create directories for file storage on the card, we must first partition the SD card and then format a logical file system on that partition
- An SD card usually has one partition.
  - It is possible to have multiple partitions and file systems on a single SD card
  - Our library only works on the first partition and FAT32

Disk Partitions

- Typical partition structure of an SDHC card:



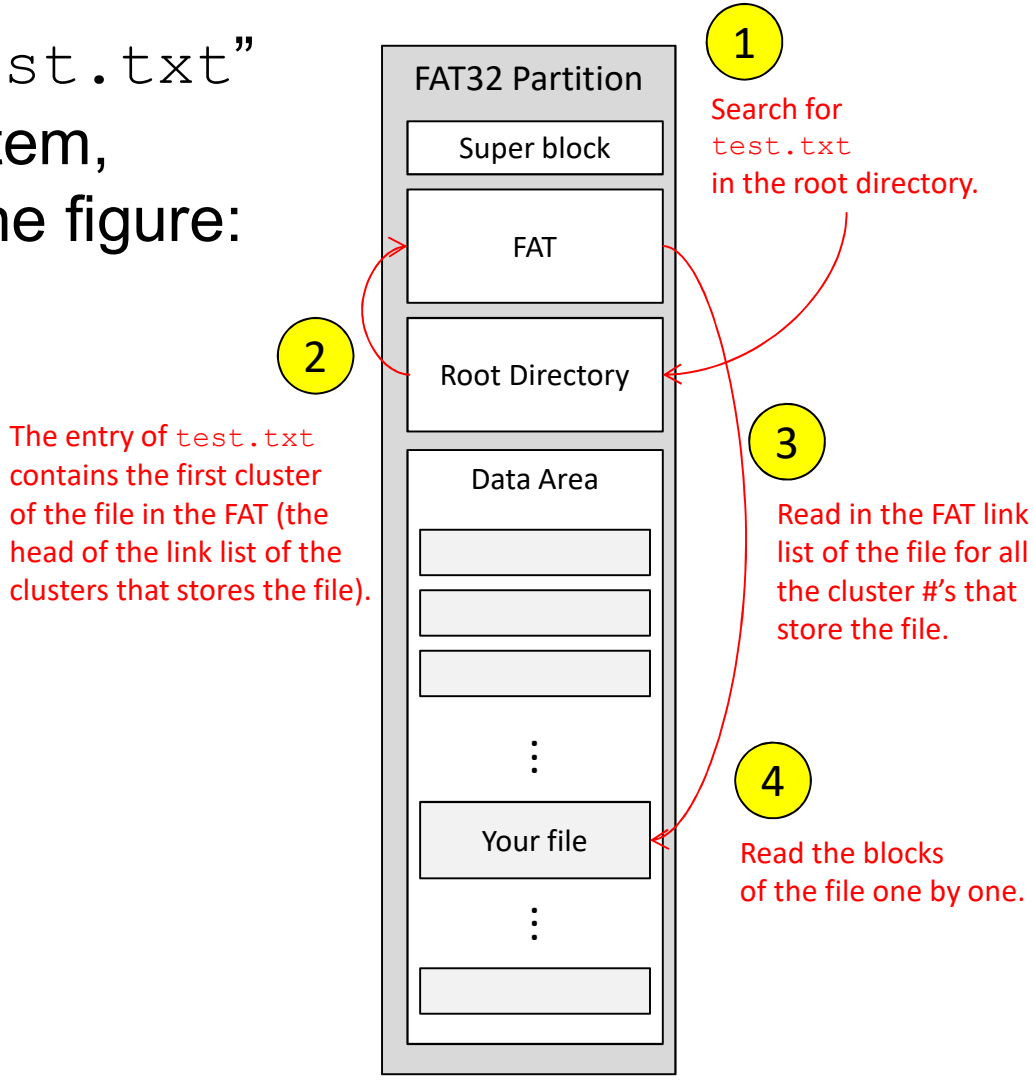
- If the card is bootable or has more than one partitions, then the Master Boot Record (MBR) and the partition table will not be empty

FAT32 File System Structure

- An FAT32 file system has the following components:
  - Boot sector: 512 bytes, possibly block#0, a.k.a. the super block
    - The boot sector contains information such as the size of the FAT, root directories, boot code, etc.
  - File allocation table (FAT): a table that shows which file is stored in which allocation units (each allocation unit is composed of several consecutive blocks); FAT basically contains many link lists of block numbers (one list per file)
  - Root directory: contains file names, file attributes, and the first allocation unit of all files in the root directory
    - Our library only read files from the root directory
  - Data area: the data blocks that actually store files

File Structure of an FAT32 Partition

- To read a file of "`\test.txt`" in an FAT32 file system, follow the steps in the figure:
- Each entry (link pointer) in the FAT is of 32-bit.



One Comments on File Name

- Although FAT32 support file names with up to 255 characters, our library only supports the classical 8+3 file names
  - 8+3 file name is a classical design from CP/M operating system, but many people call it DOS 8+3 file names
  - 8+3 file name do not distinguish between upper and lower case letters
- If you modify the file I/O library to support the FAT32 long file name, you can get bonus credit for the class!

Your Homework

- Complete the AXI4 lite bus interface for the SPI controller so the program `read_text` can read and display a text file on the SD card
- You do not have to up load report for this homework