Imperial College London

MSc in Analogue & Digital IC Design

Lecture 3 Pulse-width Modulator, Finite State Machines & Serial-Peripheral Interface

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

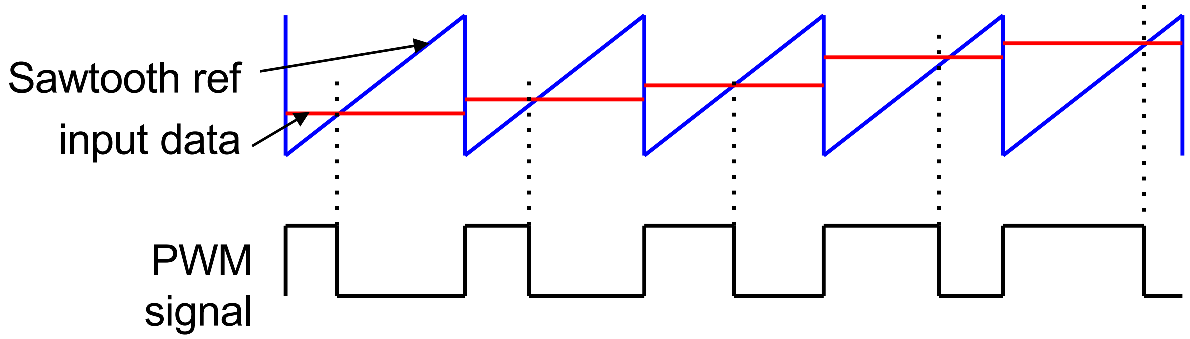
Course webpage: https://github.com/Mastering-Digital-Design/Lab-Module
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

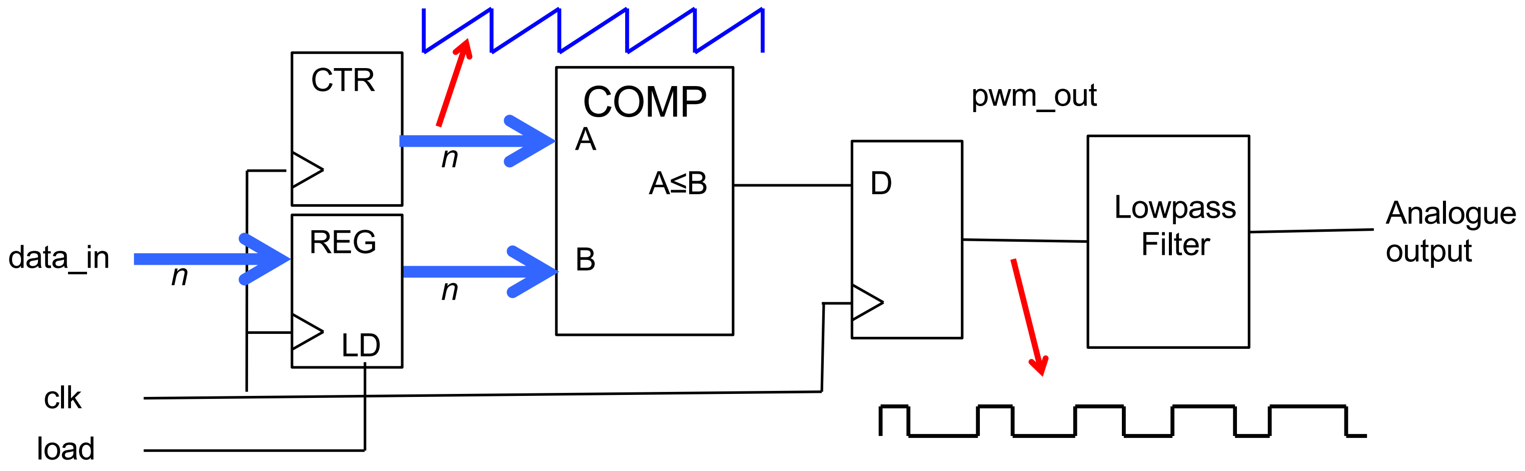
- PWM module and how it works
- Basic about Finite State Machine (FSM)
- How to specify a FSM in Verilog
- The analogue interface add-on card
- Serial Peripheral Interface (SPI) for the DAC and ADC

Pulse-width Modulated (PWM) DAC

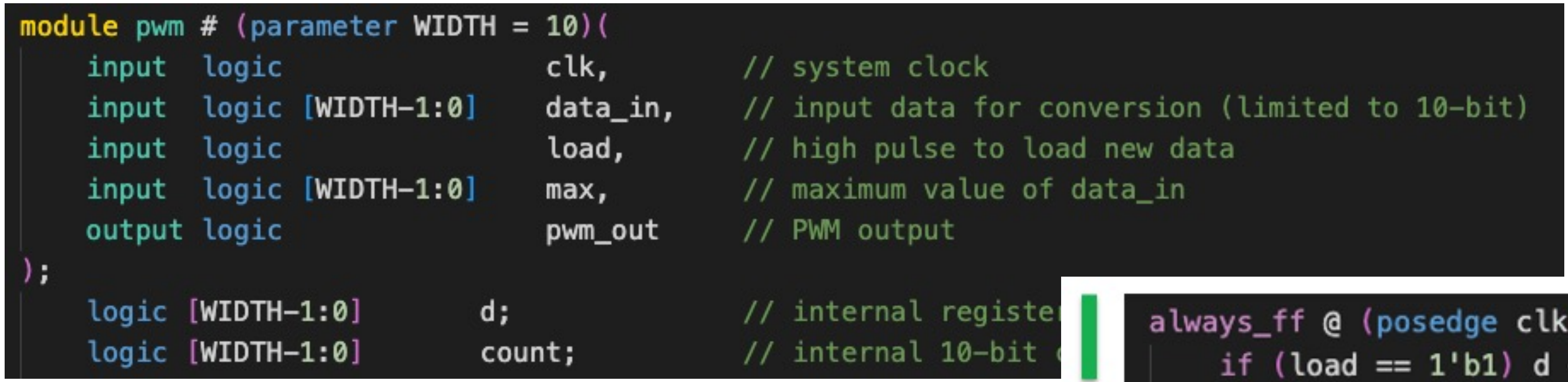
- Simple idea: PWM signal is generated by comparing a triangular reference signal with the input data value



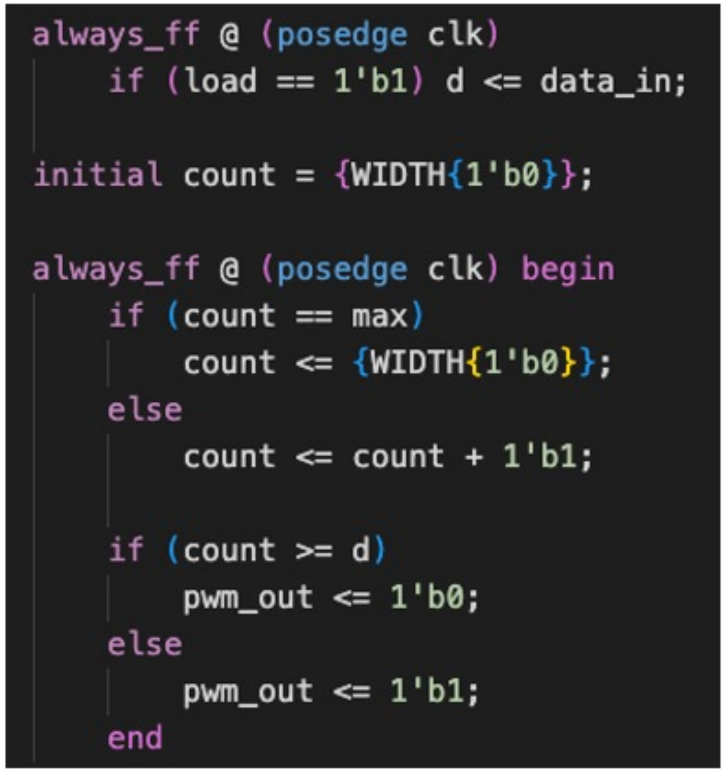
- Sawtooth value generated by a wrap-around counter
- Sample command pulse resets counter, load register and set FF
- When input value is reached by counter, comparator output a pulse to reset FF



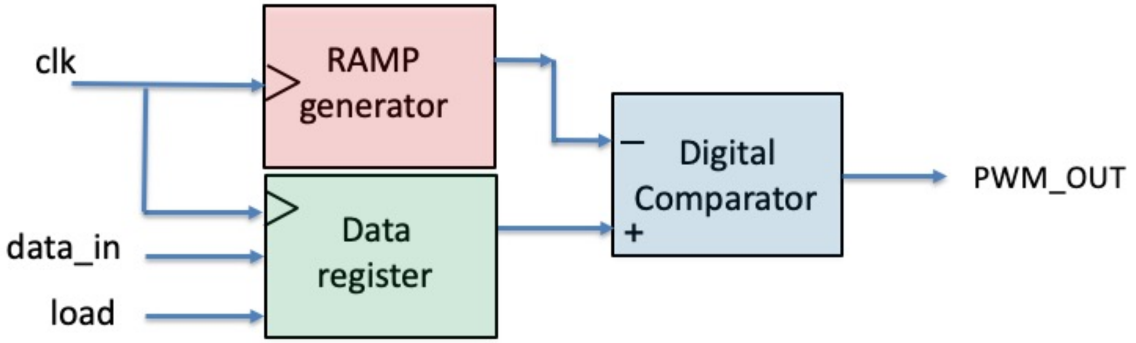
PWM DAC in SystemVerilog



```
module pwm # (parameter WIDTH = 10)(
    input  logic                clk,        // system clock
    input  logic [WIDTH-1:0]    data_in,    // input data for conversion (limited to 10-bit)
    input  logic                load,       // high pulse to load new data
    input  logic [WIDTH-1:0]    max,        // maximum value of data_in
    output logic                pwm_out     // PWM output
);
    logic [WIDTH-1:0]       d;              // internal registe
    logic [WIDTH-1:0]       count;          // internal 10-bit c
```

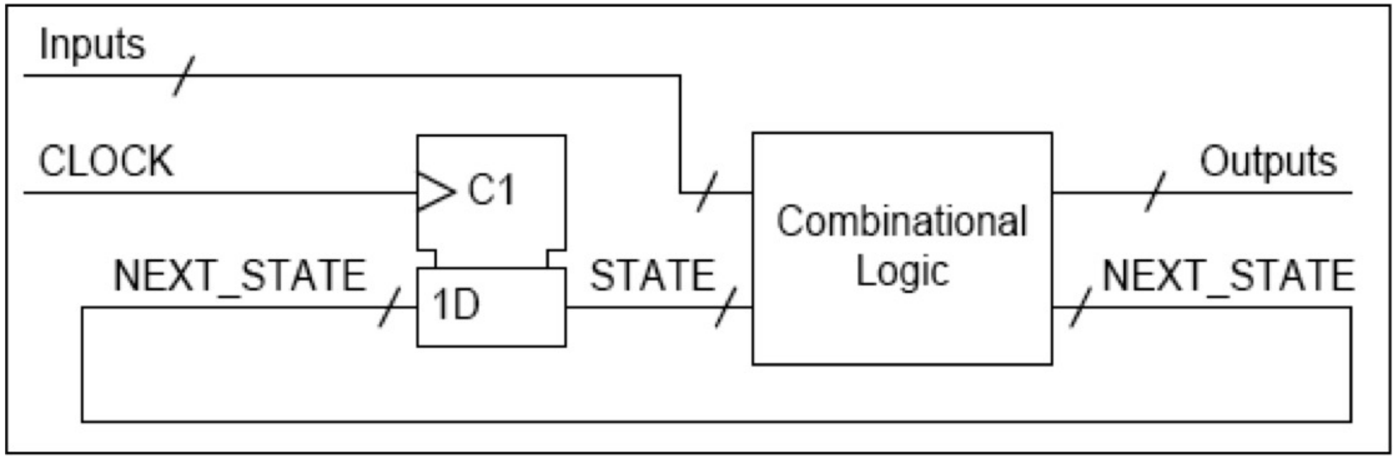


```
always_ff @ (posedge clk)
    if (load == 1'b1) d <= data_in;

initial count = {WIDTH{1'b0}};

always_ff @ (posedge clk) begin
    if (count == max)
        count <= {WIDTH{1'b0}};
    else
        count <= count + 1'b1;

    if (count >= d)
        pwm_out <= 1'b0;
    else
        pwm_out <= 1'b1;
    end
```



Synchronous State Machines

- **Synchronous State Machine (also called Finite State Machine FSM)**
  = Register + Logic

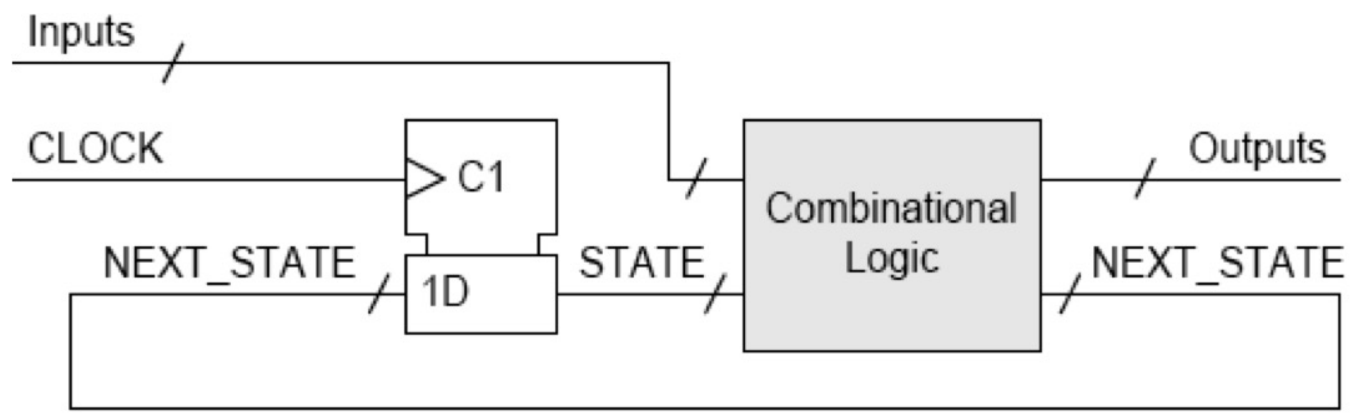


- The ***state*** is defined by the register contents
- Register has *n* flipflops ⇒ 2^n states
- The state only ever changes on CLOCK↑
  - We stay in a state for an exact number of CLOCK cycles
- The state is the only memory of the past

Rules:

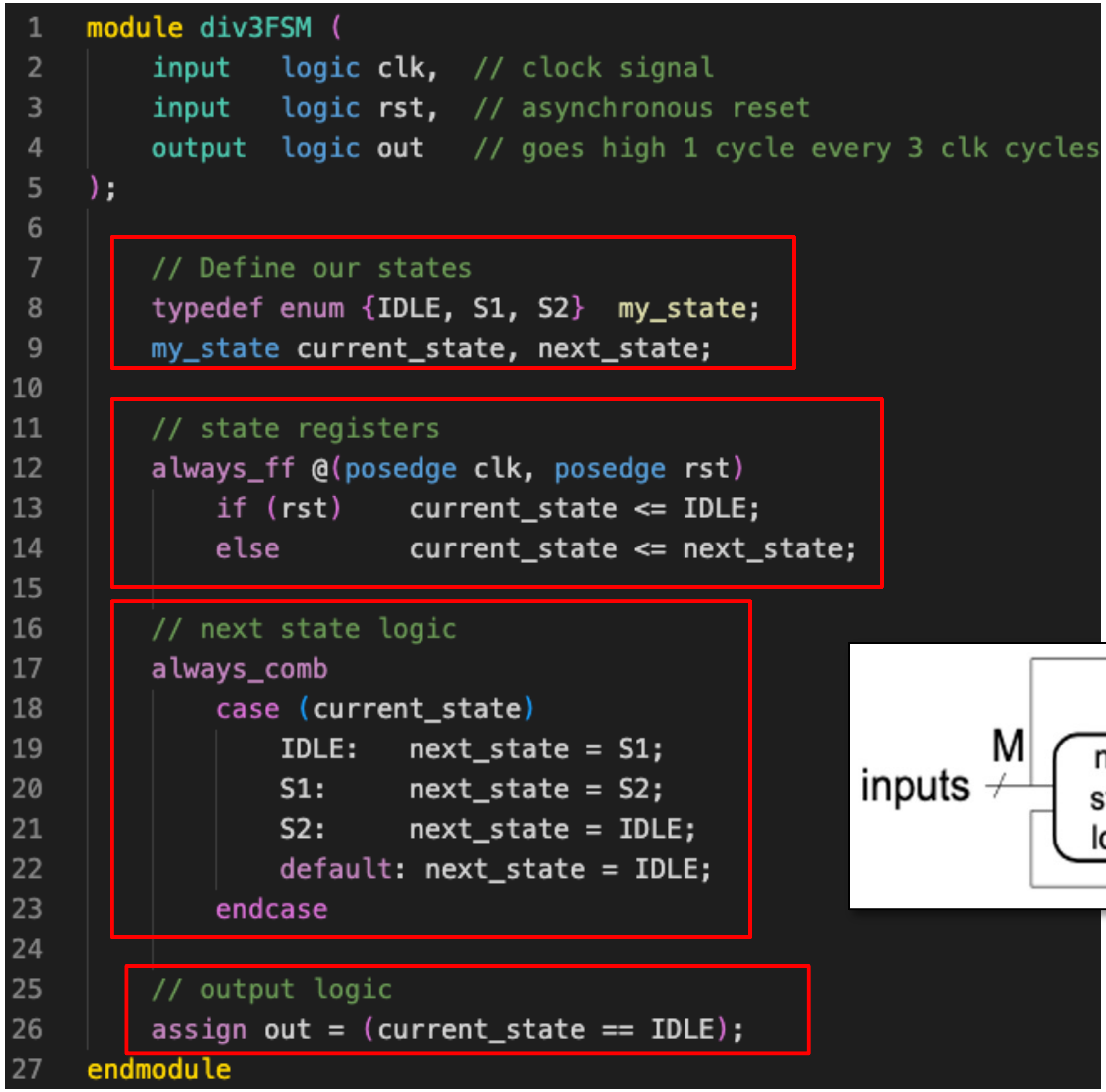
- ❑ Never mess around with the clock signal
- ❑ Never use ***asynchronous*** SET/RESET inputs to register (*asynchronous* = independent of CLOCK)

Combinational Logic Block

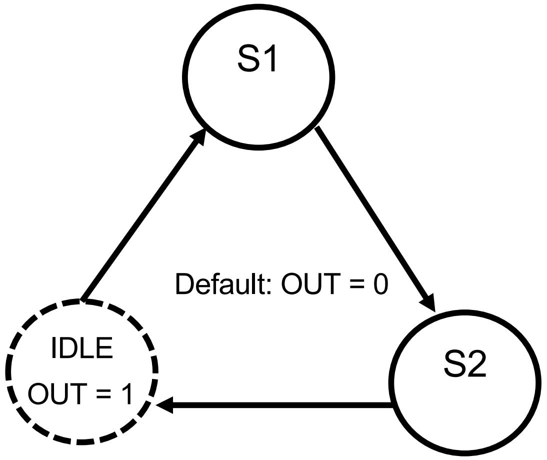


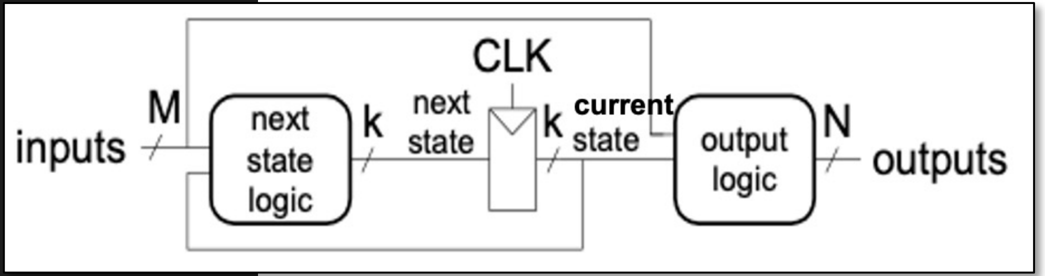
- The combinational logic outputs specify two things:
  - **the output signals during the current state**
    These may change during the state if the inputs change
  - **which state to go to at the next CLOCK**
    This too may change during a state but the only thing that matters is its value just before CLOCK
- ***combinational*** logic has no internal feedback loops $\Rightarrow$ no memory
  - combinational logic outputs are entirely determined by the **current STATE** and the **current Inputs**

Example 1: Divide by 3 FSM (Moore)



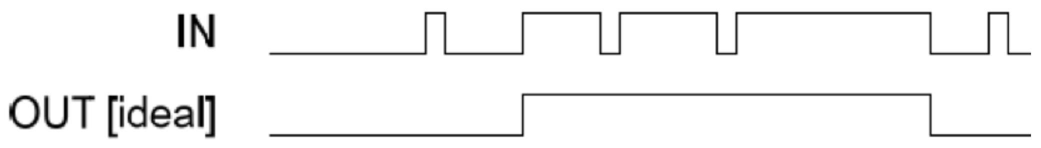
```
module div3FSM (
    input   logic clk,  // clock signal
    input   logic rst,  // asynchronous reset
    output  logic out   // goes high 1 cycle every 3 clk cycles
);

    // Define our states
    typedef enum {IDLE, S1, S2}  my_state;
    my_state current_state, next_state;

    // state registers
    always_ff @(posedge clk, posedge rst)
        if (rst)     current_state <= IDLE;
        else         current_state <= next_state;

    // next state logic
    always_comb
        case (current_state)
            IDLE:    next_state = S1;
            S1:      next_state = S2;
            S2:      next_state = IDLE;
            default: next_state = IDLE;
        endcase

    // output logic
    assign out = (current_state == IDLE);
endmodule
```





Example 2: Design a Noise Pulse Eliminator (1)

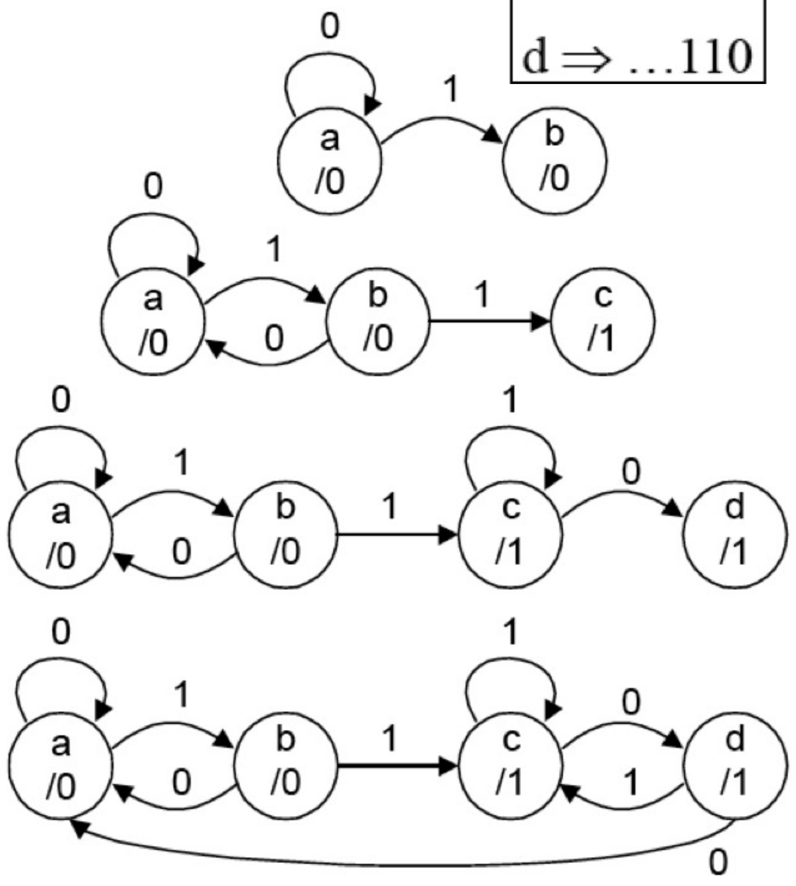
- **Design Problem**: Noise elimination circuit
  - We want to remove pulses that last only one clock cycle



$a \Rightarrow \ldots 00$
$b \Rightarrow \ldots 001$
$c \Rightarrow \ldots 11$
$d \Rightarrow \ldots 110$

- Use letters a,b,… to label states; we choose numbers later.
- Decide what action to take in each state for each of the possible input conditions.
- Use a Moore machine (i.e. output is constant in each state). Easier to design but needs more states & adds output delay.

(1)
(2)
(3)
(4)



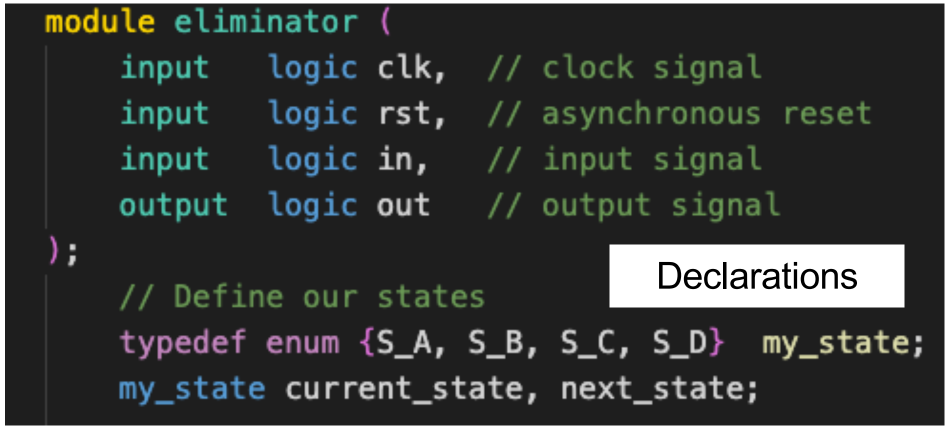
Design a Noise Pulse Eliminator (2)

1. If IN goes high for two (or more) clock cycles then OUT must go high, whereas if it goes high for only one clock cycle then OUT stays low. It follows that the two histories "IN low for ages" and "IN low for ages then high for one clock" are different because if IN is high for the next clock we need different outputs. Hence we need to introduce state b.
2. If IN goes high for one clock and then goes low again, we can forget it ever changed at all. This glitch on IN will not affect any of our future actions and so we can just return to state a.
   If on the other hand we are in state b and IN stays high for a second clock cycle, then the output must change. It follows that we need a new state, c.
3. The need for state d is exactly the same as for state b earlier. We reach state d at the end of an output pulse when IN has returned low for one clock cycle. We don't change OUT yet because it might be a false alarm.
4. If we are in state d and IN remains low for a second clock cycle, then it really is the end of the pulse and OUT must go low. We can forget the pulse ever existed and just return to state a.

Each state represents a particular history that we need to distinguish from the others:

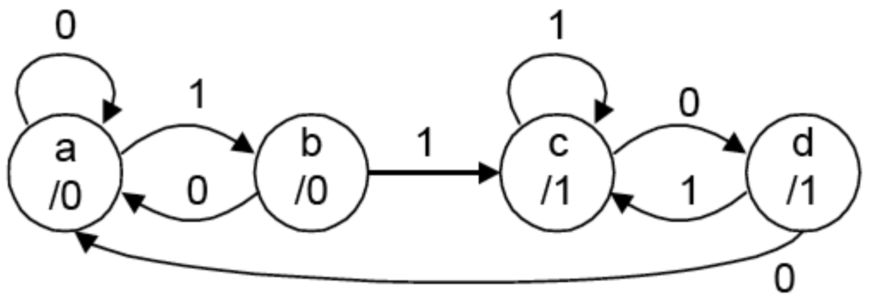
state **a**: IN=0 for >1 clock	state **b**: IN=1 for 1 clock
state **c**: IN=1 for >1 clock	state **d**: IN=0 for 1 clock

Eliminator design in SystemVerilog

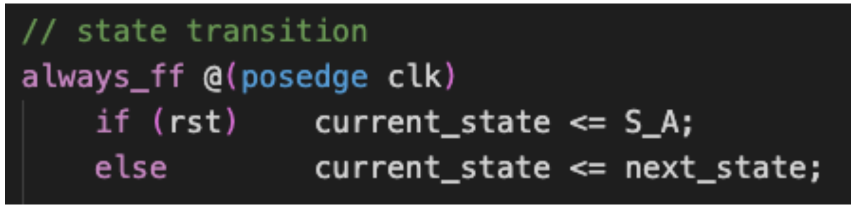


```
module eliminator (
    input   logic clk,  // clock signal
    input   logic rst,  // asynchronous reset
    input   logic in,   // input signal
    output  logic out   // output signal
);
    // Define our states
    typedef enum {S_A, S_B, S_C, S_D}  my_state;
    my_state current_state, next_state;
```

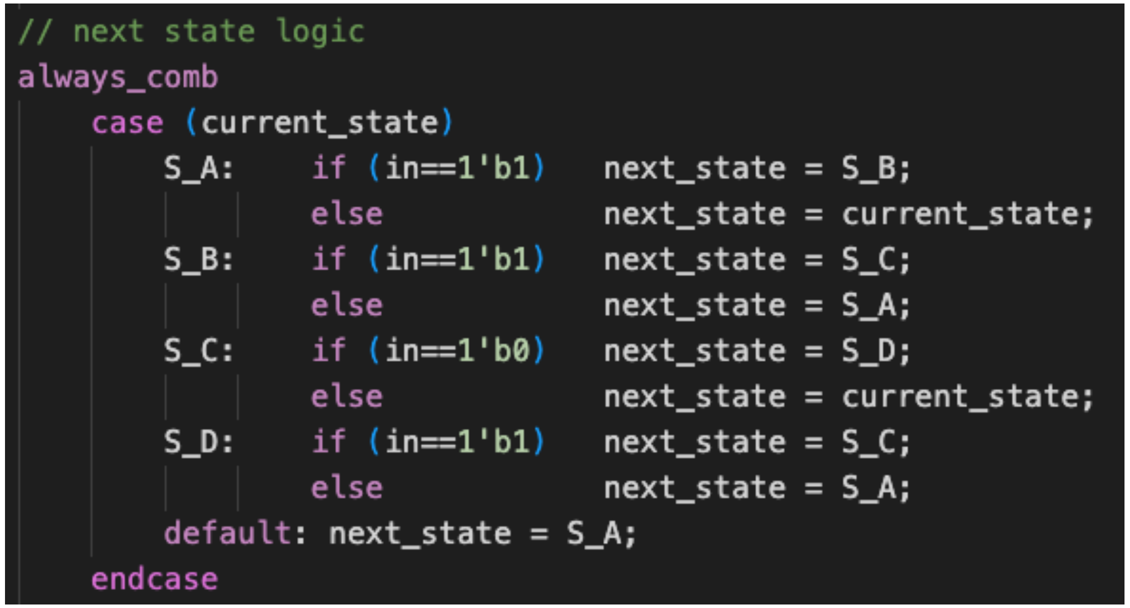
Declarations



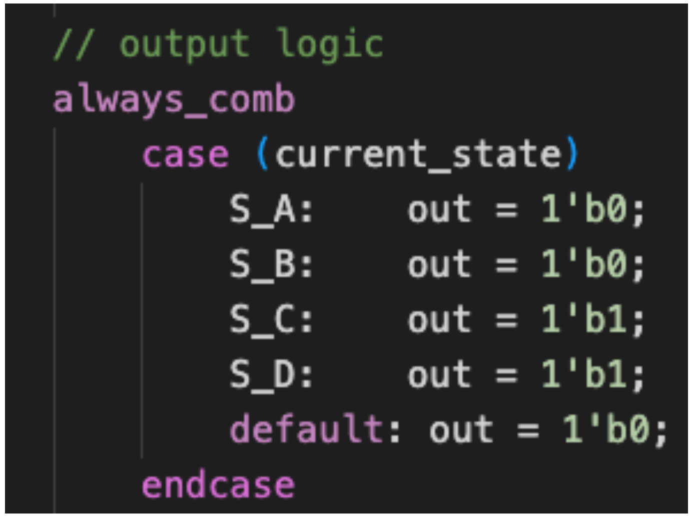
$$\begin{tikzcd}
\text{a /0} \arrow[loop above, "0"] \arrow[r, bend left, "1"] & \text{b /0} \arrow[l, bend left, "0"] \arrow[r, "1"] & \text{c /1} \arrow[loop above, "1"] \arrow[r, bend left, "0"] & \text{d /1} \arrow[l, bend left, "1"] \arrow[lll, bend left, "0"]
\end{tikzcd}$$



```
// state transition
always_ff @(posedge clk)
    if (rst)     current_state <= S_A;
    else         current_state <= next_state;
```



```
// next state logic
always_comb
    case (current_state)
        S_A:    if (in==1'b1)   next_state = S_B;
                else            next_state = current_state;
        S_B:    if (in==1'b1)   next_state = S_C;
                else            next_state = S_A;
        S_C:    if (in==1'b0)   next_state = S_D;
                else            next_state = current_state;
        S_D:    if (in==1'b1)   next_state = S_C;
                else            next_state = S_A;
        default: next_state = S_A;
    endcase
```



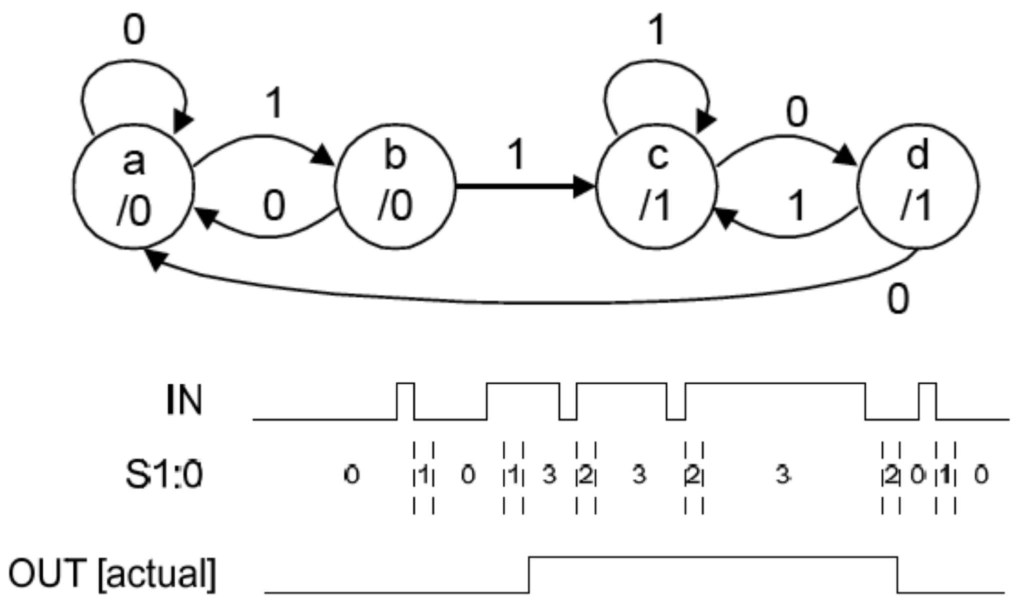
```
// output logic
always_comb
    case (current_state)
        S_A:    out = 1'b0;
        S_B:    out = 1'b0;
        S_C:    out = 1'b1;
        S_D:    out = 1'b1;
        default: out = 1'b0;
    endcase
```

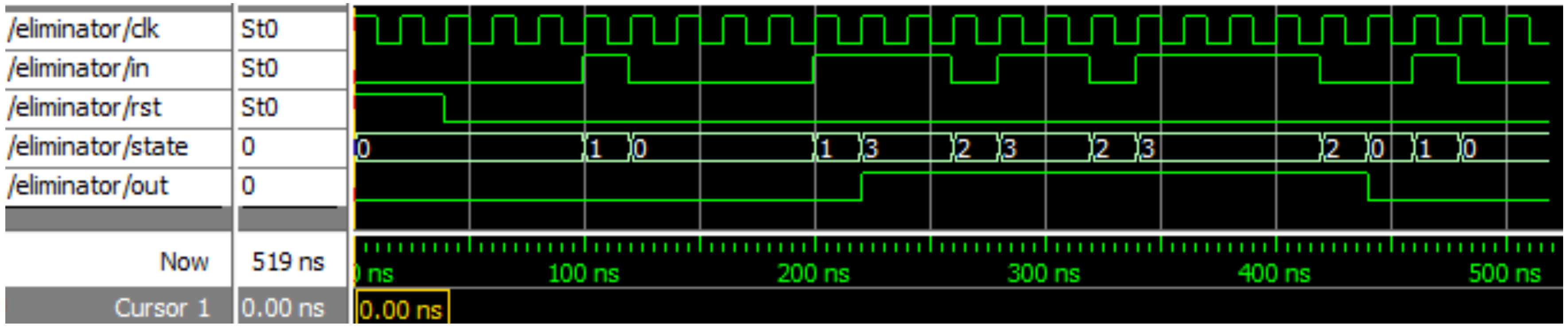
One-hot encoding

- Instead of using binary encoding, which works very well in the noise eliminator example, an alternative is to use one-hot encoding.
- In one-hot encoding, each state is encode with a binary value that has a single ‘1’ bit and the rest of the binary variables are ‘0’.
- Therefore, for the noise eliminator SSM, the states could be encoded as:

  a = 0001 b = 0010 c = 0100 d = 1000
- Using one-hot encoding would use MORE state registers. For N-states, we would need to use N flipflops.
- The advantage is that the state transition and output logic could be much simpler than using binary encoding. There is no longer need for logic to decode the binary number.
- Since FPGAs are a register-rich architecture (each FF is preceded by a small block of logic in the form of a 4-LUT or an ALM), using one-hot encoding could result in simpler and fast SSM implementations.

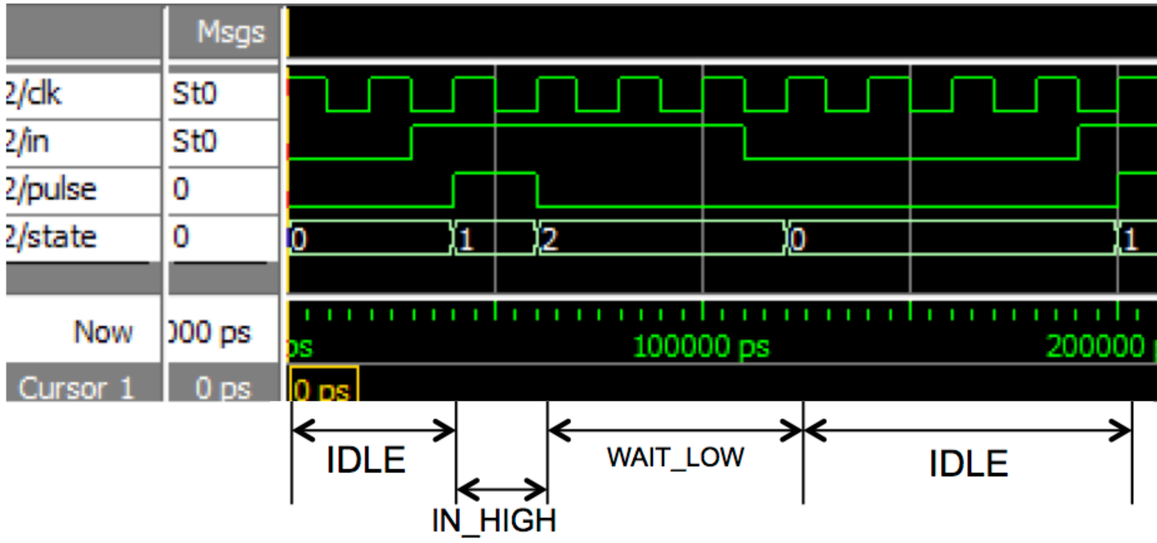
Eliminator simulation in Quartus (RTL)

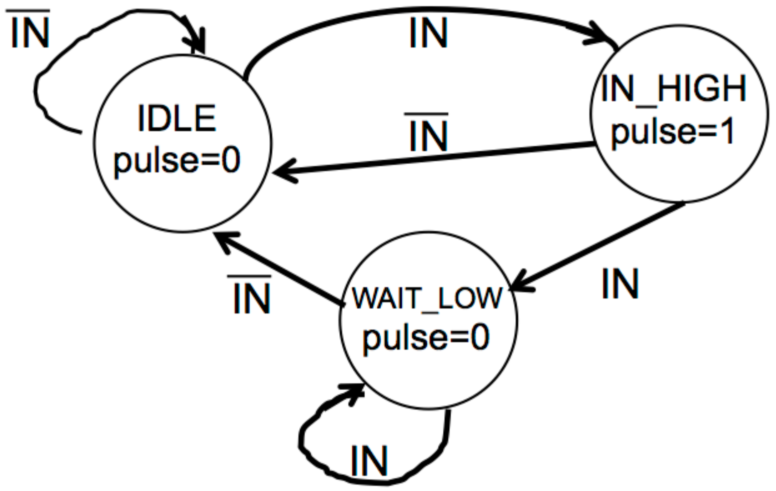




Example 3 – A pulse generator

- Design a module pulse_gen.v which does the following: on each positive edge of the input signal **IN**, it generates a pulse lasting for one period of the input **clock**.

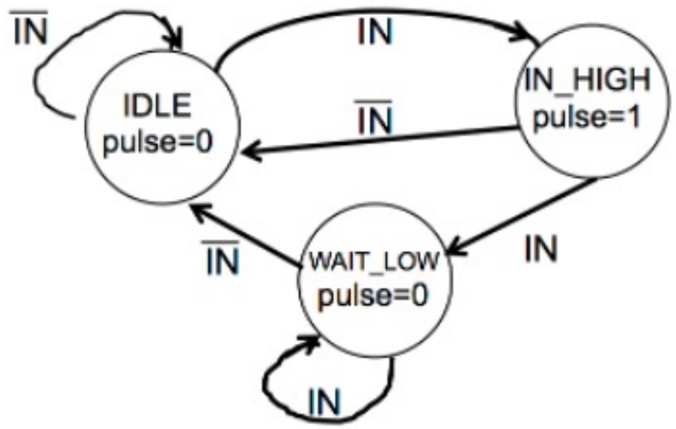


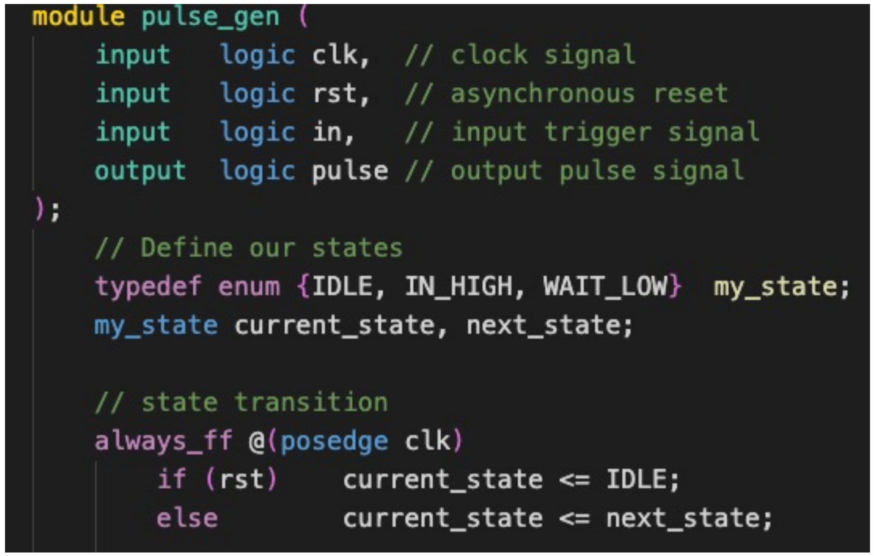


- Needs THREE states (not two).

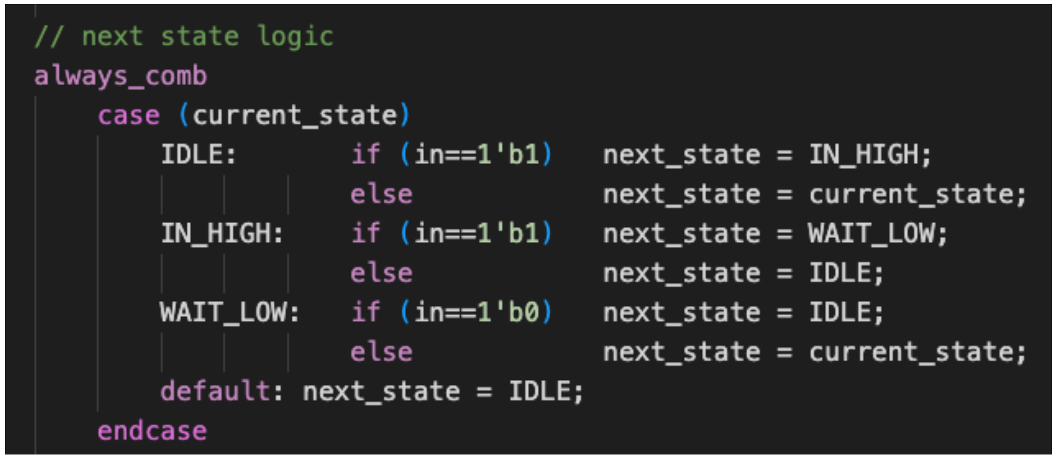
Pulse Generator in SV

- Design a module pulse_gen.v which does the following: on each positive edge of the input signal **IN**, it generates a pulse lasting for one period of the input **clk**.

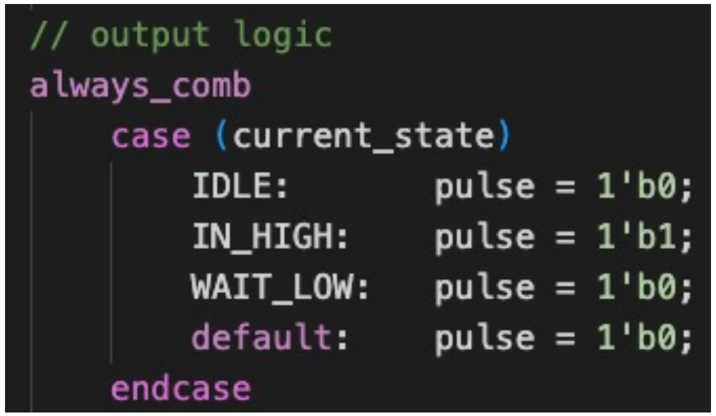




```
module pulse_gen (
    input   logic clk,  // clock signal
    input   logic rst,  // asynchronous reset
    input   logic in,   // input trigger signal
    output  logic pulse // output pulse signal
);
    // Define our states
    typedef enum {IDLE, IN_HIGH, WAIT_LOW}  my_state;
    my_state current_state, next_state;

    // state transition
    always_ff @(posedge clk)
        if (rst)     current_state <= IDLE;
        else         current_state <= next_state;
```



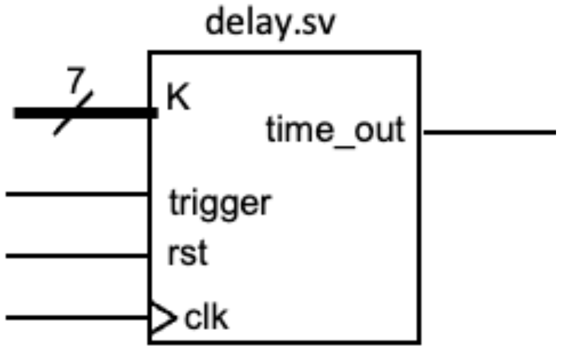
```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (in==1'b1)   next_state = IN_HIGH;
                    else            next_state = current_state;
        IN_HIGH:    if (in==1'b1)   next_state = WAIT_LOW;
                    else            next_state = IDLE;
        WAIT_LOW:   if (in==1'b0)   next_state = IDLE;
                    else            next_state = current_state;
        default: next_state = IDLE;
    endcase
```

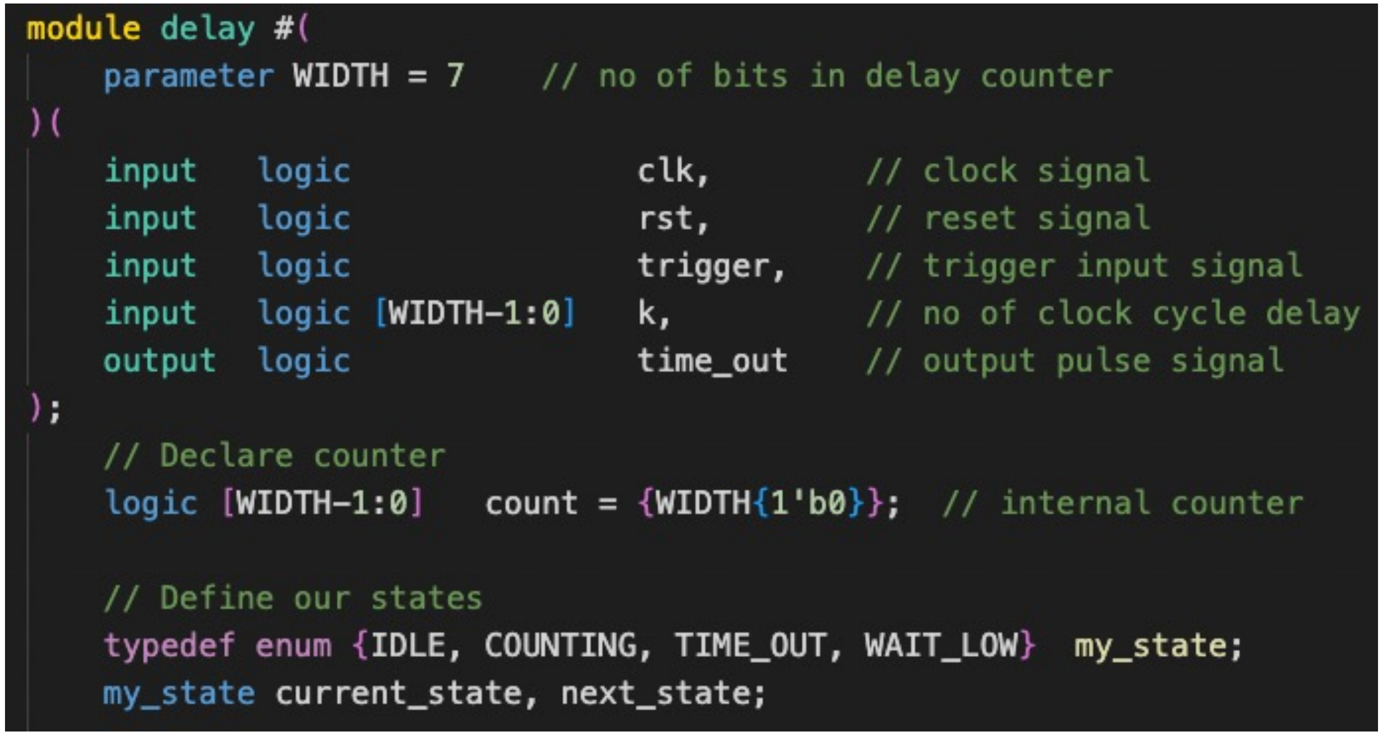


```
// output logic
always_comb
    case (current_state)
        IDLE:       pulse = 1'b0;
        IN_HIGH:    pulse = 1'b1;
        WAIT_LOW:   pulse = 1'b0;
        default:    pulse = 1'b0;
    endcase
```

Example 4: delay module (1)

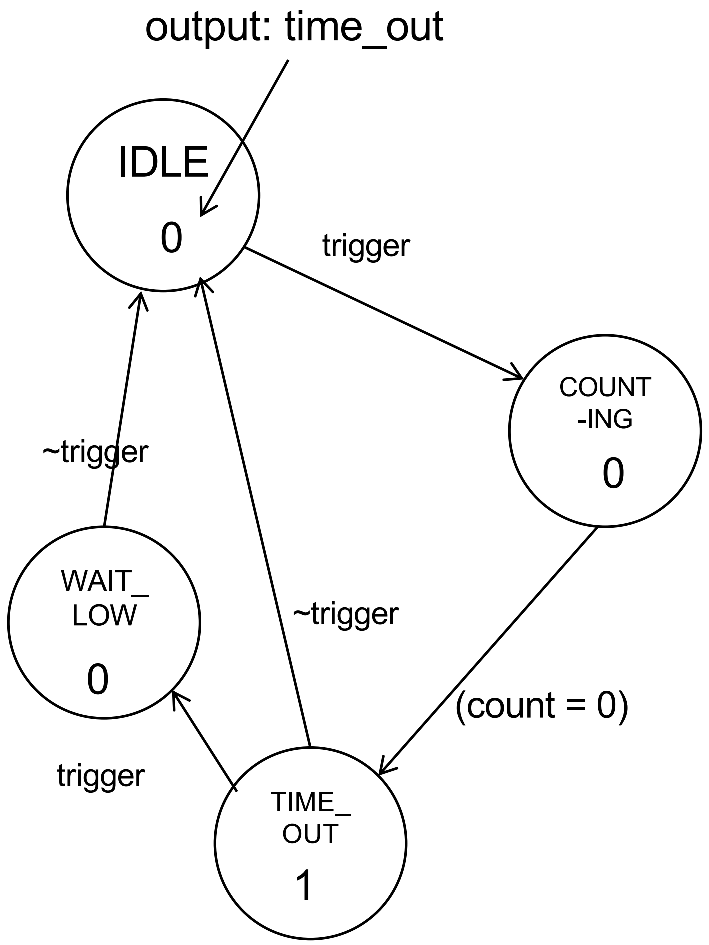
- Here is a very useful module that combines a FSM with a counter.
- It detects the rising edge on trigger, then wait (delay) for n clk cycles before producing a 1-cycle pulse on time_out.
- The external port interface for this module is shown below. We assume that n is a 7-bit number, or a maximum of 127 sysclk cycles delay.

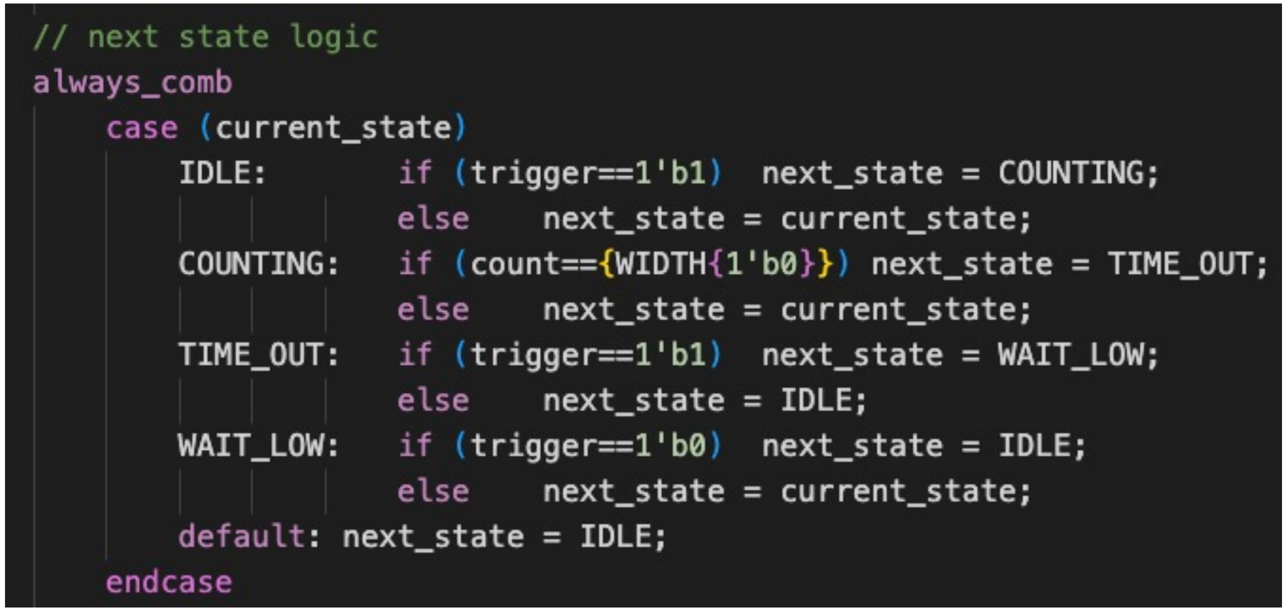




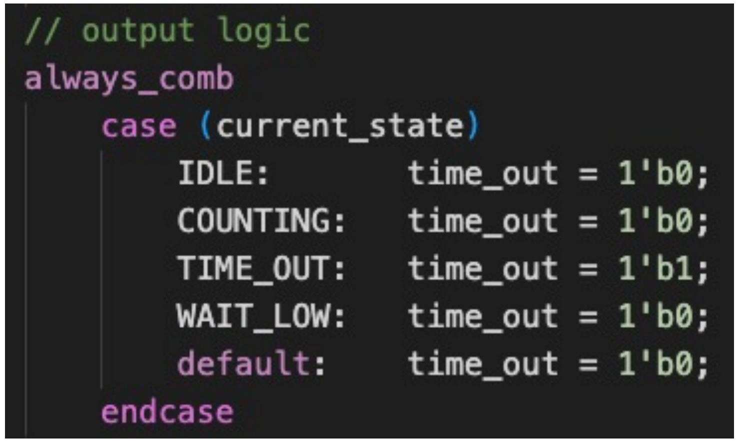
```
module delay #(
    parameter WIDTH = 7    // no of bits in delay counter
)(
    input   logic               clk,        // clock signal
    input   logic               rst,        // reset signal
    input   logic               trigger,    // trigger input signal
    input   logic [WIDTH-1:0]   k,          // no of clock cycle delay
    output  logic               time_out    // output pulse signal
);
    // Declare counter
    logic [WIDTH-1:0]   count = {WIDTH{1'b0}};  // internal counter

    // Define our states
    typedef enum {IDLE, COUNTING, TIME_OUT, WAIT_LOW}  my_state;
    my_state current_state, next_state;
```

Example 4: delay module (2)



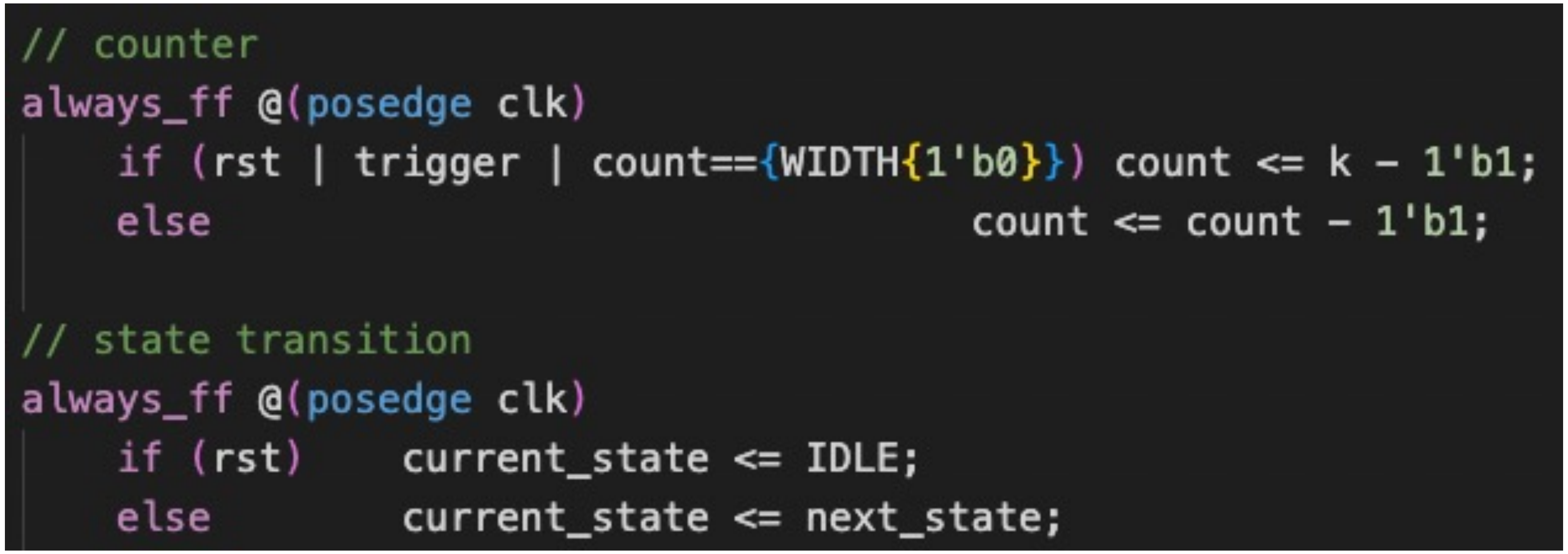


```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (trigger==1'b1)  next_state = COUNTING;
                    else     next_state = current_state;
        COUNTING:   if (count=={WIDTH{1'b0}}) next_state = TIME_OUT;
                    else     next_state = current_state;
        TIME_OUT:   if (trigger==1'b1)  next_state = WAIT_LOW;
                    else     next_state = IDLE;
        WAIT_LOW:   if (trigger==1'b0)  next_state = IDLE;
                    else     next_state = current_state;
        default: next_state = IDLE;
    endcase
```



```
// output logic
always_comb
    case (current_state)
        IDLE:       time_out = 1'b0;
        COUNTING:   time_out = 1'b0;
        TIME_OUT:   time_out = 1'b1;
        WAIT_LOW:   time_out = 1'b0;
        default:    time_out = 1'b0;
    endcase
```

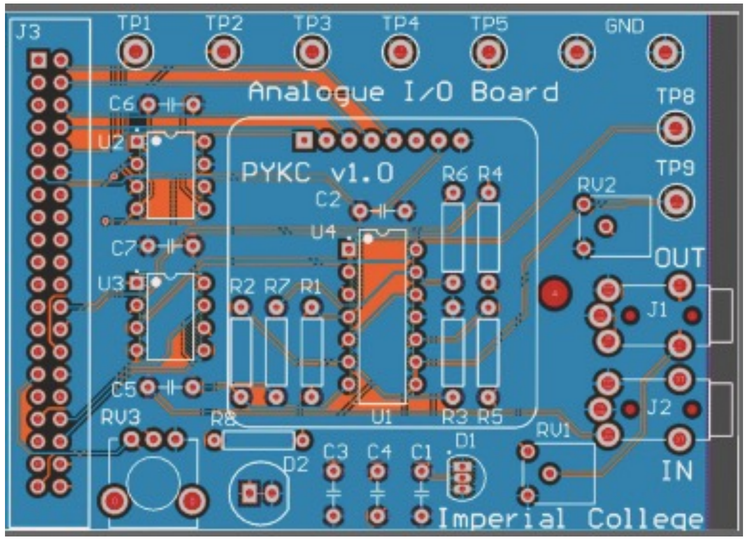
Example 4: delay module (3)

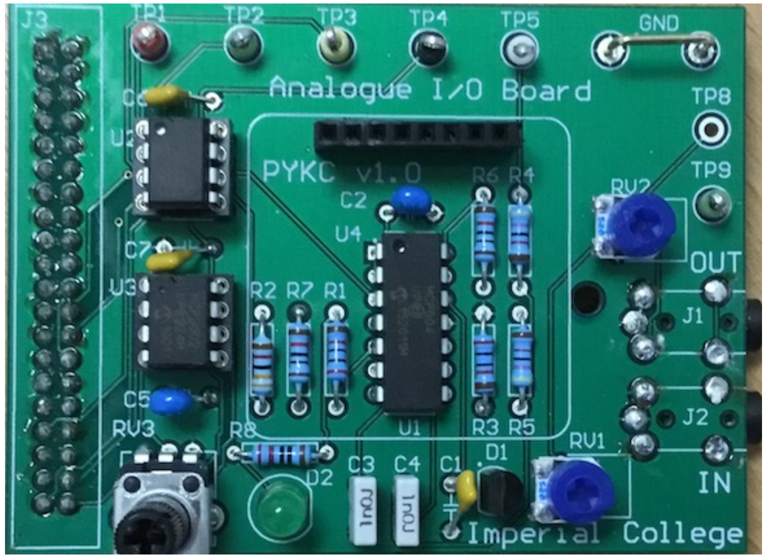


```
// counter
always_ff @(posedge clk)
    if (rst | trigger | count=={WIDTH{1'b0}}) count <= k - 1'b1;
    else                                 count <= count - 1'b1;

// state transition
always_ff @(posedge clk)
    if (rst)     current_state <= IDLE;
    else         current_state <= next_state;
```

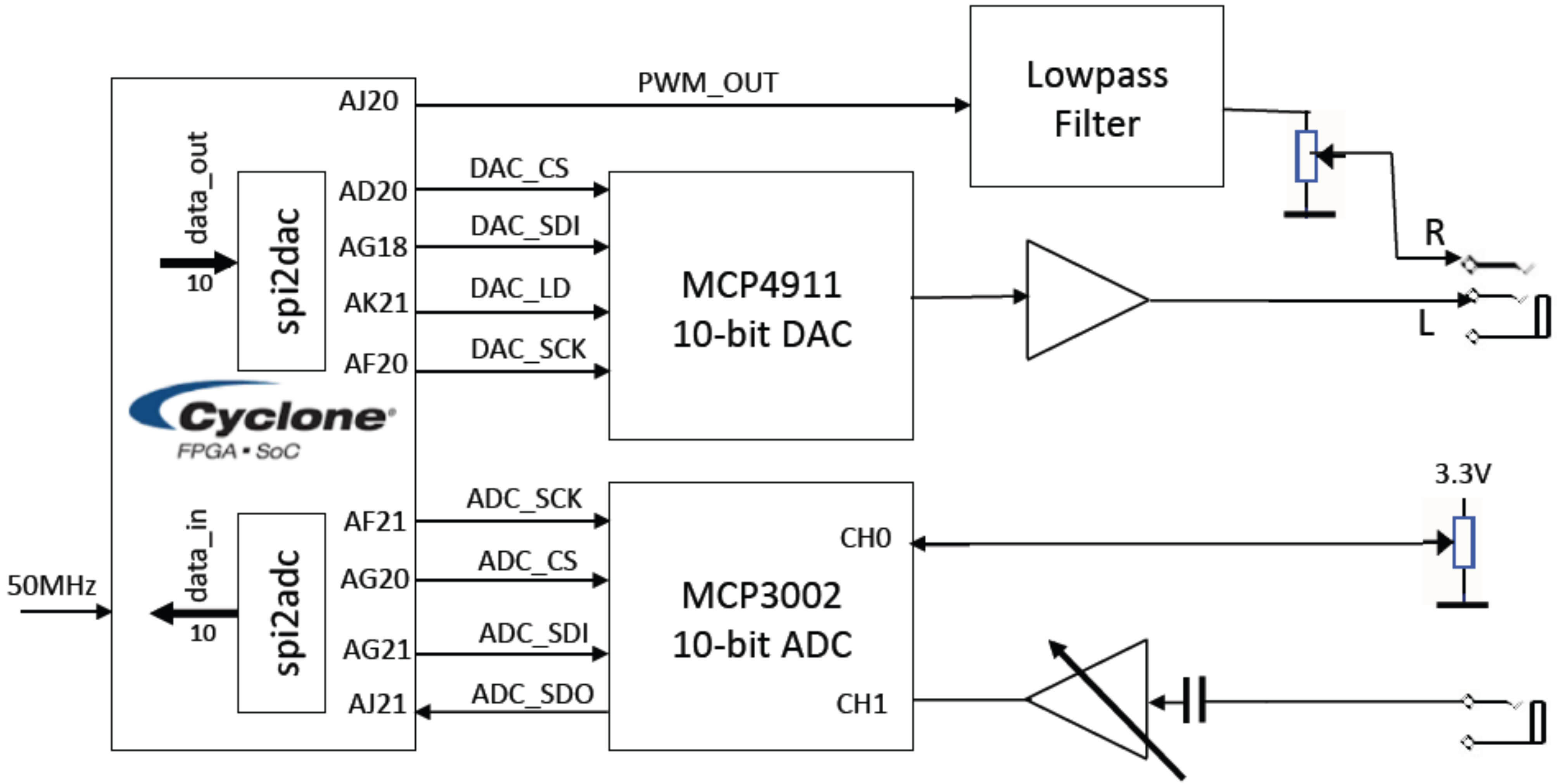
The Analogue I/O Card

- Provides analogue inputs and outputs
- Contains 2 channels ADC, one for a dc voltage set by a potentiometer & another from a socket
- Has 1 DAC to connected to the right channel, and a digital output to the left channel of a headphone socket
- Includes low-pass filter and operational amplifiers
- Will be using this board for Experiment: VERI part 3 and 4



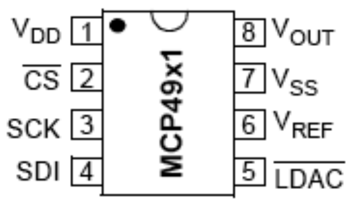


Schematic of the Analogue I/O Card



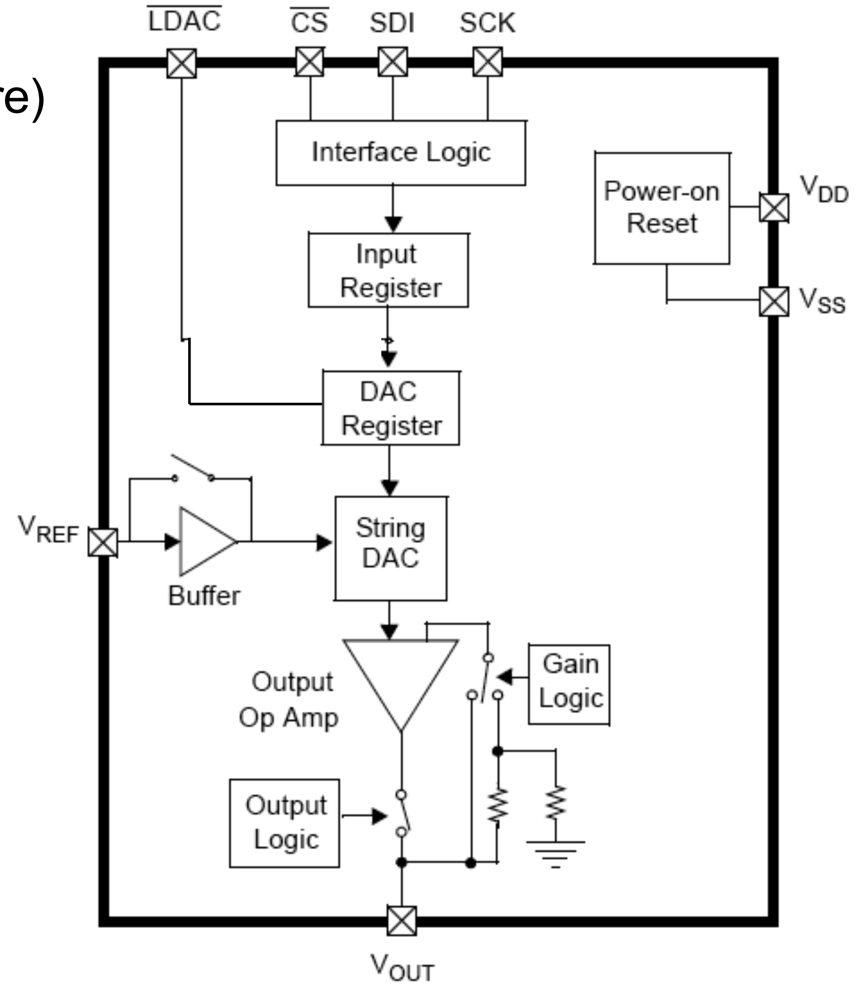
DAC – used in analogue I/O card

- **Microchip MCP4911** 10-bit DAC
- Uses **resistor string** architecture (earlier lecture)
- Serial Peripheral Interface (SPI)



- Rail-to-Rail Output
- SPI Interface with 20 MHz Clock Support
- Simultaneous Latching of the DAC Output with $\overline{LDAC}$ Pin
- Fast Settling Time of 4.5 µs
- Selectable Unity or 2x Gain Output
- External Voltage Reference Input
- External Multiplier Mode

Symbol	Description
V_{DD}	Supply Voltage Input (2.7V to 5.5V)
$\overline{CS}$	Chip Select Input
SCK	Serial Clock Input
SDI	Serial Data Input
$\overline{LDAC}$	DAC Output Synchronization Input. This pin is used to transfer the input register (DAC settings) to the output register (V_{OUT})
V_{REF}	Voltage Reference Input
V_{SS}	Ground reference point for all circuitry on the device
V_{OUT}	DAC Analog Output



Serial Peripheral Interface for DAC (SPI)

bit 15
- 0 = Write to DAC register
- 1 = Ignore this command

bit 14 **BUF:** V_{REF} Input Buffer Control bit
- 1 = Buffered
- 0 = Unbuffered

V_{REF} = 1.23V

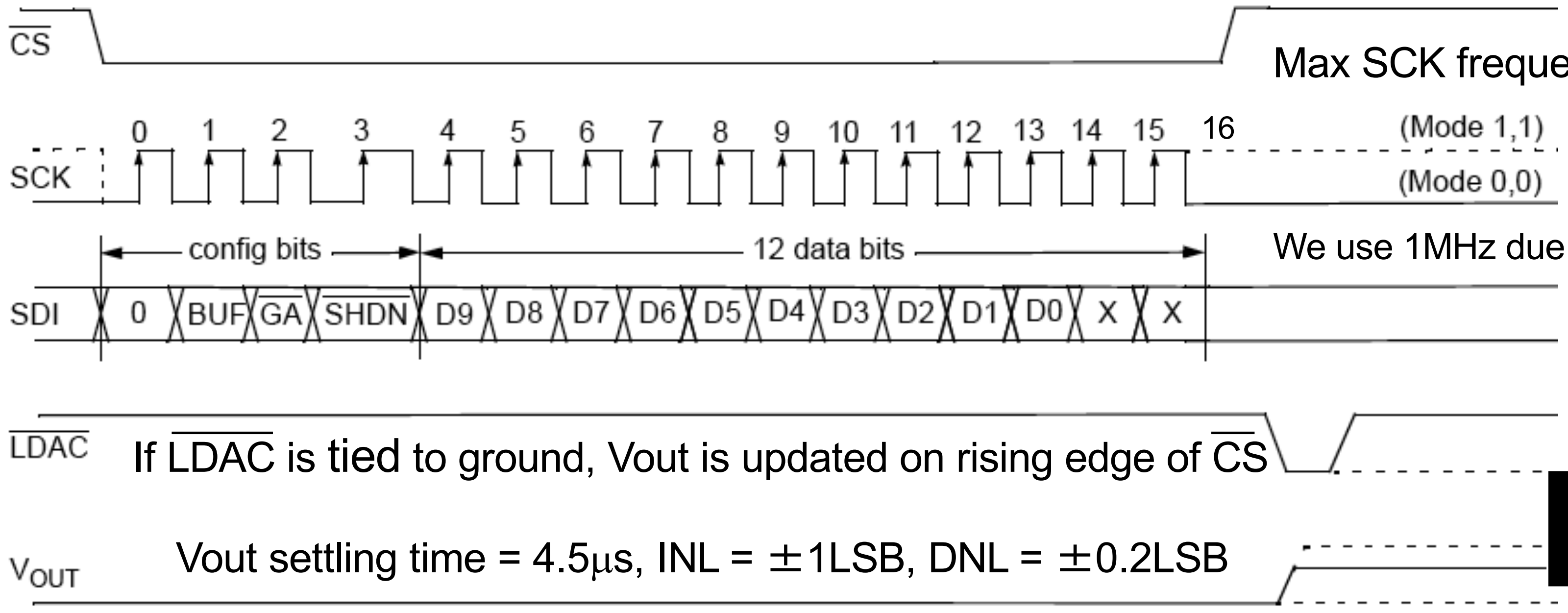
bit 13 **$\overline{\text{GA}}$:** Output Gain Selection bit
- 1 = 1x ($V_{OUT} = V_{REF} * D/4096$)
- 0 = 2x ($V_{OUT} = 2 * V_{REF} * D/4096$)

bit 12 **$\overline{\text{SHDN}}$:** Output Shutdown Control bit
- 1 = Active mode operation. VOUT is available.
- 0 = Shutdown the device.

bit 11-0 **D11:D0:** DAC Input Data bits. Bit x is ignored.

bit 11-2 **D9:D0:** DAC input data bit

$Vout = V_{REF} * (D[9:0]/1024)$



Max SCK frequency is 20MHz

We use 1MHz due to ADC limits.

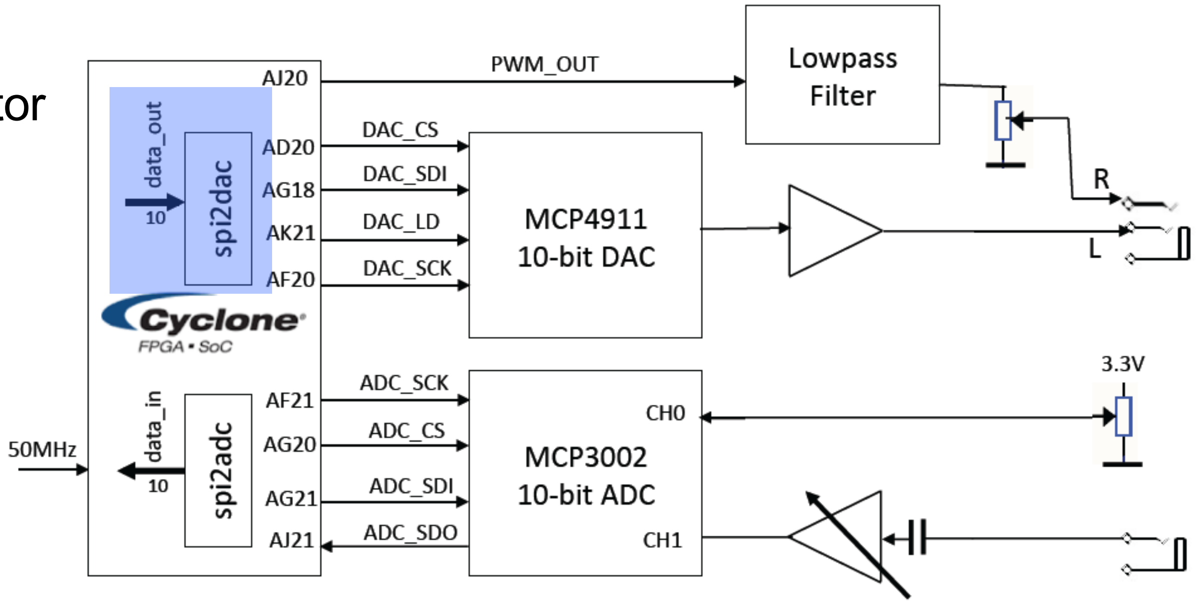
If $\overline{\text{LDAC}}$ is tied to ground, Vout is updated on rising edge of $\overline{\text{CS}}$

Vout settling time = 4.5μs, INL = ±1LSB, DNL = ±0.2LSB

datasheet p23-25

Interfacing the FPGA to the DAC and ADC

- Overview of the DAC/ADC
- DAC is DC coupled (no capacitor in signal path)
- ADC is AC coupled (why?)
- Interface circuit to DAC:
  - spi2dac.sv
- Interface circuit to ADC
  - spi2adc.sv



Important points to note

- DAC and ADC function are NOT done within Cyclone V FPGA
- Conversion from/to analogue signals are done with 2 8-pin chips on Add-on card
- Why do we need serial-parallel interface circuits? To fit everything within 8-pin package
- A single serial clock is used for both ADC and DAC – set at 1MHz
- This is different from the system clock of 50MHz (fixed within DE1-SoC)
- Chip-select is low only when sending serial data to DAC chip on SDI pin
- LDA is low only when all 10-bit data sent and DAC to be loaded with new value