Disk storage implementation for FPGA

Using ESP32!

Demetre Lataria

2024-12-3

Contents

Chapter 1

Introduction

Research explores implementation of disk storage for FPGA using a microcontroller through GPIO connection. Now we will go over the required equipment and software used for the implementation:

1.1 Hardware

1.1.1 FPGA board

Of course, we will need a fpga board. The one we will be using is Altera Cyclone V GX Starter Kit, however any fpga board with gpio headers can be used since implementation itself is in verilog.

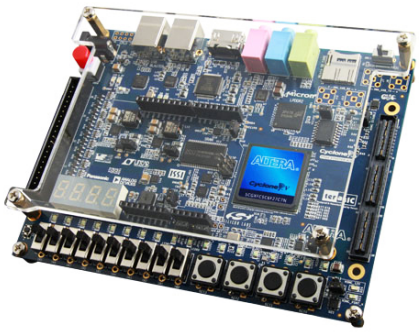

Figure 1.1 FPGA board.

1.1.2 Microcontroller

For a microcontroller I have chosen ESP32 for it's versatility and affordability. However, again, any microcontroller can be used, although code must be rewritten to be compatible with another microcontroller.

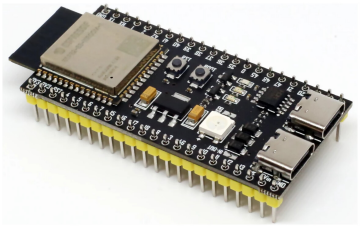

Figure 1.2 ESP32S3 Microcontroller.

1.1.3 Medium

We will also require some storage medium to operate with. For our purposes I have chosen SD card to serve as disk storage.

Not verified (Onboard slot on the FPGA board might be used but needs further testing)
In case microcontroller of your choice doesn't come with the slot for using SD card storage, you will also require SD module to interface with SD storage. I am using the following module:

Figure 1.3 Micro SD card and SD card module.

1.1.4 Miscellaneous

You will also require a lot of jumper cables so be prepared. Having such useful tools as multimeter or oscilloscope can be very helpful to test connections and diagnose any issues but are not required.

1.2 Software

1.2.1 FPGA software development kit

To program the fpga you will need the development suite of programs suitable for your FPGA board. In our case we use Intel's Quartus software. You can download it from the Intel's FPGA Software Download Center.

1.2.2 Microcontroller Integrated Development Environment

Programming microcontrollers can be done with many programs such as Arduino's IDE. We will be using ESP–IDF to program and monitor the ESP32. The IDE and instructions on how to set it up can be found in Espressif's GitHub Repository.

Chapter 2

Implementation

Now we will discuss the implementation itself.

2.1 Hardware Connections

2.1.1 SD Card protocol

We will be using SPI protocol to communicate to SD card from Microcontroller. It requires only four data wires:

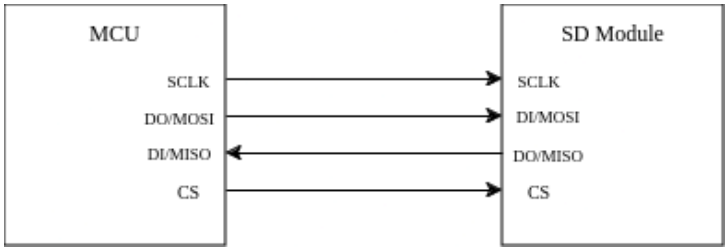

Figure 2.1 SPI protocol.

- SCK/SCLK: Clock Out signal.
- DO/MOSI: Master Out Slave In. One-directional signal that carries data from Master to Slave.
- DI/MISO: Master In Slave Out. One-directional signal that carries data from Slave to Master.
- CS: Chip Select. Used when we have multiple Slave modules on one SPI bus. Not required for our purposes.

2.1.2 Protocol between FPGA and Microcontroller

Now let's discuss the connections between FPGA and microcontroller. The following diagram defines data packet which is sent between FPGA board and ESP32:

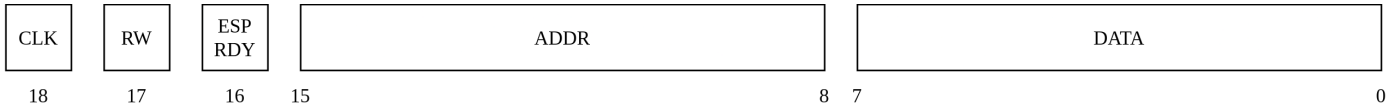

Figure 2.2 Protocol between ESP microcontroller and FPGA board.

We have the following elements in the data packet:

- DATA: 8-bit bidirectional wire that transmits data read from the storage to the FPGA or data that has to be written to ESP32.
- ADDR: 8-bit directional wire that transmits address value required to store or load data to/from the storage transmitted to the ESP32.
- RW: 1-bit flag that is 0 if read operation is requested or 1 if write operation is requested.
- ESP RDY: 1-bit flag that is 1 if ESP32 has completed the request and ready for new command.
- CLK: 1-bit wire that sends clock signal to the ESP32 to process requests on rising edge of the clock from the FPGA.

Here's the full diagram of how FPGA board and ESP32 are connected:

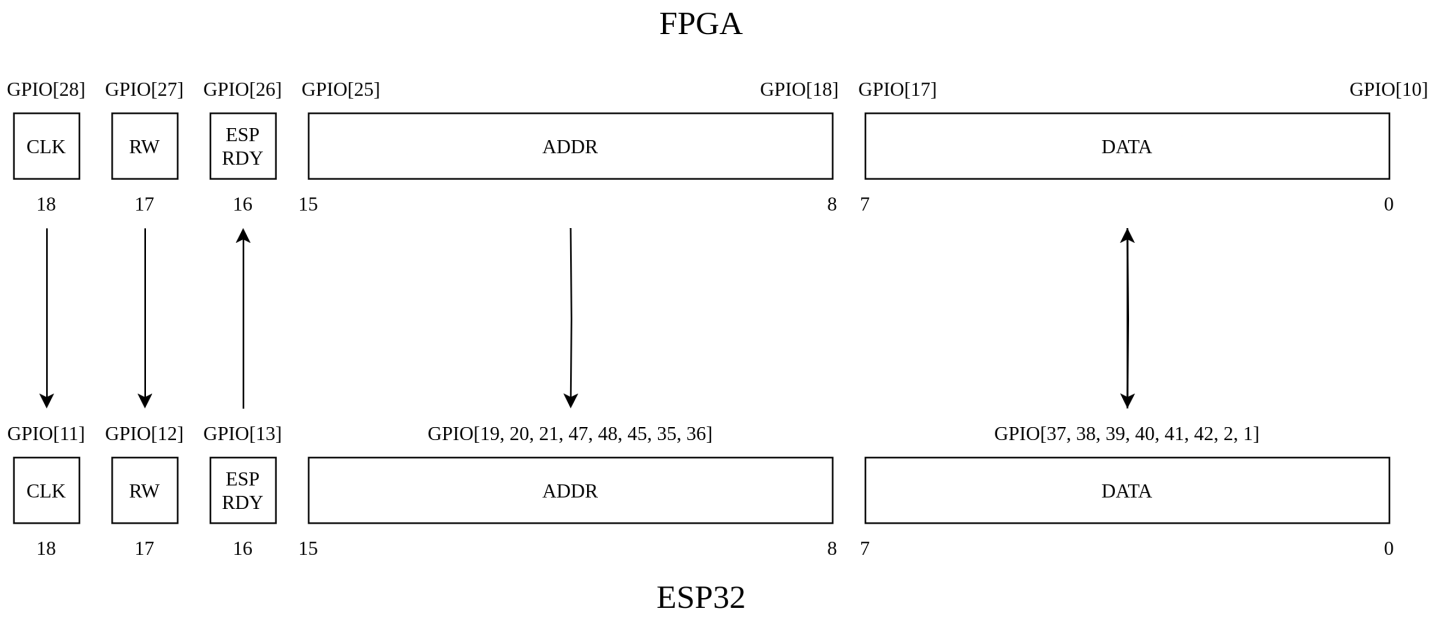

Figure 2.3 Signal directions in Microcontroller and FPGA protocol.

2.1.3 Full Electrical Diagram

This is the full diagram of all the components connected together.

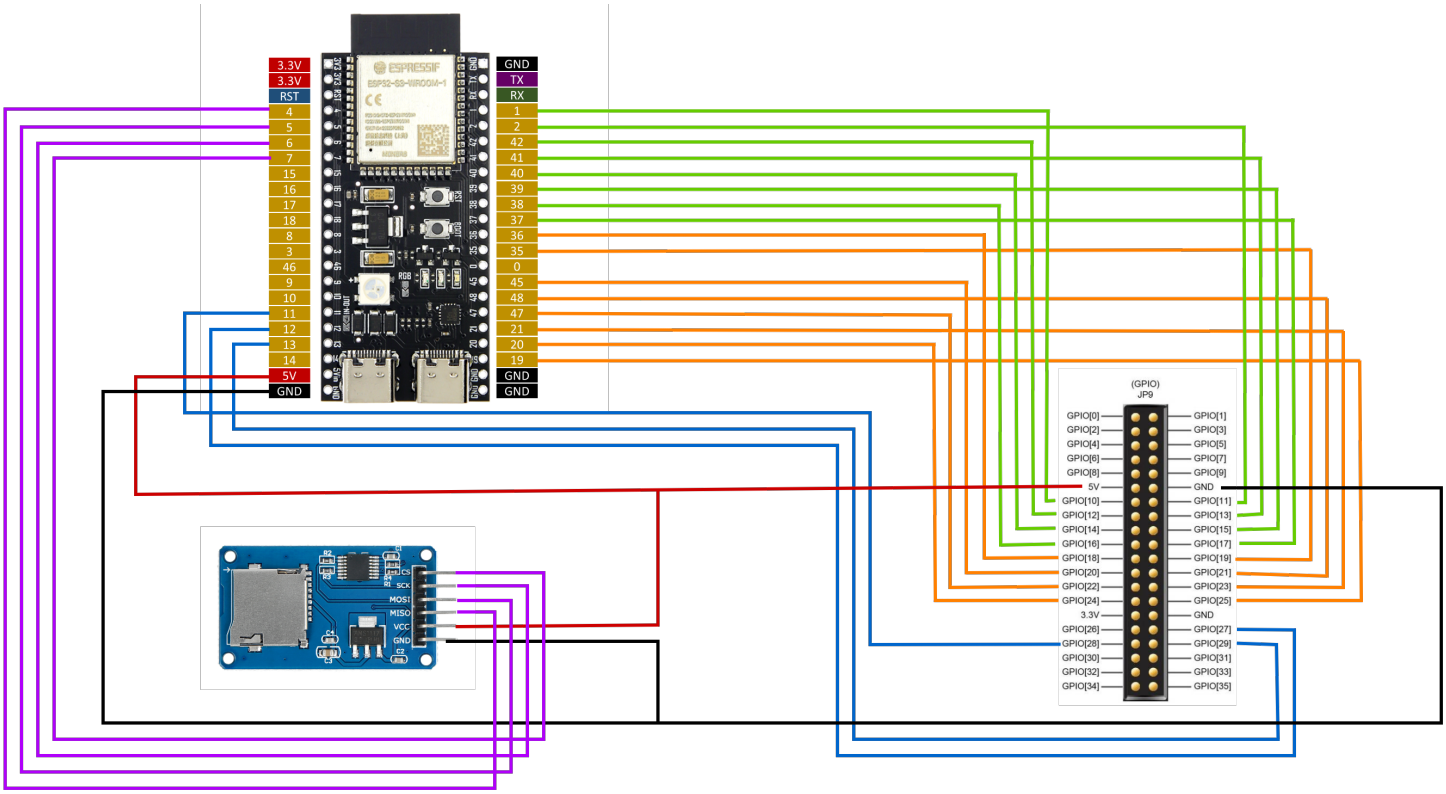

Figure 2.4 Map of the connections between Microcontroller, FPGA board and the SD card module.

2.2 Deep dive into the protocol

Let's discuss the protocol that is used for communication between the microcontroller and the fpga board. On the figure 2.3 is illustrated the directions of communication between these two devices. Let the configuration of the microcontroller be denoted by esp and the configuration of the fpga board by the $fpga$.

Let sd denote the continuous byte-addressable memory buffer where we can store or read 1 byte (8-bit value) by writing:

$$data \subset \mathbb{B}_8$$

$$addr \subset \mathbb{B}_8$$

$$sd[addr] = data \quad \text{Writing to the address}$$

$$data = sd[addr] \quad \text{Reading from the address}$$

esp consists of signals:

$$esp.ready$$

$esp.ready$ denotes the state of microcontroller if it can respond to the commands sent by the fpga.

$$esp.data$$

$esp.data$ denotes the byte which is returned to the fpga as the response.

Now let's define $fpga$ configuration:

$$fpga.clk$$

$fpga.clk$ defines the signal on which esp must proccess the signal if not busy.

$$fpga.rw$$

$fpga.rw$ defines the type of the operation requested. If $fpga.rw$ is 1, we treat the next command as the write operation, and if $fpga.rw$ is 0, as the read operation.

$$fpga.data$$

$fpga.data$ defines the input sent as the part of the command and is only used in case of the write operation.

$$fpga.addr$$

$fpga.addr$ defines the input sent as the part of the command and is used always in both read and write operations.

Let's define the logic using these notations:

$$esp.data = \begin{cases} fpga.data & \text{if } fpga.rw = 1 \wedge esp.ready = 1 \\ sd[fpga.addr] & \text{if } fpga.rw = 0 \wedge esp.ready = 1 \\ esp.data & \text{otherwise} \end{cases}$$

$$fpga.data = \begin{cases} esp.data & \text{if } fpga.rw = 0 \wedge esp.ready = 1 \\ fpga.data & \text{otherwise} \end{cases}$$

$$sd[fpga.addr] = fpga.rw \wedge esp.data$$

This is done on each time we get positive edge signal from the clock $fpga.clk$.

2.3 Implementation on ESP32 side

Below you see the main loop of the program that runs on ESP32:

```
while(running)
{
    if(!rising_edge && gpio_get_level(CLOCK_IN))
        rising_edge = true;
    else if(rising_edge && !gpio_get_level(CLOCK_IN))
    {
        rising_edge = false;
        continue;
    }
    else
    {
        vTaskDelay(10);
        continue;
    }

    gpio_set_level(FLAG_READY_PIN, 1);
    bool write = gpio_get_level(FLAG_RW_PIN);
    ESP_LOGI("DISK", "Reading␣RW␣flag:␣%d", write);

    if(write)
        fpga_filesys_write();
    else
        fpga_filesys_read();

    vTaskDelay(pdMS_TO_TICKS(DELAY_MS));
}
```

The loop is running until the ESP32 it powered off. First, it reads CLOCK_IN signal driven from the fpga board. This acts similarly to the always block in verilog. Each time CLOCK_IN is driven to high, we continue with executing the rest of the loop and stop until the CLOCK_IN signal is driven low and then high again.

```
uint8_t fpga_filesys_read()
{
    uint8_t addr = read_addr_pins();
    uint8_t data = read_data(current_file, addr, 8);
    ESP_LOGI("FILESYS_READ", "Reading␣data:␣%d␣from␣addr:␣%d", data, addr)
        ;
    write_to_data_pins(data);

    return data;
}

void fpga_filesys_write()
{
    uint8_t addr = read_addr_pins();
    uint8_t data = read_data_pins();
    ESP_LOGI("FILESYS_WRITE", "Writing␣data:␣%d␣to␣addr:␣%d.", data, addr)
        ;
    write_data(current_file, addr, data, 8);
}
```

On each clock signal we read RW_PIN to check which operation has been requested by the fpga board. After according to the value of RW_PIN, we either read values of address pins, read inside the file (stored inside the sd card) the value at that address and write the value to the data pins to send result to the fpga board;

2.4 Implementation on FPGA side

On FPGA's side of implementation we have an example verilog code, which uses onboard switches and buttons for inputting data and address and setting if we want to read or write the data. Onboard red and green LEDs are used to display the result.

```
`define ENABLE_GPIO
module baseline_c5gx(
        /* FPGA board inputs and outputs */
        );

        assign LEDG[7:0] = GPIO[17:10];
        assign LEDR[7:0] = GPIO[25:18];

        assign GPIO[27] = SW[9];
        assign LEDR[9] = SW[9];

        assign GPIO[28] = !KEY[1];
        assign LEDR[8] = !KEY[1];

        esp_storage espstrg(
                .data(GPIO[17:10]),
                .addr(GPIO[25:18]),

                .rw(SW[9]),
                .ready(GPIO[26]),

                .store_data(!KEY[3]),
                .store_addr(!KEY[2]),
                .switches(SW[7:0])
        );

        assign GPIO[17:10] = 8'b00000000;

endmodule
```

As discussed in the last section we are using GPIO pins [17:10] for data and [25:18] for address. Here are the list of each button and switch used and their functionality:

- SW0 to SW7: Used to input values for data and address bits.
- SW9: Sets readwrite operation.
- KEY3: Set data wires according to the first 8 switches.
- KEY2: Set address wires according to the first 8 switches.
- KEY1: Clock signal sent to ESP32.
- LEDG0 to LEDG7: Represents 8bit value of the data wire.
- LEDR0 to LEDR7: Represents 8bit value of the address wire.

Inside the esp_storage module we observe very simple logic. Since all the heavy lifting is done by the ESP32, FPGA only has to drive wires according to the inputs and deal with bidirectional connection.

```
module esp_storage(
        inout [7:0] data,
        output [7:0] addr,

        input rw,
        input ready,

        input store_data,
        input store_addr,
        input [7:0]switches
);
        wire [7:0]internal_data;

        assign data = (rw & ready) ? internal_data : 1'bZ;

        assign internal_data = (store_data) ? switches : internal_data;
        assign addr = (store_addr) ? switches : addr;
endmodule
```