Design and Verification Report for Dual-Channel ADC SPI Controller

Based on `ADC.v` and `ADC_tb.v` Files

December 4, 2025

Abstract

This document reports on the design and functional verification of a digital controller, `ADC2`, implemented in Verilog HDL. The module is designed to operate as a Serial Peripheral Interface (SPI) master to configure and acquire data from a dual-channel Analog-to-Digital Converter (ADC). The controller manages clock generation, gain setting, conversion pulse generation, and serial data capture over a 34-bit transaction window. Functional correctness was verified using the self-checking testbench `adc_tb`, which successfully emulated the ADC's serial data transmission.

1 Introduction

The objective of this project is to implement a robust SPI master module capable of controlling a dual-channel, 14-bit ADC device. The controller, named `ADC2`, is responsible for the synchronous sequencing of control signals (`amp_cs`, `adc_conv`) and the clock (`spi_sck`), as well as shifting out configuration data (`spi_mosi`) and capturing the conversion results (`spi_miso`).

2 Design Specification (`ADC.v`)

The `ADC2` module utilizes a sequential state machine synchronized by a divided clock to manage the complex timing requirements of the SPI protocol.

2.1 System Clock and Timing

The module is driven by a high-frequency system clock, `clk` (assumed to be 50 MHz based on typical FPGA constraints). A slower clock, `clk_out`, is generated to run the main state machine, providing stable, synchronized control signals.

- **Input Clock (`clk`)**: 50 MHz (20 ns period).
- **Clock Divider**: The logic uses counters (`pos_count`, `neg_count`) to divide the input clock by a factor of 25.
- **State Machine Clock (`clk_out`)**: The state machine operates at $f_{clk}/25$, resulting in a frequency of 2 MHz.

2.2 SPI Interface Signals

The core communication uses a standard 4-wire SPI connection, along with an ADC-specific conversion signal.

- **Serial Clock (`spi_sck`)**: Output clock generated by the master for synchronization.
- **Chip Select (`amp_cs`)**: Active-low signal to enable communication with the ADC.
- **Master Out Slave In (`spi_mosi`)**: Carries 8 bits of gain configuration data (`8'b00010001`).
- **Master In Slave Out (`spi_miso`)**: Serial input line carrying the ADC conversion results.
- **ADC Conversion (`adc_conv`)**: Active-high pulse to initiate the sampling and conversion cycle.

The design also includes logic to disable other peripheral chip selects on the SPI bus (`spi_ss_b`, `sf_ce0`, `fpga_init_b`, `dac_cs`), ensuring the `ADC2` is the only selected slave.

2.3 State Machine Flow

The finite state machine (FSM) controls the ADC data acquisition sequence. The critical sequence involves:

1. **Gain Setting (States 4-6)**: The 8-bit gain configuration is transmitted MSB-first while `amp_cs` is held low.
2. **Idle/Delay (States 7-10)**: Introduces a necessary delay using the counter `cnt` before conversion.
3. **Conversion Start (States 16-18)**: The `adc_conv` signal is pulsed high for one `clk_out` cycle, initiating the ADC conversion.
4. **Data Acquisition (States 20-23)**: The module generates 34 pulses on `spi_sck` to receive the converted data. The data is captured bit-by-bit from `spi_miso`.

2.4 Data Alignment and Output

The ADC returns data for two 14-bit channels in a single 34-bit stream, aligned as follows:

$$\texttt{adc\_data} = \{2'bzz, \texttt{adc\_data1[13:0]}, 2'bzz, \texttt{adc\_data2[13:0]}, 2'bzz\}$$

- **Bits 1-2 (`adc_clk_count` ≤ 2)**: Ignored/High-impedance from ADC.
- **Bits 3-16 (`adc_clk_count` ≤ 16)**: Captured into `adc_data1` (Channel A).
- **Bits 17-18 (`adc_clk_count` ≤ 18)**: Ignored/High-impedance.
- **Bits 19-32 (`adc_clk_count` ≤ 32)**: Captured into `adc_data2` (Channel B).
- **Bits 33-34 (`adc_clk_count` ≤ 34)**: Ignored/High-impedance.

The captured 14-bit words are output on `adc_data1` and `adc_data2`.

3 Design Implementation (`ADC.v`)

The complete Verilog HDL code for the `ADC2` master controller module is provided below.

```
module ADC2 (
    input  wire        clk,
    input  wire        enable,
    input  wire        spi_miso, // ADC output and inpput to master
    output wire        clk_out,
    output wire        a1,
    output wire        a2,
    output reg         spi_sck, // SPI clock
    output reg         amp_cs,
    output reg         adc_conv, // ADC controller conversion signal
    output reg         spi_mosi,
    output reg         amp_shdn,
    output wire        spi_ss_b, sf_ce0, fpga_init_b, dac_cs, // disabling
        signal
    output reg  [13:0] adc_data1,
    output reg  [13:0] adc_data2,
    output wire [33:0] adc_data // 2'bzz + 14 bit (ch A output) + 2'bzz 14
        bit (ch B output) + 2'bzz
);                                  // 2 + 14 + 2 + 14 + 2 = 34

reg adc_sent = 0;

reg [2:0] cnt = 0;
reg [3:0] clk_10_count = 0;
reg [6:0] adc_clk_count = 0; // For 34 clock pulse
reg [5:0] adc_bit_count = 14;    // count for 16 clock pulse (14 databit 2
    high impedence state)
reg [3:0] gain_count = 8;   // gain count of 8 bit
reg [4:0] pos_count, neg_count;

reg [7:0] data_gain = 8'b00010001; // setting GAIN = -1 (INITIALIZATION)

reg [5:0] state = 6'b000000;

// Disabling other peripheral communicating with SPI BUS

assign spi_ss_b = 0;    // SPI Serial Flash
assign sf_ce0 = 1;  // StrataFlash Parallel Flash PROM
assign fpga_init_b = 1; // platform Flash PROM
assign dac_cs = 1;  // DAC (Digital to Analog Converter)

// clok division by 25 (Spartan-3E --> 50mhz)
always @ (posedge clk or posedge enable) begin
    if (enable) begin
        pos_count <= 0;
    end
    else begin
        if (pos_count == 24) pos_count <= 0;
        else pos_count <= pos_count + 1;
    end
end

always @ (negedge clk or posedge enable) begin
    if (enable) begin
```

```
52          neg_count <= 0;
53      end
54      else begin
55          if (neg_count == 24) neg_count <= 0;
56          else neg_count <= neg_count + 1;
57      end
58  end
59
60  assign clk_out = ((pos_count > (25 >> 1)) | (neg_count > (25 >> 1) ));
61  assign a1 = amp_cs;
62  assign a2 = adc_conv;
63
64  //The sampled analog value is converted to digital data 32 SPI_SCK cycles
65  //after asserting AD_CONV
66
67   assign adc_data = {2'bzz, adc_data1, 2'bzz, adc_data2, 2'bzz};
68
69  always @ (posedge clk_out or posedge enable) begin
70      if (enable) begin
71          spi_sck <= 0;
72          amp_shdn <= 0;
73          adc_conv <= 0;
74          amp_cs <= 1;
75          spi_mosi <= 0;
76          // ac_data1 <= 14'b10110010000111
77          state <= 1;
78          end
79
80          else begin
81
82          case (state) // states of ADC
83
84          1: begin
85              state <= 2;
86          end
87          2: begin
88              spi_sck <= 0;
89              amp_cs <= 0;
90              state <= 3;
91          end
92          3: begin
93              spi_sck <= 0;
94              state <= 4;
95          end
96          4: begin    // Gain setting
97              spi_sck <= 0;
98              amp_shdn <= 0;
99              amp_cs <= 0;
100             spi_mosi <= data_gain [gain_count - 1];
101             gain_count <= gain_count - 1;
102             state <= 5;
103         end
104         5: begin
105             amp_cs <= 0;
106             spi_sck <= 1;
107             if (gain_count > 0) state <= 6;
108             else begin
109                 spi_sck <= 1;
```

```
                amp_shdn <= 0;
                amp_cs <= 0;
                gain_count <= 8;
                state <= 7;
            end
        end
        6: begin
            spi_sck <= 1;
            state <= 3;
        end
        7: begin
            amp_cs <= 0;
            spi_sck <= 1;
            state <= 8;
        end
        8: begin
            spi_sck <= 0;
            state <= 9;
        end
        9: begin
            spi_sck <= 0;
            state <= 10;
        end
        10: begin
            if (cnt > 5) begin // DELAY
                spi_sck <= 0;
                state <= 11;
                cnt <= 0;
            end
            else begin
                cnt <= cnt + 1;
                spi_sck <= 0;
            end
        end

        11: begin
            amp_cs <= 1; // Disabling gain setting after setting gain
            spi_sck <= 0;
            state <= 12;
        end

        12: begin
            spi_sck <= 0;
            state <= 13;
        end

        13: begin
            spi_sck <= 1;
            state <= 14;
        end

        14: begin
            spi_sck <= 1;
            state <= 15;
        end

        15: begin
            spi_sck <= 0;
```

```
168             state <= 30;
169         end
170
171         30: begin
172             spi_sck <= 0;
173             state <= 16;
174         end
175
176         16: begin
177             adc_conv <= 1;  // Start ADCs from here
178             spi_sck <= 0;
179             state <= 17;
180         end
181
182         17: begin
183             spi_sck <= 0;
184             state <= 18;
185         end
186
187         18: begin
188             adc_conv <= 0;
189             spi_sck <= 0;
190             state <= 19;
191         end
192
193         19: begin
194             if (cnt > 3) begin    // DELAY
195                 spi_sck <= 0;
196                 cnt <= 0;
197                 state <= 20;
198             end
199             else begin
200                 cnt <= cnt + 1;
201                 state <= 19;
202             end
203         end
204
205         20: begin
206             spi_sck <= 0;
207             state <= 21;
208         end
209
210         21: begin
211             spi_sck <= 0;
212             adc_conv <= 0;
213             adc_clk_count <= adc_clk_count + 1;
214             //adc_bit_count <= adc_bit_count - 1;
215             state <= 22;
216         end
217
218         22: begin
219             spi_sck <= 1;
220             state <= 23;
221         end
222
223         23: begin
224             spi_sck <= 1;
225             if (adc_clk_count == 34) begin
```

```
226                 adc_sent <= 1;
227                 //spi_sck <= 0;
228                 state <= 24;
229             end
230
231             else if (adc_clk_count <= 2) begin // first two clock where ADC
                    output = Z
232                 //spi_sck <= 0;
233                 state <= 20;
234             end
235
236             else if ((adc_clk_count>2) && (adc_clk_count<=16)) begin //
                   first 14 bits
237                 //spi_sck <= 0;
238                 adc_data1[adc_bit_count - 1] <= spi_miso; // output of ADC1
239                 adc_bit_count <= adc_bit_count - 1;
240                 state <= 20;
241             end
242
243             else if ((adc_clk_count > 16) && (adc_clk_count <= 18)) begin
                   // Here ADC output = Z
244                 //spi_sck <= 0;
245                 adc_bit_count <= 14;
246                 state <= 20;
247             end
248
249             else if ((adc_clk_count > 18) && (adc_clk_count <= 32)) begin
                   // for another 14 bit
250                 //spi_sck <= 0;
251                 adc_data2[adc_bit_count - 1] <= spi_miso;
252                 adc_bit_count <= adc_bit_count - 1;
253                 state <= 20;
254             end
255
256             else if (adc_clk_count == 33) begin // 33 clk pulse
257                 //spi_sck <= 0;
258                 state <= 20;
259             end
260         end
261
262         24: begin
263             adc_clk_count <= 0;
264             adc_bit_count <= 14;
265             spi_sck <= 0;
266             state <= 25;
267         end
268
269         25: begin
270             spi_sck <= 0;
271             adc_sent <= 0;
272             // adc_conv <= 1;
273             state <= 26;
274         end
275
276         26: begin
277             spi_sck <= 1;
278             amp_shdn <= 0;
279             state <= 27;
```

```
280         end
281
282         27: begin
283             spi_sck <= 1;
284             state <= 28;
285         end
286
287         28: begin
288             if (cnt > 4) begin
289                 spi_sck <= 0;
290                 state <= 16; // getting ADC output in 2 channels
                        simultaneously
291                                 // after 34 clock cycle and when adc_conv = 1'
                                    b1 again for next propagation
292
293                 cnt <= 0;
294             end
295             else begin
296                 cnt <= cnt + 1;
297                 spi_sck <= 0;
298                 state <= 28;
299             end
300         end
301         endcase
302         end
303     end
304 endmodule
```

Listing 1: Verilog Code for the ADC2 Controller Module

4 Verification Methodology (ADC_tb.v)

The testbench is designed for functional verification of the ADC2 module's control and data capture logic.

4.1 Test Environment Setup

- **Timescale**: Defined as `1ns/1ps` for high-precision simulation.
- **Clock Generation**: A 50 MHz clock (`clk`) is generated using an `always` block with a 10 ns delay (`#10 clk = ~clk;`).
- **DUT Instantiation**: The ADC2 module (DUT) is instantiated, connecting all interface signals.

4.2 ADC Slave Emulation

The testbench acts as a behavioral model of the ADC (slave) to provide realistic data on the `spi_miso` line. Two expected 14-bit values are defined:

- E_1: `14'b00001010001001` (Decimal 657)
- E_2: `14'b00001101101001` (Decimal 873)

The data output logic uses an `always @(posedge spi_sck)` block to monitor the master's clock count (`dut.adc_clk_count`) and feed the appropriate bit of E_1 or E_2 onto `spi_miso` on the rising edge of `spi_sck`. This timing mimics a common SPI configuration where the slave prepares data on one clock edge, and the master samples it on the opposite edge (implicitly on the falling edge of the master's state logic in `ADC.v`).

4.3 Self-Checking Mechanism

The `initial` block executes the test sequence and performs a comparison check:

- After sufficient time (`#119240`), the simulation halts.
- It verifies if the captured data (`adc_data1` and `adc_data2`) is identical to the expected data (E_1 and E_2).
- The simulation displays a clear `*** PASS ***` or `*** FAIL ***` message with the binary and decimal values of the expected and received data.

5 Testbench Implementation (`ADC_tb.v`)

The complete Verilog HDL code for the self-checking testbench module is provided below.

```
`timescale 1ns/1ps

module adc_tb;
    // Clock Generation (50 MHz)
    reg clk;
    reg enable;
    reg spi_miso; // ADC output and input to master

    wire clk_out, a1, a2, spi_sck;
    wire [13:0] adc_data1, adc_data2;
    wire [33:0] adc_data;

    wire amp_cs_w;
    wire adc_conv_w;
    wire spi_mosi_w;
    wire amp_shdn_w;

    ADC2 dut (
        .clk(clk),
        .enable(enable),
        .spi_miso(spi_miso),
        .clk_out(clk_out),
        .a1(a1),
        .a2(a2),
        .spi_sck(spi_sck),
        .amp_cs(amp_cs_w),      // Connect to wire
        .adc_conv(adc_conv_w),  // Connect to wire
        .spi_mosi(spi_mosi_w),  // Connect to wire
        .amp_shdn(amp_shdn_w),  // Connect to wire
        .spi_ss_b(),
        .sf_ce0(),
        .fpga_init_b(),
        .dac_cs(),
        .adc_data1(adc_data1),
```

```
        .adc_data2(adc_data2),
        .adc_data(adc_data)
    );

     always #10 clk = ~clk;    // 50 MHz (Period = 20 ns)

    reg [13:0] e1 = 14'b00001010001001;
    reg [13:0] e2 = 14'b00001101101001;

    integer idx1, idx2;

    always @(posedge spi_sck) begin
        if (dut.adc_clk_count > 2 && dut.adc_clk_count <= 16) begin
            if (idx1 >= 0) begin
                spi_miso <= e1[idx1];
                idx1 = idx1 - 1;
            end
        end

        else if (dut.adc_clk_count > 18 && dut.adc_clk_count <= 32) begin
            if (idx2 >= 0) begin
                spi_miso <= e2[idx2];
                idx2 = idx2 - 1;
            end
        end

        else begin
            spi_miso <= 0;
        end
    end


initial begin
        $dumpfile("adc_wave.vcd");
        $dumpvars(0, adc_tb);

        clk = 0;
        enable = 1;
        spi_miso = e1[0];
        idx1 = 13;
        idx2 = 13;

        #100; enable = 0;

        #119240;

        $display("\n--- Simulation Complete ---");
        if (adc_data1 === e1 && adc_data2 === e2) begin
            $display("*** PASS: ADC words match expected ***\n");
            $display("ADC_DATA1 (Received): %b (Dec: %d)", adc_data1,
                adc_data1);
            $display("ADC_DATA2 (Received): %b (Dec: %d)", adc_data2,
                adc_data2);
            $display("E1 (Expected):        %b (Dec: %d)", e1, e1);
            $display("E2 (Expected):        %b (Dec: %d)", e2, e2);
        end else begin
            $display("*** FAIL ***\n");
            $display("E1 (Expected):        %b (Dec: %d)", e1, e1);
```

```
91              $display("Got1 (Received):      %b (Dec: %d)", adc_data1,
                    adc_data1);
92              $display("E2 (Expected):        %b (Dec: %d)", e2, e2);
93              $display("Got2 (Received):      %b (Dec: %d)", adc_data2,
                    adc_data2);
94          end
95
96          $finish;
97      end
98
99  endmodule
```

Listing 2: Verilog Code for the `adc_tb` Testbench Module

6 Results and Conclusion

The Verilog design for the ADC SPI master controller (`ADC2`) successfully implements the required state transitions, clock division, and 34-bit serial data capture protocol for a dual-channel 14-bit ADC. The testbench confirms that the state machine correctly sequences the gain setting and conversion phases, and the data acquisition logic correctly samples and stores both 14-bit channels into `adc_data1` and `adc_data2`. The design is functionally verified to meet the implicit timing specifications of the target ADC device.