Mastering Digital Design

with Verilog on FPGAs

John Wickerson

Lecture 3

This lecture

- Pulse-width modulation (PWM)
- Finite state machines (FSMs)
- The analogue add-on card

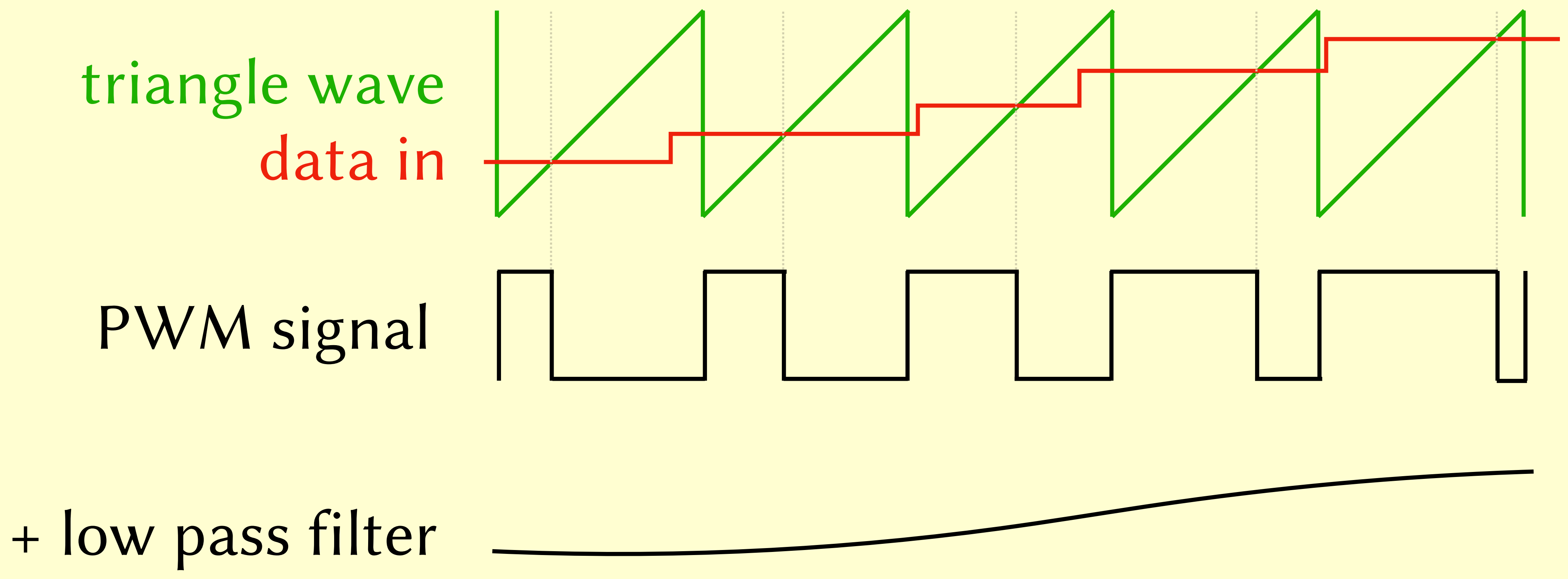
This lecture

- **Pulse-width modulation (PWM)**
- Finite state machines (FSMs)
- The analogue add-on card

DAC via PWM

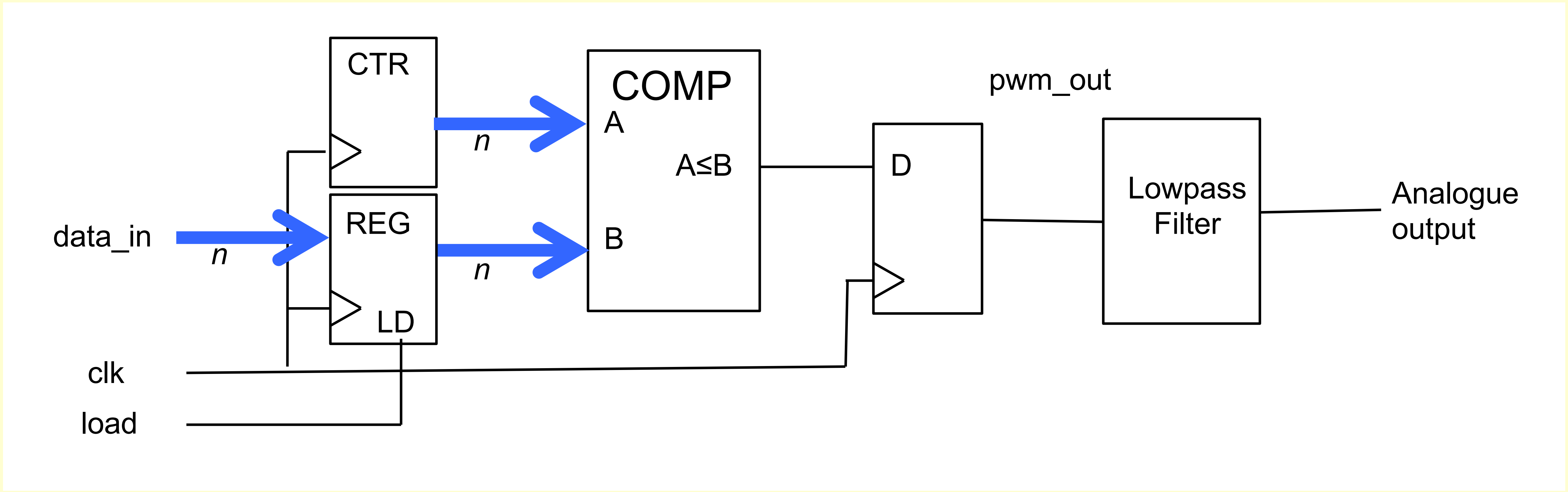
- **Digital-to-analogue conversion** (DAC) can be done using a network of resistors.

- It can also be done using digital components, via **pulse-width modulation** (PWM).



DAC via PWM

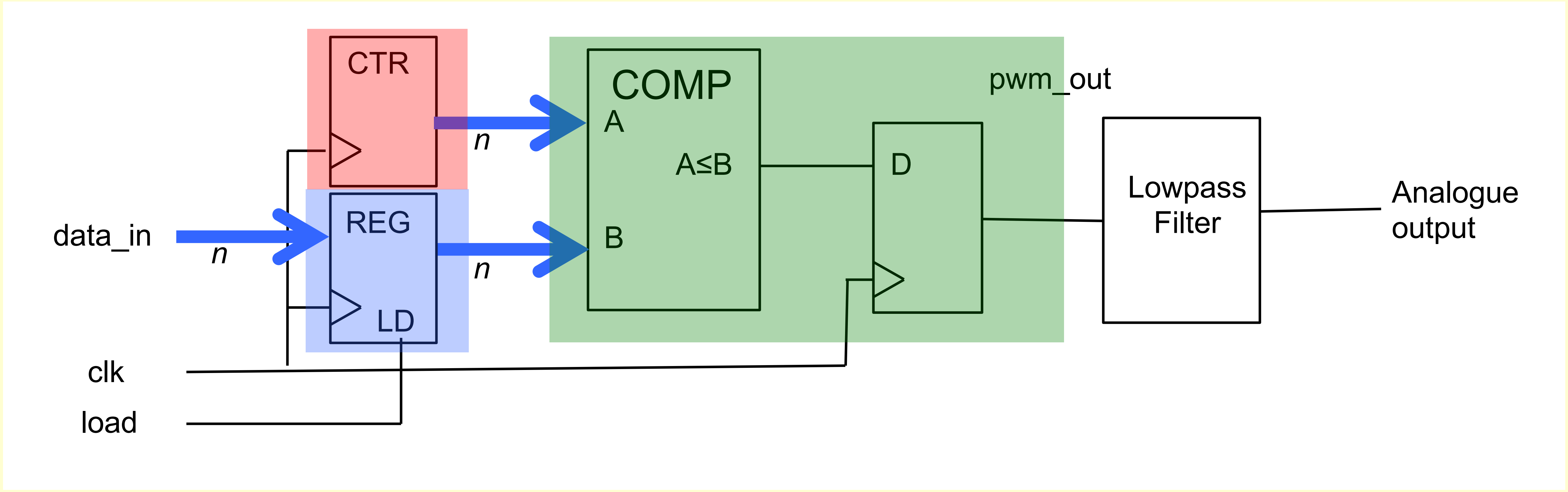
- **Digital-to-analogue conversion** (DAC) can be done using a network of resistors.
- It can also be done using digital components, via **pulse-width modulation** (PWM).



DAC via PWM

```
module pwm (clk, data_in, load, pwm_out);
  input        clk;
  input [9:0]  data_in;
  input        load;
  output       pwm_out;

  reg [9:0]    d;
  reg [9:0]    count;
  reg          pwm_out;

  initial count = 10'b0;
```

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

  always @(posedge clk) begin
    count <= count + 1'b1;
    if (count > d)
      pwm_out <= 1'b0;
    else
      pwm_out <= 1'b1;
  end
endmodule
```



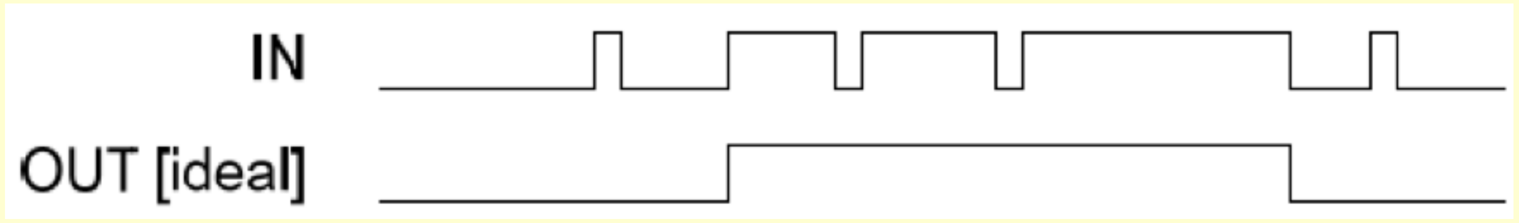
This lecture

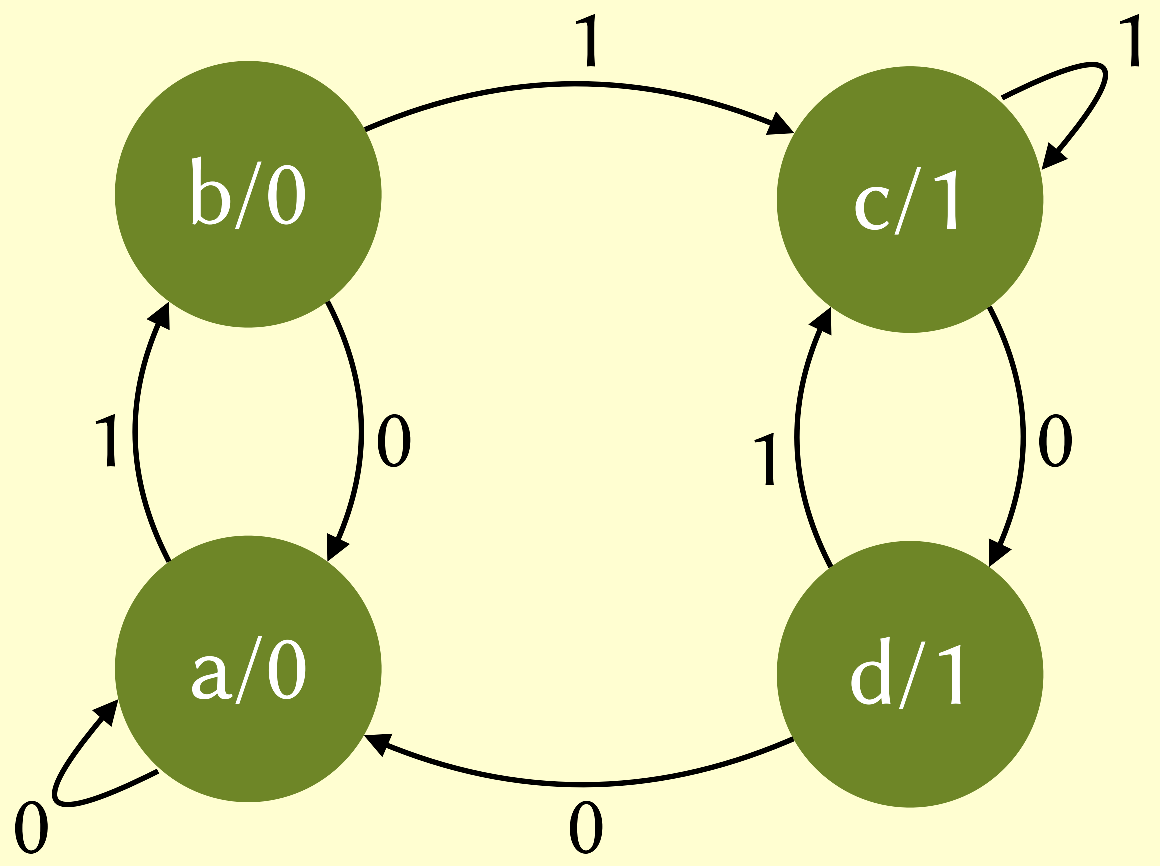
- Pulse-width modulation (PWM)
- Finite state machines (FSMs)
- The analogue add-on card

This lecture

- Pulse-width modulation (PWM)
- **Finite state machines (FSMs)**
- The analogue add-on card

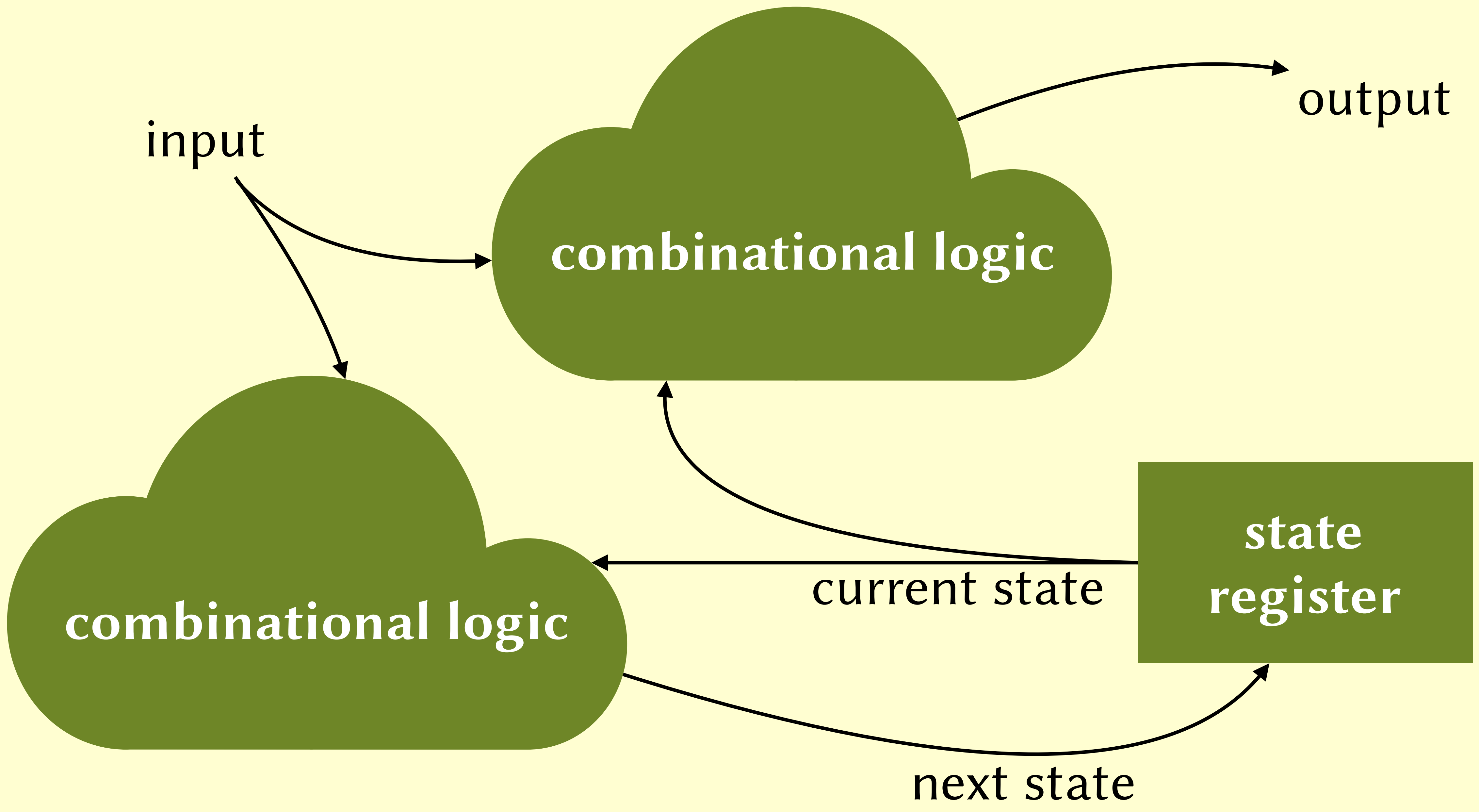
Example 1: denoiser

- Task: 

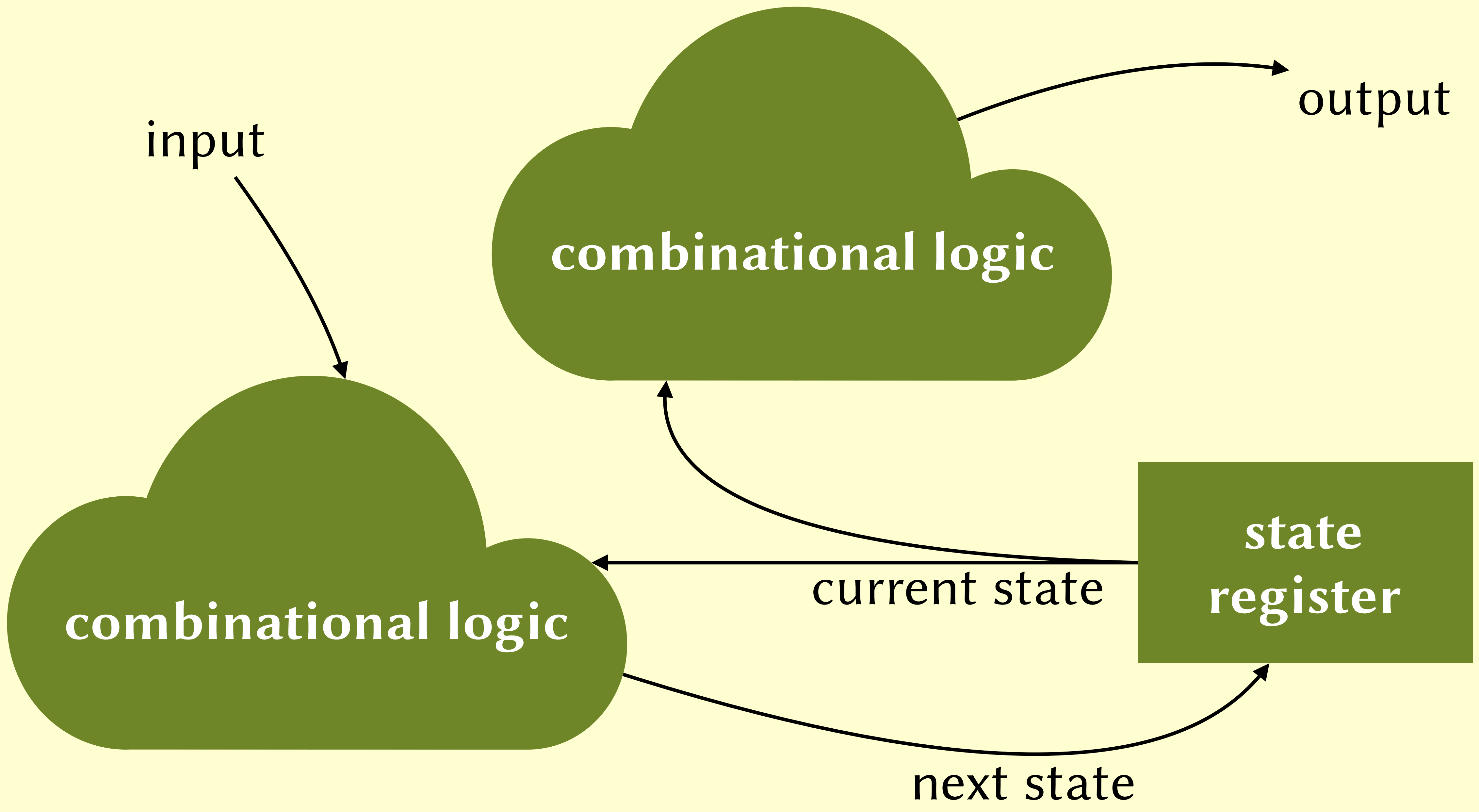

$$\begin{tikzcd}
b/0 \arrow[r, bend left, "1"] \arrow[d, bend left, "0"] & c/1 \arrow[loop right, "1"] \arrow[d, bend left, "0"] \\
a/0 \arrow[u, bend left, "1"] \arrow[loop left, "0"] & d/1 \arrow[u, bend left, "1"] \arrow[l, bend left, "0"]
\end{tikzcd}$$

state	meaning	output
a	"definitely low"	0
b	"maybe rising"	0
c	"definitely high"	1
d	"maybe falling"	1

Mealy machines



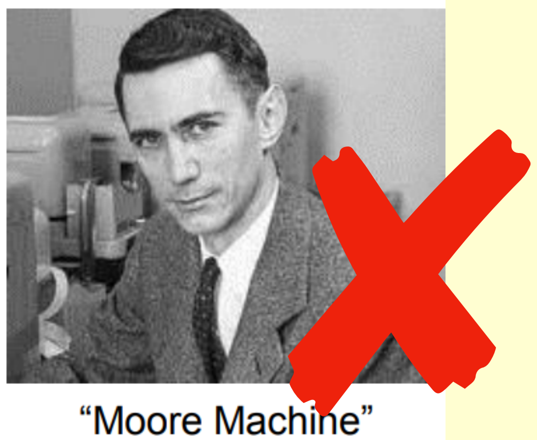
Moore machines



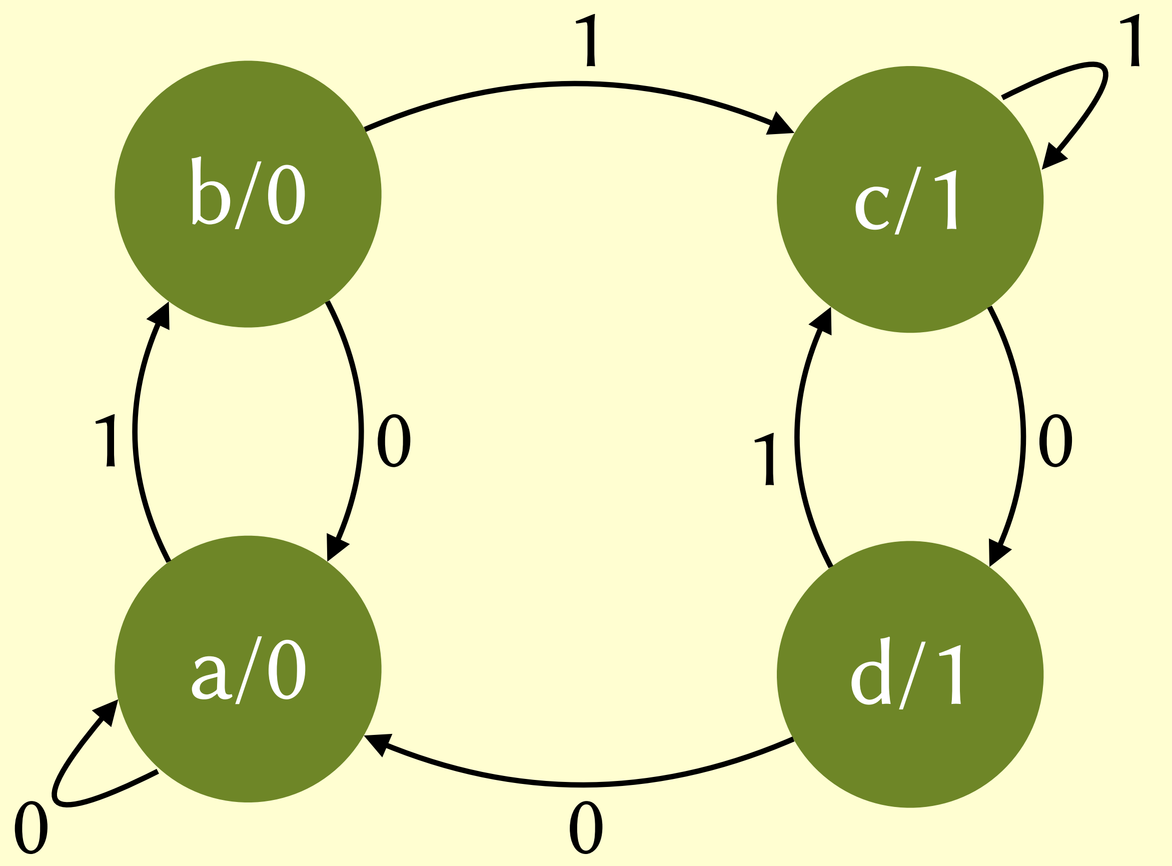
Mealy vs Moore

- Mealy machines are a bit more complicated to design.
- Moore machines tend to require more states.
- Outputs of Moore machines are delayed.
- Named after George Mealy (1927–2010) and Edward Moore (1925–2003).

"Mealy Machine" developed in 1955 by George H. Mealy


"Moore Machine" developed in 1956 by Edward F. Moore

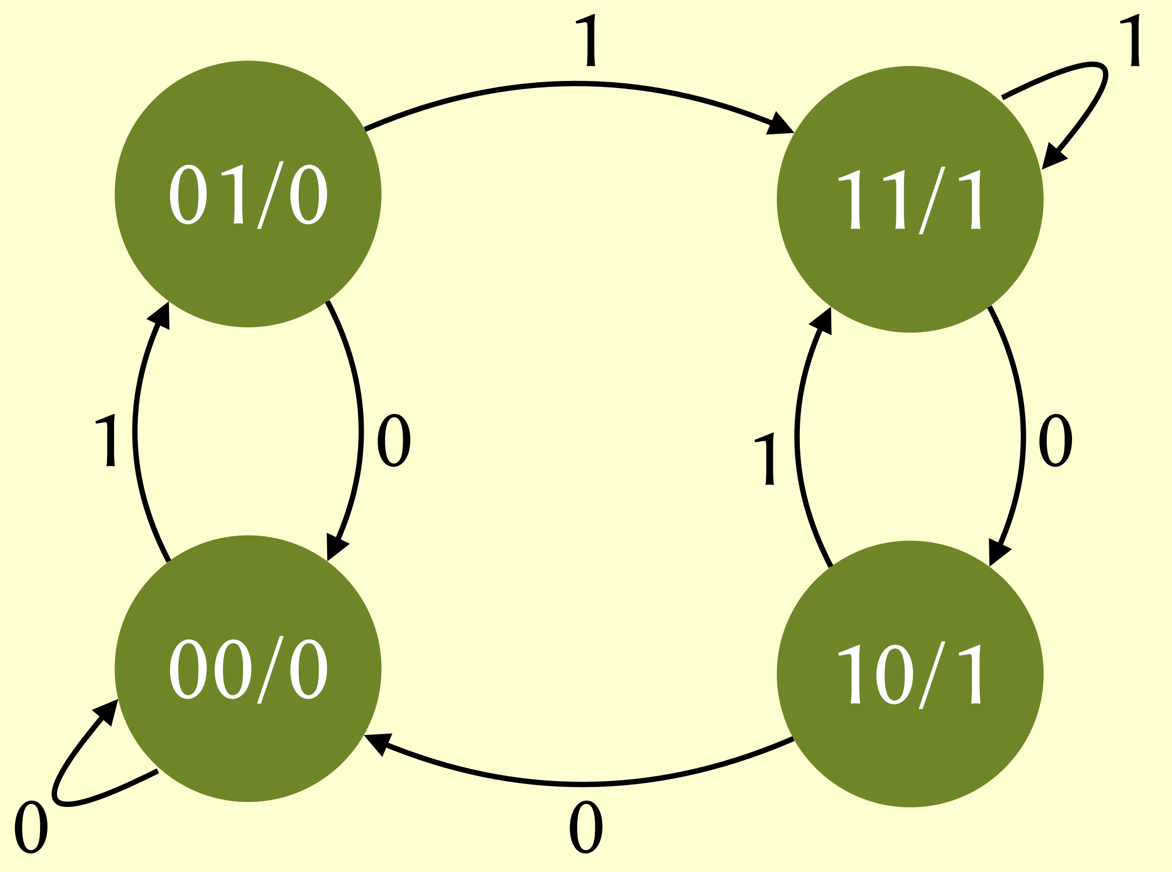
Implementing the FSM



state	meaning	output
a	"definitely low"	0
b	"maybe rising"	0
c	"definitely high"	1
d	"maybe falling"	1

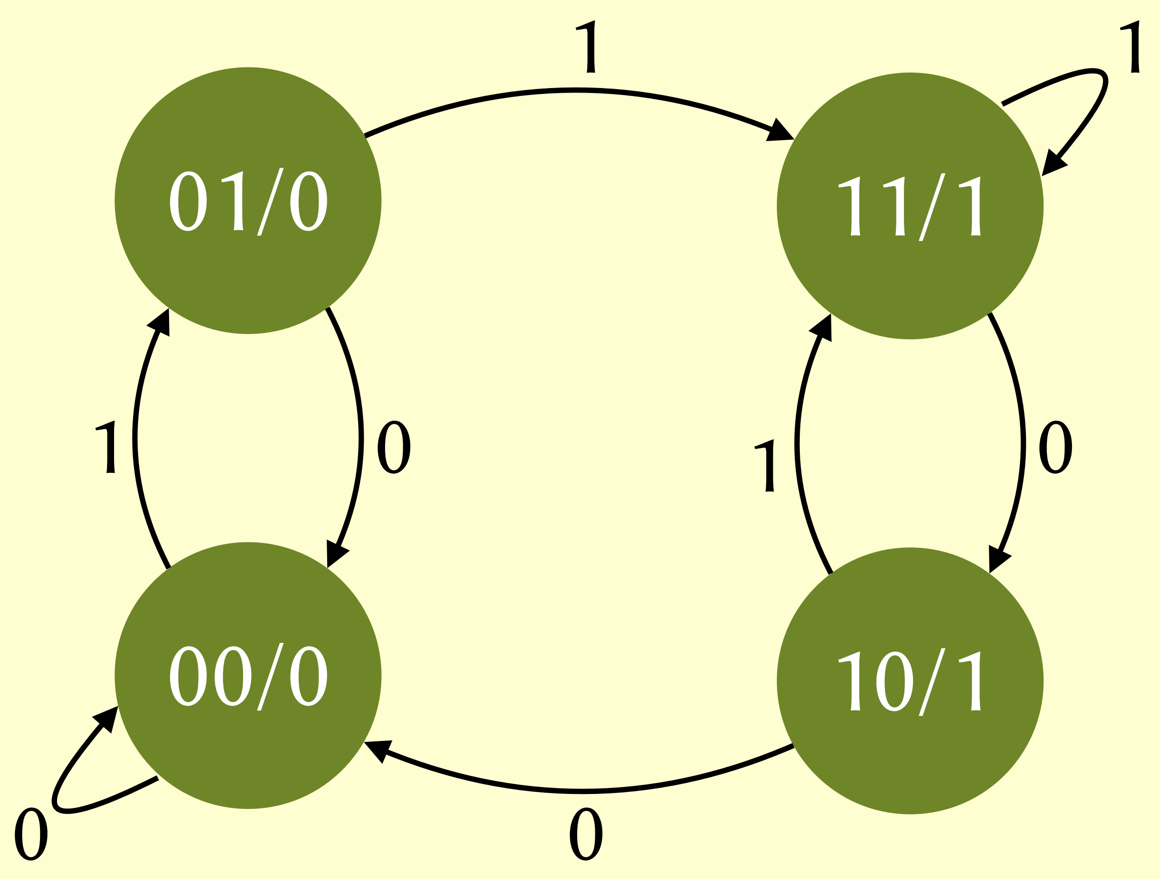
Implementing the FSM

- Try to make the output value and the state number similar.
- Try to change one bit of the state number per transition.
- If two parts of the FSM have identical transitions, try to give the corresponding states similar numbers



state	meaning	output
00	"definitely low"	0
01	"maybe rising"	0
11	"definitely high"	1
10	"maybe falling"	1

Implementing the FSM



state `s[1:0]`	meaning	output `out`	input `in`	next state `ns[1:0]`
00	"definitely low"	0	0	00
			1	01
01	"maybe rising"	0	0	00
			1	11
11	"definitely high"	1	0	10
			1	11
10	"maybe falling"	1	0	00
			1	11

Implementing the FSM

```
assign out = s[1];
assign ns[1] = s[1] & s[0] | in & (s[0] | s[1]);
```

ns[1]	in=0	in=1
s=00	0	0
s=01	0	1
s=11	1	1
s=10	0	1

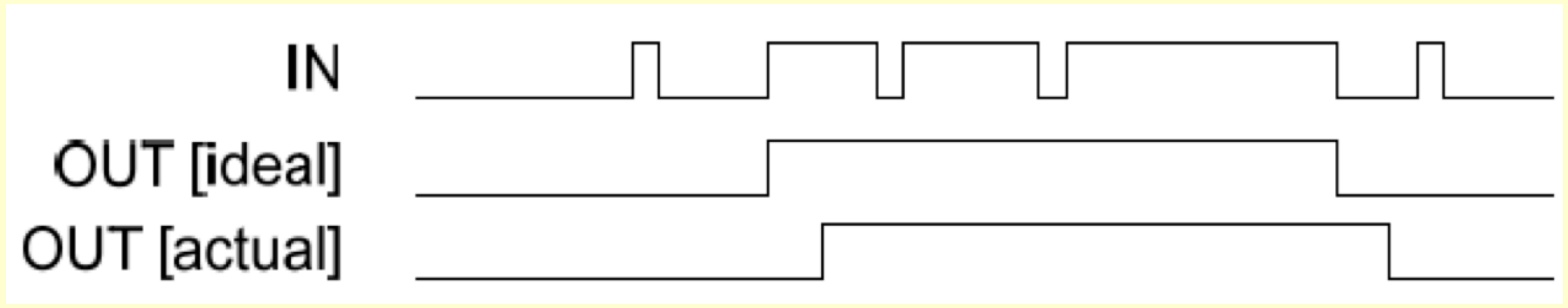
state s[1:0]	meaning	output out	input in	next state ns[1:0]
00	"definitely low"	0	0	00
			1	01
01	"maybe rising"	0	0	00
			1	11
11	"definitely high"	1	0	10
			1	11
10	"maybe falling"	1	0	00
			1	11

Implementing the FSM

```
assign out = s[1];
assign ns[1] = s[1] & s[0] | in & (s[0] | s[1]);
assign ns[0] = in;
```

ns[0]	in=0	in=1
s=00	0	1
s=01	0	1
s=11	0	1
s=10	0	1

state s[1:0]	meaning	output out	input in	next state ns[1:0]
00	"definitely low"	0	0	00
			1	01
01	"maybe rising"	0	0	00
			1	11
11	"definitely high"	1	0	10
			1	11
10	"maybe falling"	1	0	00
			1	11



```
assign out = s[1];
assign ns[1] = s[1] & s[0] | in & (s[0] | s[1]);
assign ns[0] = in;
```

ns[0]	in=0	in=1
s=00	0	1
s=01	0	1
s=11	0	1
s=10	0	1

state s[1:0]	**meaning**	**output** out	**input** in	**next state** ns[1:0]
00	"definitely low"	0	0	00
			1	01
01	"maybe rising"	0	0	00
			1	11
11	"definitely high"	1	0	10
			1	11
10	"maybe falling"	1	0	00
			1	11

One-hot encoding

- "One-hot" means "exactly one bit of a binary number is 1".

state	encoding as a binary number	one-hot encoding
a	00	0001
b	01	0010
c	11	0100
d	10	1000

- One-hot encoding uses more bits, but can lead to simpler logic.
- **Exercise.** Re-implement the FSM using one-hot encoding. Is the logic simpler?

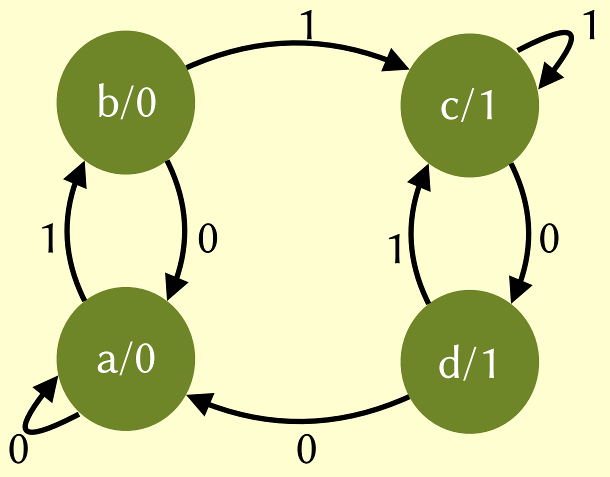
Automated FSM design

- We usually design FSMs at a higher level of abstraction.

declare states

calculate next state

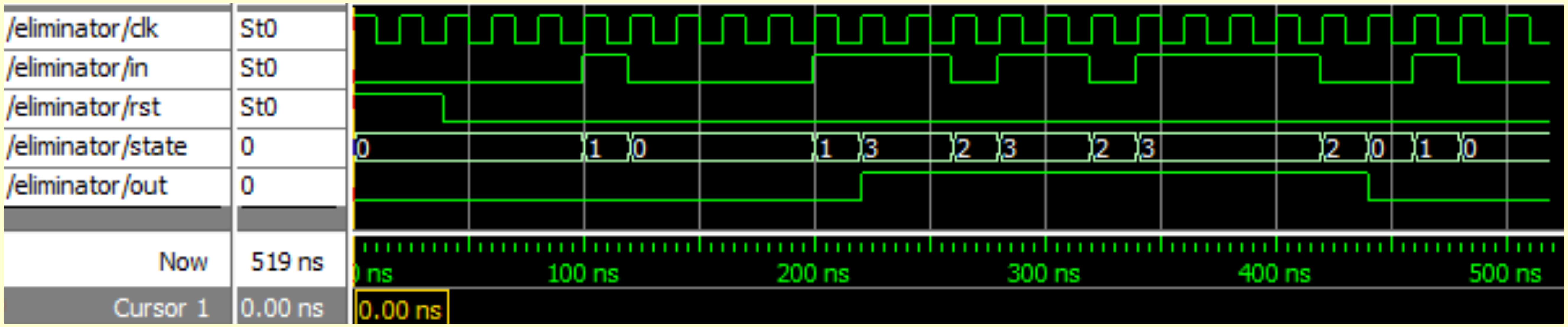
```
module denoiser (out, in, clk, rst);
  input in, clk, rst; output out;
  parameter sA = 4'b0001;
  parameter sB = 4'b0010;
  parameter sC = 4'b0100;
  parameter sD = 4'b1000;
  reg [3:0] s; initial s = sA;
  reg out;     initial out = 1'b0;

  always @ (posedge clk)
    if (rst==1'b1) s <= sA;
    else case (s)
      sA: if (in==1'b1) s <= sB;
      sB: if (in==1'b1) s <= sC;
          else          s <= sA;
      sC: if (in==1'b0) s <= sD;
      sD: if (in==1'b1) s <= sC;
          else          s <= sA;
    endcase
```



```
  always @ *
    case (s)
      sA: out = 1'b0;
      sB: out = 1'b0;
      sC: out = 1'b1;
      sD: out = 1'b1;
    endcase
endmodule
```

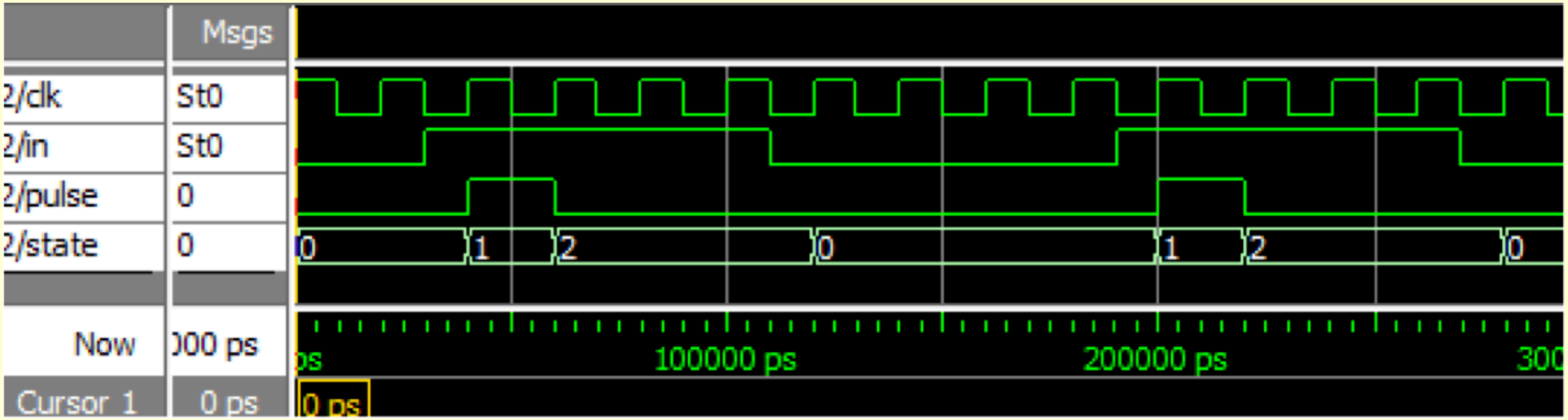
calculate output

The result


/eliminator/clk
St0
/eliminator/in
St0
/eliminator/rst
St0
/eliminator/state
0
/eliminator/out
0
0 1 0 1 3 2 3 2 3 2 0 1 0
Now
519 ns
Cursor 1
0.00 ns
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
0.00 ns

Example 2: pulser

- Task: Whenever the input rises, output a 1-cycle pulse.



$$\begin{tikzcd}
\text{idle/0} \arrow[loop left, "0"] \arrow[r, bend left, "1"] & \text{high/1} \arrow[l, bend left, "0"] \arrow[r, bend left, "1"] & \text{wait/0} \arrow[loop right, "1"] \arrow[ll, bend left=40, "0"]
\end{tikzcd}$$

Example 2: pulser

declare states

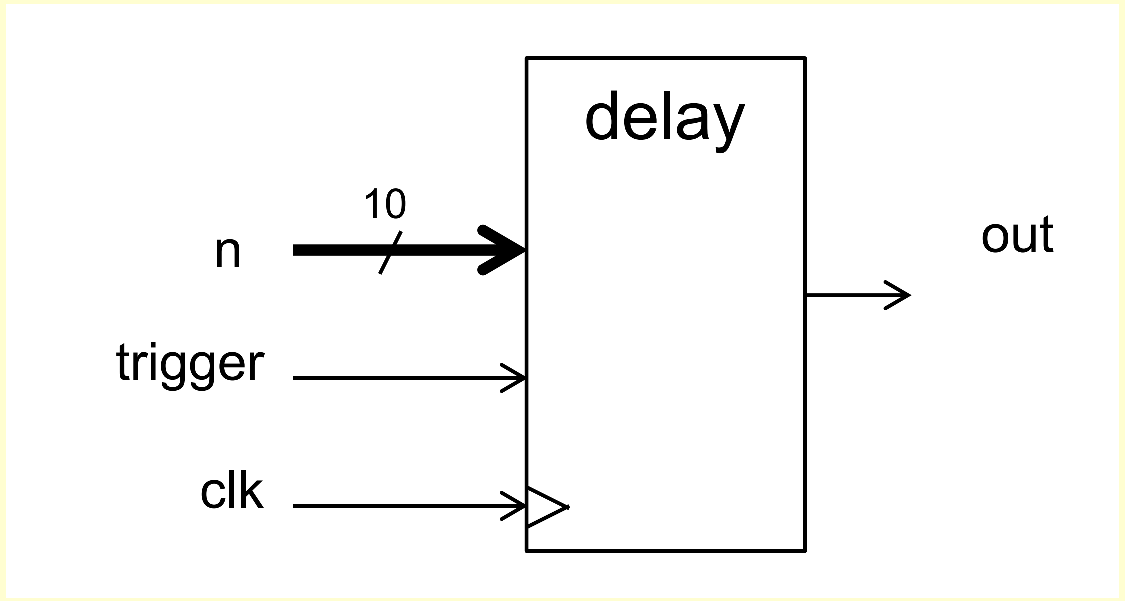
calculate next state

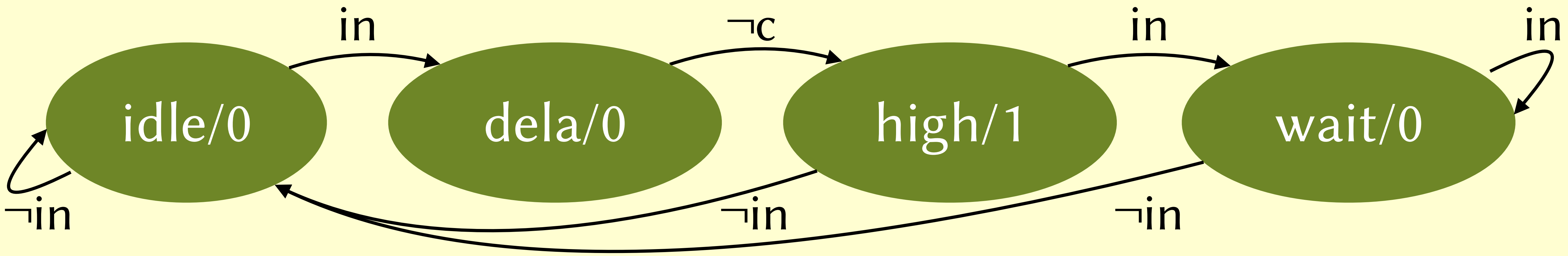
calculate output

```
module pulser (out, in, clk);
  input in, clk; output out;
  parameter sIDLE = 2'b00;
  parameter sHIGH = 2'b01;
  parameter sWAIT = 2'b10;
  reg [1:0] s; initial s = sIDLE;
  reg out;     initial out = 1'b0;

  always @ (posedge clk)
    case (s)
      sIDLE: if (in==1'b1) s <= sHIGH;
      sHIGH: if (in==1'b1) s <= sWAIT;
             else          s <= sIDLE;
      sWAIT: if (in==1'b0) s <= sIDLE;
    endcase

  always @ *
    case (s)
      sIDLE: out = 1'b0;
      sHIGH: out = 1'b1;
      sWAIT: out = 1'b0;
    endcase
endmodule
```

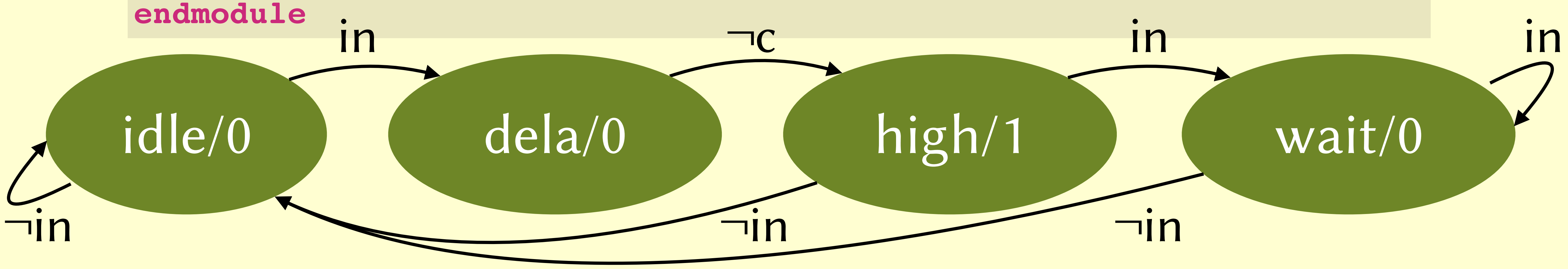
Example 3: delayer

- Task: when the 'trigger' input rises, wait for n cycles, then produce a 1-cycle pulse.





```
module delayer (out, in, n, clk);
  input in, clk; input [9:0] n; output out;
  parameter sIDLE = 2'b00;  parameter sDELA = 2'b01
  parameter sHIGH = 2'b11;  parameter sWAIT = 2'b10;
  reg [1:0] s; initial s = sIDLE;
  reg out;     initial out = 1'b0;
  reg [9:0] c;

  always @ (posedge clk)
    case (s) sIDLE: if (in==1'b1) s <= sDELA;
             sDELA: if (c==0) begin c <= n - 1'b1; s <= sHIGH; end
                    else            c <= c - 1'b1;
             sHIGH: if (in==1'b1) s <= sWAIT;
                    else          s <= sIDLE;
             sWAIT: if (in==1'b0) s <= sIDLE;
    endcase

  always @ *
    case (s) sIDLE: out = 1'b0;
             sDELA: out = 1'b0;
             sHIGH: out = 1'b1;
             sWAIT: out = 1'b0;
    endcase
endmodule
```



This lecture

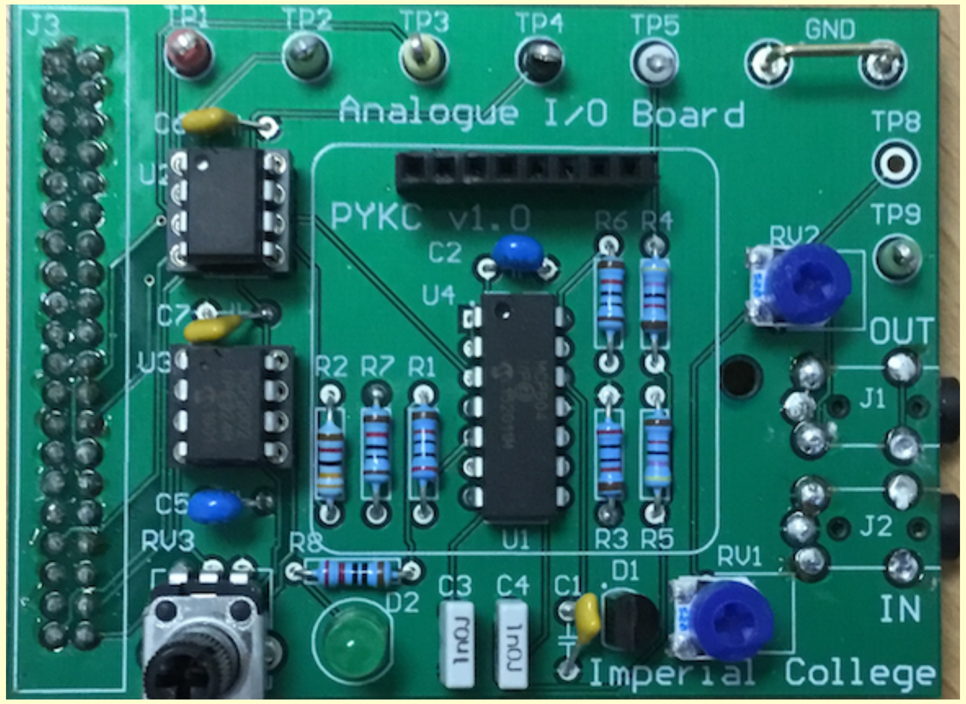
- Pulse-width modulation (PWM)
- Finite state machines (FSMs)
- The analogue add-on card

This lecture

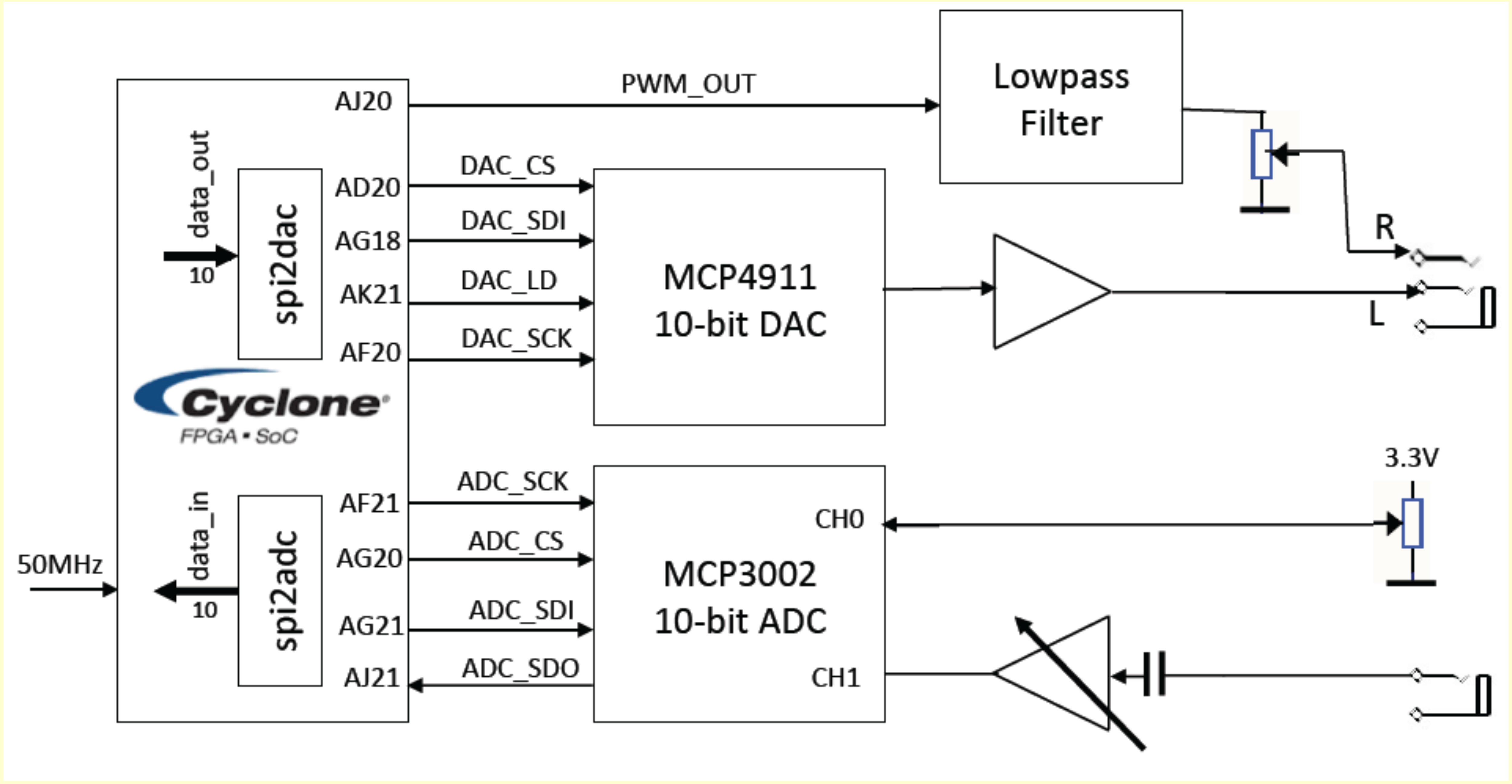
- Pulse-width modulation (PWM)
- Finite state machines (FSMs)
- **The analogue add-on card**

The analogue I/O card

- Provides analogue inputs and outputs.
- Contains an ADC, a DAC, a low-pass filter, and an op-amp.
- Will be used for parts 3 and 4 of the experiment.

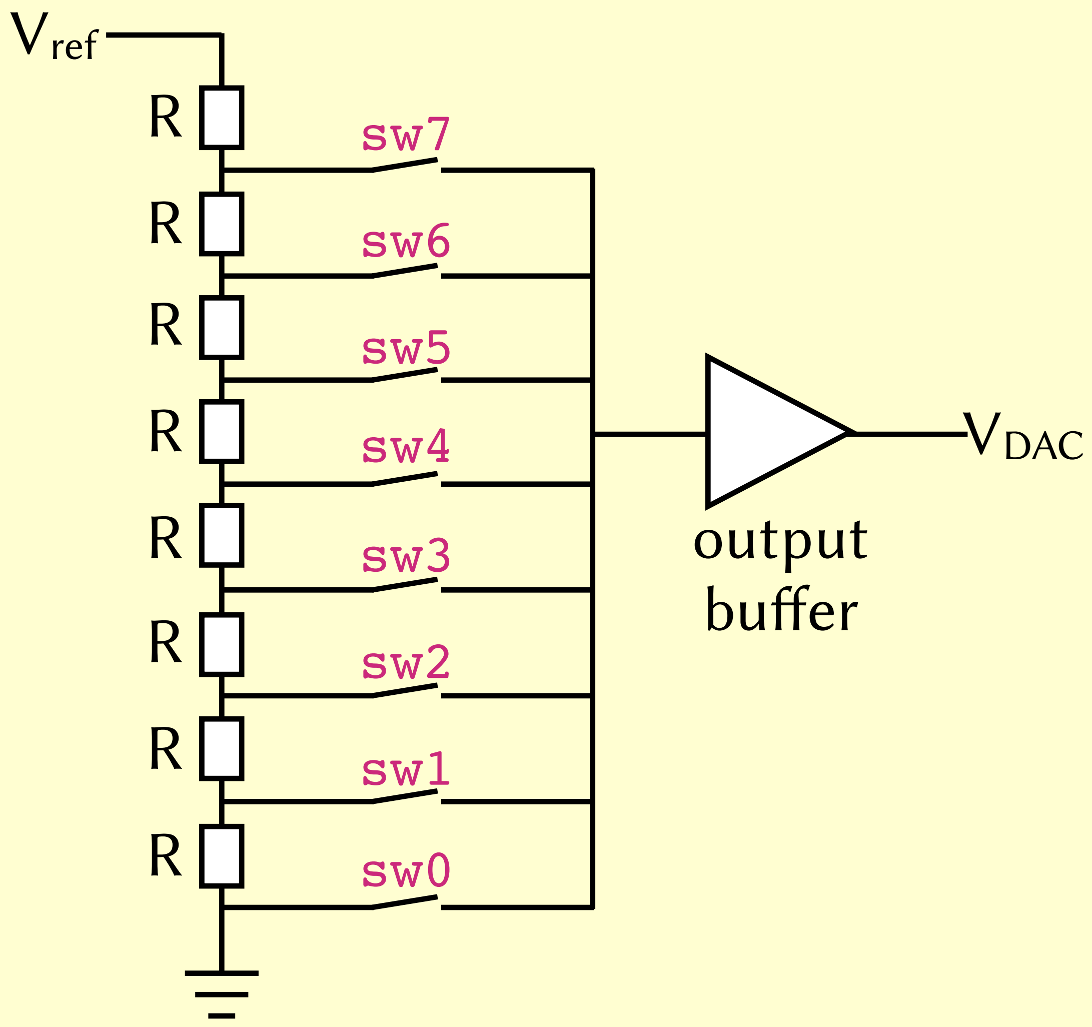


Schematic


PWM_OUT
AJ20
Lowpass Filter
data_out
10
spi2dac
AD20
DAC_CS
AG18
DAC_SDI
AK21
DAC_LD
AF20
DAC_SCK
MCP4911 10-bit DAC
R
L
Cyclone FPGA • SoC
3.3V
data_in
10
spi2adc
50MHz
AF21
ADC_SCK
AG20
ADC_CS
AG21
ADC_SDI
AJ21
ADC_SDO
MCP3002 10-bit ADC
CH0
CH1

DAC

- The DAC component uses a **resistor string** architecture



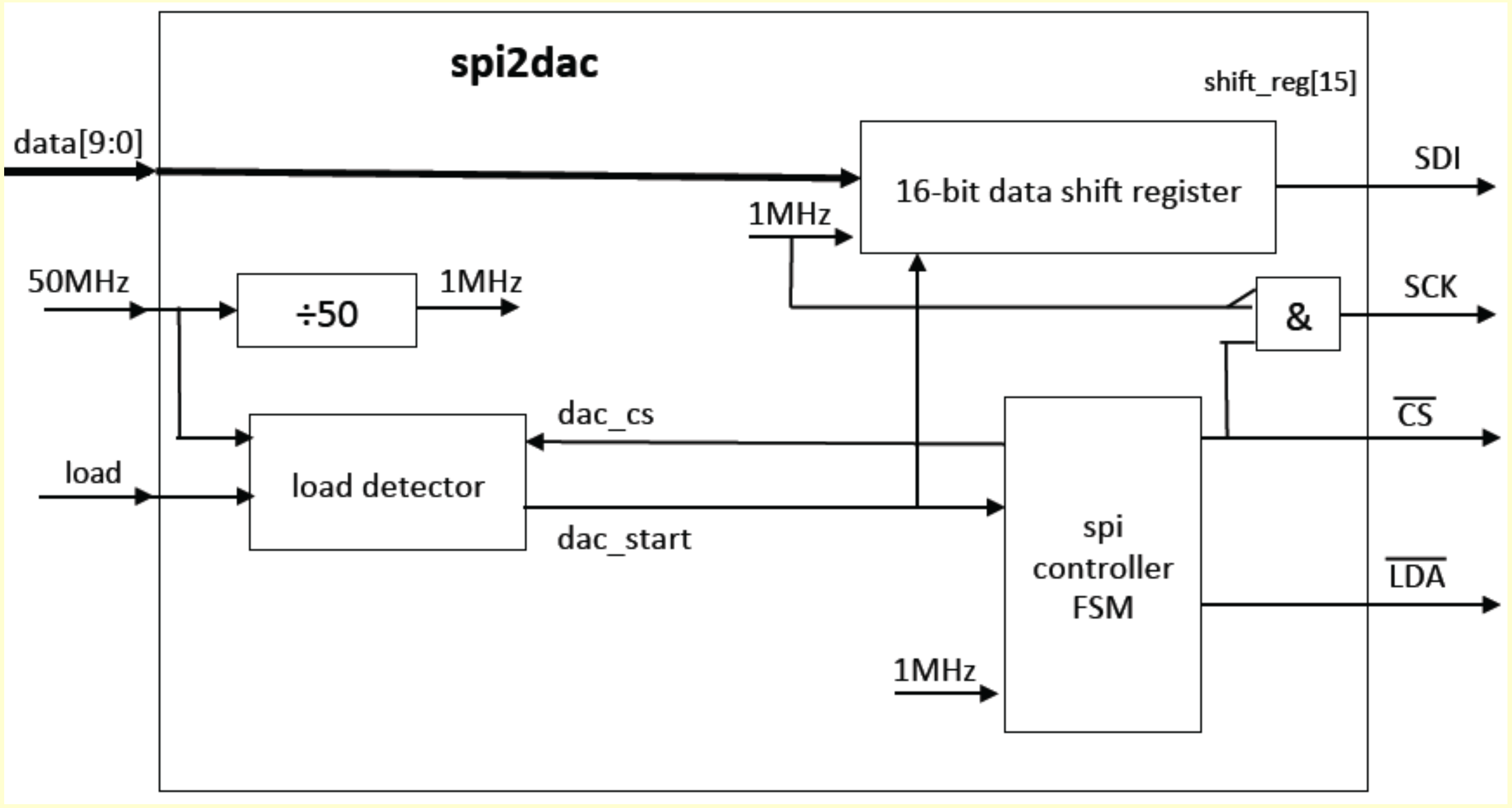
- Here is a 3-bit DAC. It uses $2^3 = 8$ resistors. (The add-on card has a 10-bit DAC.)
- When the digital input is 5, we close (only) switch `sw5`, which means V_{DAC} is $\frac{5}{8}V_{ref}$.

Series-parallel interface for DAC

- To send a value to the DAC, 16 bits are transmitted **serially**.
- **Chip select** (SC) going low means the start of transmission.
- Then **serial data in** (SDI) takes the following sequence of values.

bit	name	interpretation
15		always 0
14	BUF	whether V_{ref} is buffered
13	¬GA	high = 1x gain, low = 2x gain
12	¬SHDN	high = normal, low = shutdown
11–2		the 10 bits of actual data
1–0		these are ignored

spi2dac



- More on this circuit next week!