Imperial College
London

Lecture 16
(Supplementary)

spi2dac.v Explained

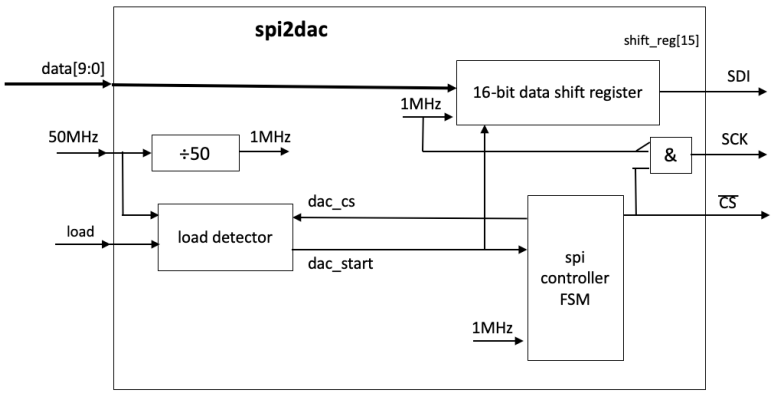
Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/E2_CAS/
E-mail: p.cheung@imperial.ac.uk

Spi2dac.v design overview

- The components inside spi2dac are:

1. Clock divider
2. Load detector to detect load pulse
3. FSM to control the spi interface
4. Parallel to serial shift register to shift OUT the command and data to the DAC
5. Various gates e.g. inverters and AND gates



- Note that the Verilog code is designed to match the block diagram shown here
- It consists of TWO state machines, a counter and a shift register

In order to use the DAC, you have to include the interface module "spi2dac" in your design. This module has a schematic shown above. It takes two inputs (in addition to the 50MHz clock signal): data[9:0] is the 10-bit digital data to be converted by the DAC, and a load signal which is a high pulse to trigger the spi2dac module to send the 10-bit data to the DAC.

The internal working of sp2dac can be divided into 4 main modules. The divide-by-50 module is straight forward – it produces a 1MHz clock for the finite state machine, and is gated through the AND gate to generate the serial clock signal (at 1MHz).

The load detector module handles the load command and produces control signals to the SPI state machine and the shift register.

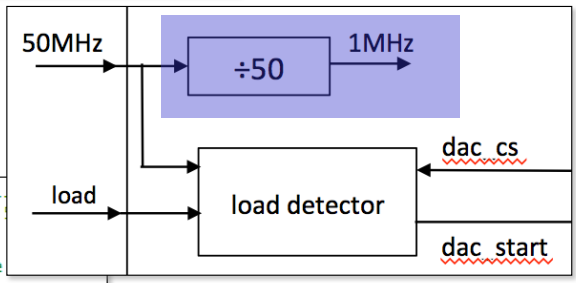
The shift register sends the control bits and the 10-bit data serially to the SDI output.

The spi controller FSM is the main control module designed as a state machine.

We will now consider each sub-module individually.

The 1MHz clock generator

```
parameter    BUF=1'b1;       // 0:no buffer, 1:Vref buffered
parameter    GA_N=1'b1;      // 0:gain = 2x, 1:gain = 1x
parameter    SHDN_N=1'b1;    // 0:power down, 1:dac active

wire [3:0] cmd = {1'b0,BUF,GA_N,SHDN_N};  // wire to VDD or GND
```

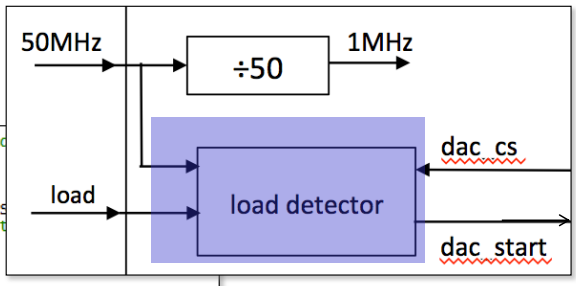


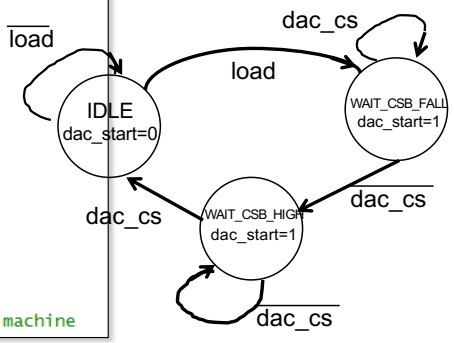

```
// --- internal 1MHz symmetical clock generator ---
reg          clk_1MHz;   // 1Mhz clock derived from
reg [4:0]    ctr;        // internal counter

parameter    TC = 5'd24;  // Terminal count - change
initial begin
   clk_1MHz = 0;         // don't need to reset - don't care
   ctr = 5'b0;           //  ... Initialise when FPGA is confi
end

always @ (posedge sysclk)
  if (ctr==0) begin
     ctr <= TC;
     clk_1MHz <= ~clk_1MHz; // toggle the output clock for sc
   end
  else
     ctr <= ctr - 1'b1;
// ---- end internal 1MHz symmetical clock generator --------
```


This is a straight forward clock divider. The Terminal Count (TC) is set to 24. Divide by 50 is done by toggling the output (clk_1MHz) after 25 clock cycles. Note that I generally prefer to use a down-counter instead of an up-counter. The counter (ctr) is set to 24, it then counts to zero. Output is toggled and the counter (ctr) is reset to the initial value of 24 again.

The load pulse detector

```
// ---- FSM to detect rising edge of load and falling ed
// .... sr_state set on posedge of load
// .... sr_state reset when dac_cs goes high at the end
reg [1:0]   sr_state;
parameter   IDLE  = 2'b00,WAIT_CSB_FALL = 2'b01, WAIT_CS
reg         dac_start;      // set if a DAC write is det

initial begin
   sr_state = IDLE;
   dac_start = 1'b0; // set while sending data to DAC
   end

always @ (posedge sysclk)  // state transition
   case (sr_state)
      IDLE: if (load==1'b1) sr_state <= WAIT_CSB_FALL;
      WAIT_CSB_FALL: if (dac_cs==1'b0) sr_state <= WAIT_CSB_HIGH;
      WAIT_CSB_HIGH: if (dac_cs==1'b1) sr_state <= IDLE;
      default: sr_state <= IDLE;
   endcase

always @ (*)
   case (sr_state)
      IDLE: dac_start = 1'b0;
      WAIT_CSB_FALL: dac_start = 1'b1;
      WAIT_CSB_HIGH: dac_start = 1'b0;
      default: dac_start = 1'b0;
   endcase

//------- End circuit to detect start and end of conversion state machine
```



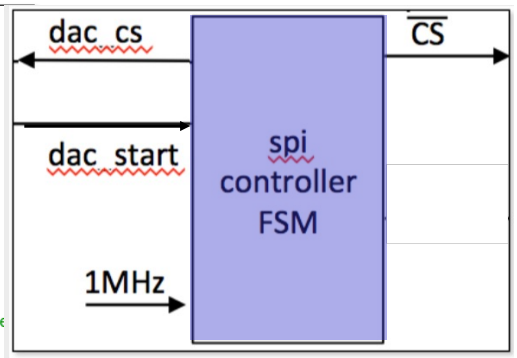


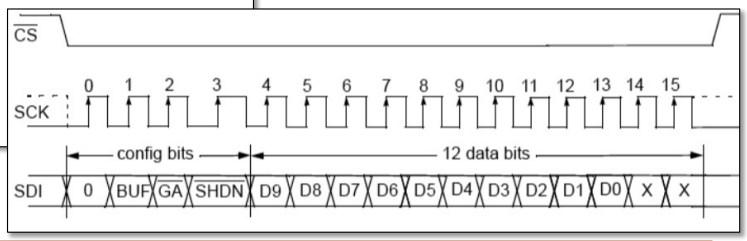
We have TWO signals to detect: the load pulse and the dac_cs signal.

Starting in the IDLE state, when load signal is asserted, we start the DAC cycle by entering the WAIT_CSB_FALL state. In this state, dac_start is asserted, and we wait for DAC_CS to go low from the SPI controller circuit. In this condition, the DAC is in the middle of accepting a new data for conversion. We go to state WAIT_CSB_HIGH TO wait for the conversion to be completed, which is indicated by DAC_CS going high. When that happens, we return to the IDLE state waiting for another 10-bit data to be loaded.

The SPI Controller FSM

```
//------- spi controller FSM
// .... with 17 states (idle, and S1-S16
// .... for the 16 cycles each sending 1-bit to dac)
reg [4:0]   state;

initial  begin
   state = 5'b0; dac_cs = 1'b1;
   end

always @(posedge clk_1MHz)  // FSM state transition
   case (state)
      5'd0: if (dac_start == 1'b1)    // waiting to start
               state <= state + 1'b1;
            else
               state <= 5'b0;
      5'd17:   state <= 5'd0;  // go back to idle state
      default: state <= state + 1'b1;  // default go to next state
   endcase

always @ (*)   begin       // FSM output
   dac_cs = 1'b0;
   case (state)
      5'd0:    dac_cs = 1'b1;
      5'd17:   dac_cs = 1'b1;
      default: dac_cs = 1'b0;
      endcase
   end //always
// --------- END of spi controller FSM
```





The controlling FSM controller is actually simpler than it first appears.

We need a FSM to have 18 states. State 0 is the idle state, waiting for a new data to be sent to the DAC. Here DAC_CS (which is low active) is '1' and we wait for the dac_start to be asserted.

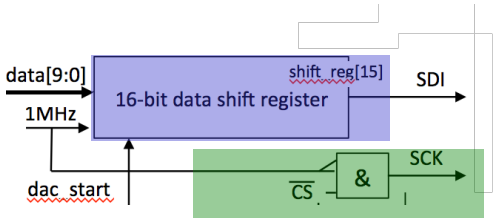
The default value of dac_cs and state are specified first. By default we always go to the next state, i.e. state value goes up by 1.

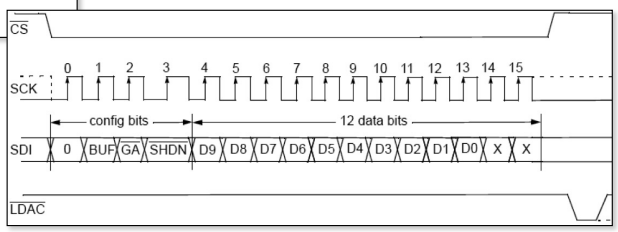
Once the state machine moves to state 1, it just go through to state 16, which corresponds to cycle 0 to 15 in the timing diagram here. At the end of state 16, we de-assert dac_cs (i.e. go high), and go back to the IDLE state.

The data shift register

```
parameter    BUF=1'b1;       // 0:no buffer, 1:Vref buffered
parameter    GA_N=1'b1;      // 0:gain = 2x, 1:gain = 1x
parameter    SHDN_N=1'b1;    // 0:power down, 1:dac active

wire [3:0] cmd = {1'b0,BUF,GA_N,SHDN_N};  // wire to VDD or GND
```

```
// shift register for output data
reg [15:0] shift_reg;
initial  begin
   shift_reg = 16'b0;
   end

always @(posedge clk_1MHz)
   if((dac_start==1'b1)&&(dac_cs==1'b1))      //
      shift_reg <= {cmd,data_in,2'b00};
   else                                       //
      shift_reg <= {shift_reg[14:0],1'b0};

// Assign outputs to drive SPI interface to DAC
      assign dac_sck = !clk_1MHz&!dac_cs;
      assign dac_sdi = shift_reg[15];
```





PYKC 2 Dec 2025 EE2 Circuits and Systems Lecture 16 Slide 6

Finally, the data and clock output is specified here. SDI is driven through a parallel in, serial out shift register.

We use a number of useful tricks here:

1.cmd is a 4-bit value defining the first four bits of the SDI data values. We use symbolic variable names to make the code easy to read.

2.Shift_reg <= {cmd, data_in, 2'b00} - parallel load the 16-bit value into the shift register.

3.Shift_reg <= {shift_reg[14:0], 1'b0} - perform left shift

The SDI is taken from the MSB of the shift register. The serial clock Is !dac_cs (low active) ANDed with the inverter version of the clock (making the rising edge of the SCK signal in the middle of the data bit).