Advanced Digital Design

Session1 – Digital Design with System Verilog & FSM

Lucas Valentín
Ruben Salvador

Introduction

- Instructors
  - Lucas Valentín - *Digital Design Engineer at Analog Devices*
  - Rubén Salvador – *Digital Design Engineer at Analog Devices*
- Lecture + Lab in each session
- Evaluation based on Labs work

Session 1	Digital Design with SystemVerilog + Lab1 *(Lucas/Rubén)*
Session 2	Multi-Clock Digital Design + Lab2 *(Rubén)*
Session 3	Digital Synthesis and DFT + Lab3 *(Lucas)*
Session 4	Digital PnR Implementation + Lab4 *(Lucas)*
Session 5	Low Power and Timing Analysis + Lab5 *(Rubén)*

About Lucas Valentín

Academical

- MsC Electrical Engineering (Ingeniero Telecomunicaciones esp. Electrónica) '07 (UPV)
- MsC Natural Language Technologies '22 (UNED)

Professional

- Digital Design @ Analog Devices RF group '06 (Cork, Ireland)
- Digital Design @ Analog Devices Video group '07 (València)
- ML and Algorithms engineer @ Analog Devices Medical Products group '20 (València)
- Digital Design @ Analog Devices Automotive group '25 (València)

Teaching

- Digital IC design seminar, MUISE UPV 2020-2024

About Rubén Salvador

Academical

- MsC Electrical Engineering (Ingeniero Telecomunicaciones esp. Electrónica) '07 (UPV)
- MsC Integrated Systems Engineering, MUISE '09 (UPV)

Professional

- Digital Design @ Analog Devices Video group '07 (Limerick, Ireland)
- Digital Design @ Analog Devices IoT Low Power group '09 (València)
- Digital Design @ Analog Devices Consumer group '15 (València)

Teaching

- Digital IC design seminar, MUISE UPV 2015-2019

Session1 – Digital Design with System Verilog

1. **Introduction**
2. HDL description of digital circuits in SystemVerilog
3. Finite State Machines in SystemVerilog
4. Common Digital Blocks

Lab 1

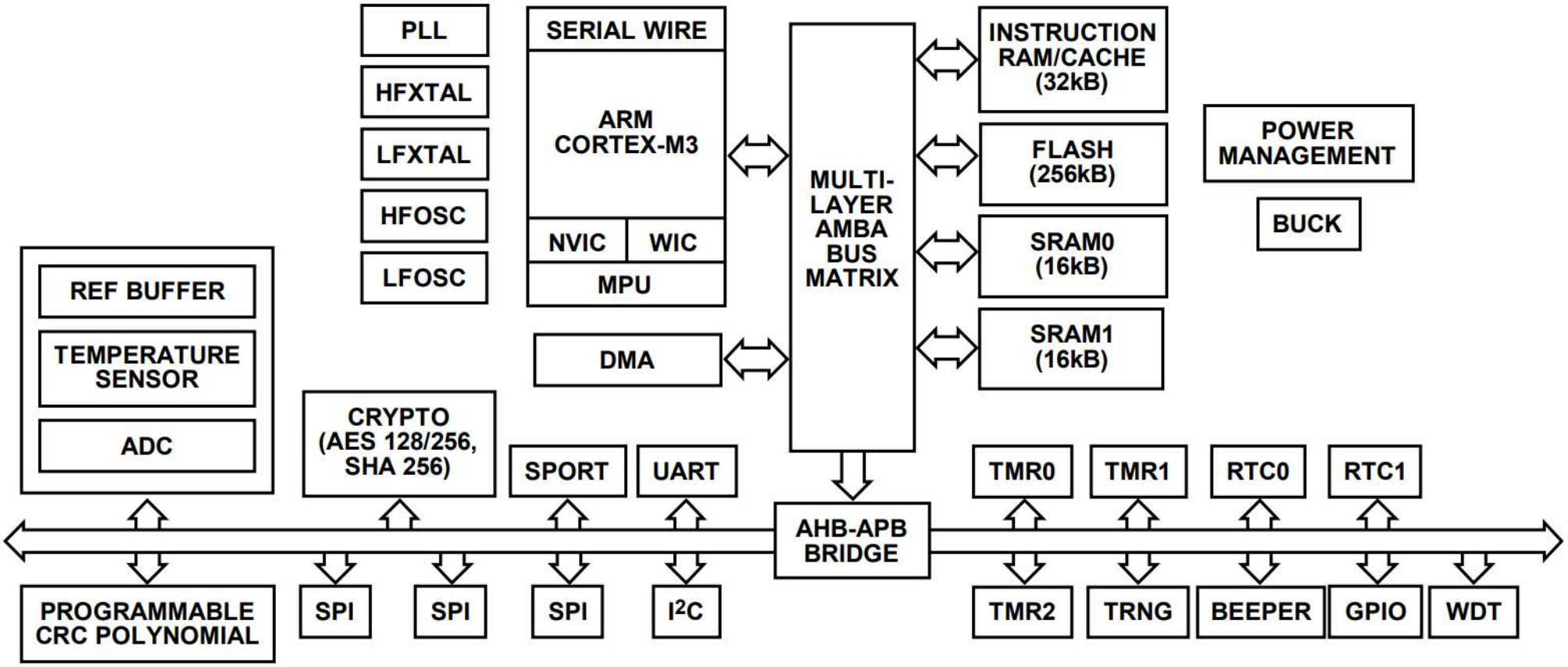
Course Motivation and Goals

- Digital Design is typically learned by implementing synchronous datapaths and FSMs in FPGA. FPGA's offer a variety of components and resources are already available to use, like clock sources and clock management macrocells, memory resources, dsp macros, serial interfaces, etc.

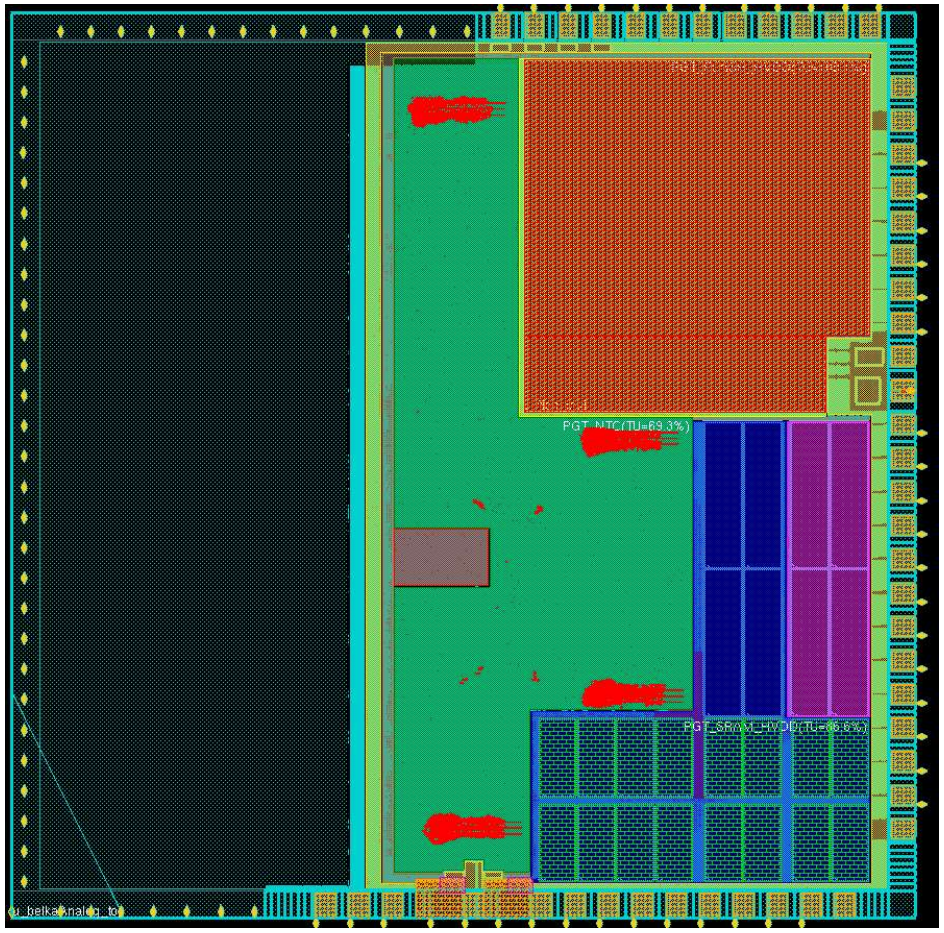
- ASIC designs are optimized for Performance, Power & Area. All blocks in an ASIC need to be designed and analyzed. Digital designs can be made of different subsystems, operating at different speed, and even at different voltage.

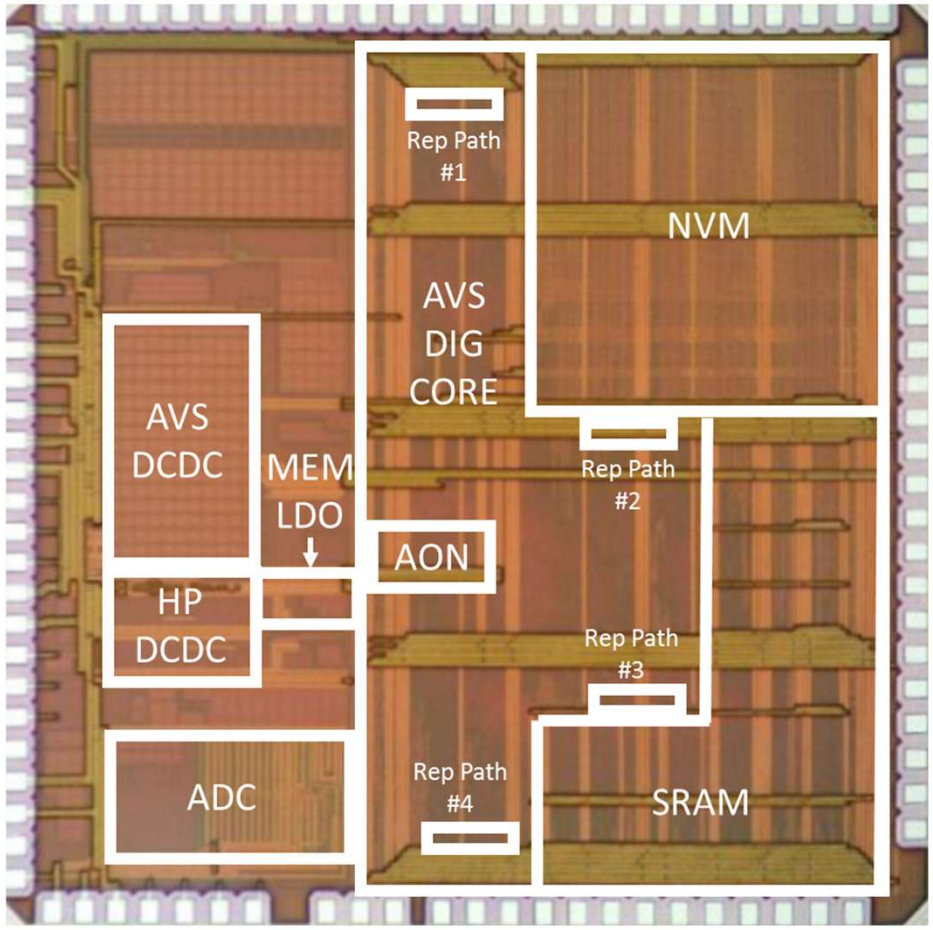
- The topics covered in this course are common challenges in ASIC design (also among the most demanded skills):
  - Timing Sing-off and Clocking
  - Clock Domain Crossing design & sign-off
  - Power Reduction Techniques
  - Constraint definition, Synthesis, DFT, PNR and Sign-off
  - Static Timing Analysis

Clocking and Power Management in ASICs

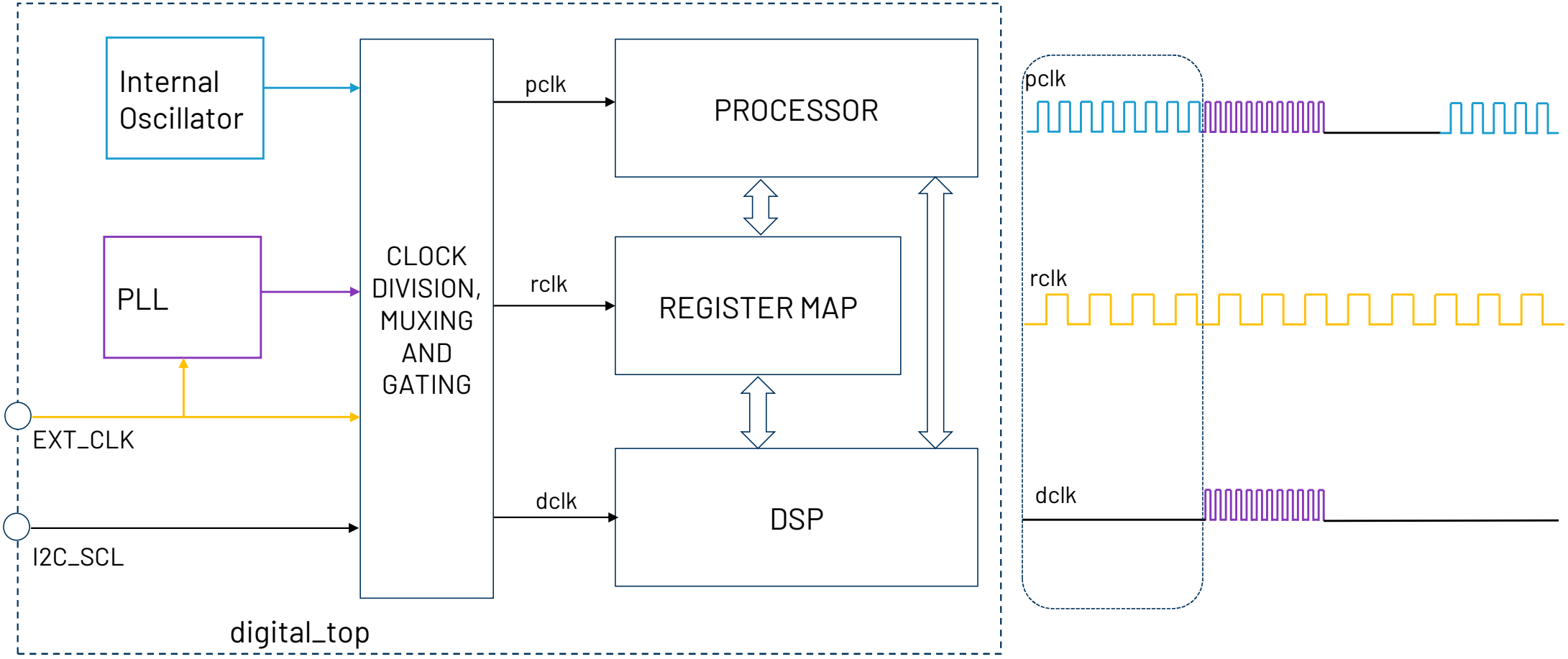


Clocking and Power Management in ASICs

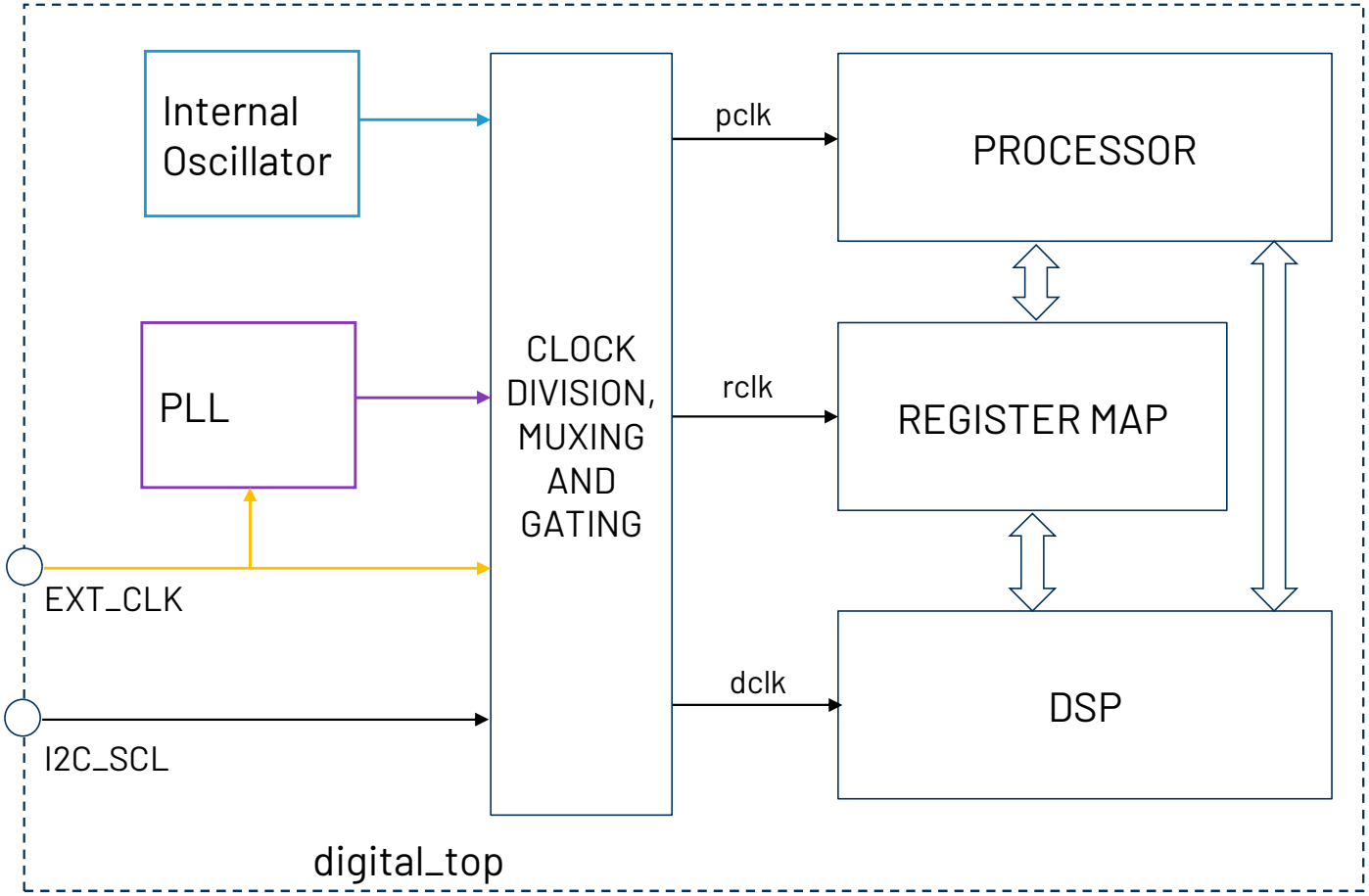


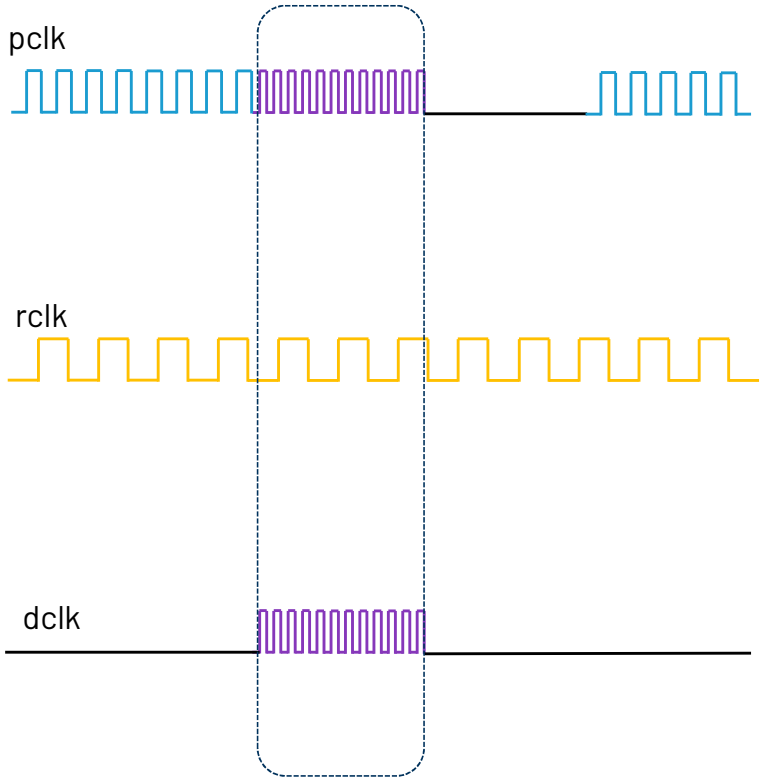


Multiple Clock Sources and Multiple Operation Modes

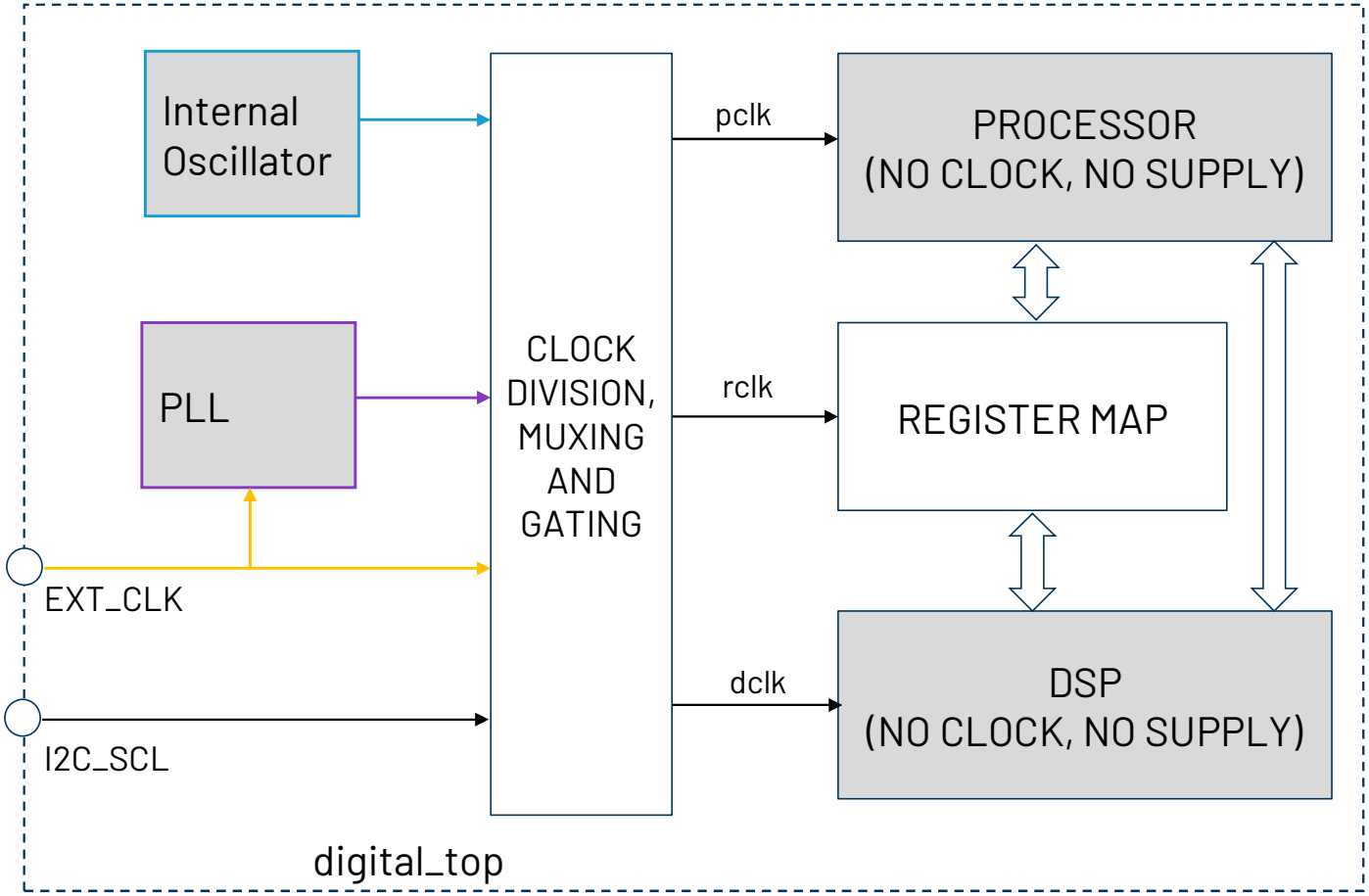


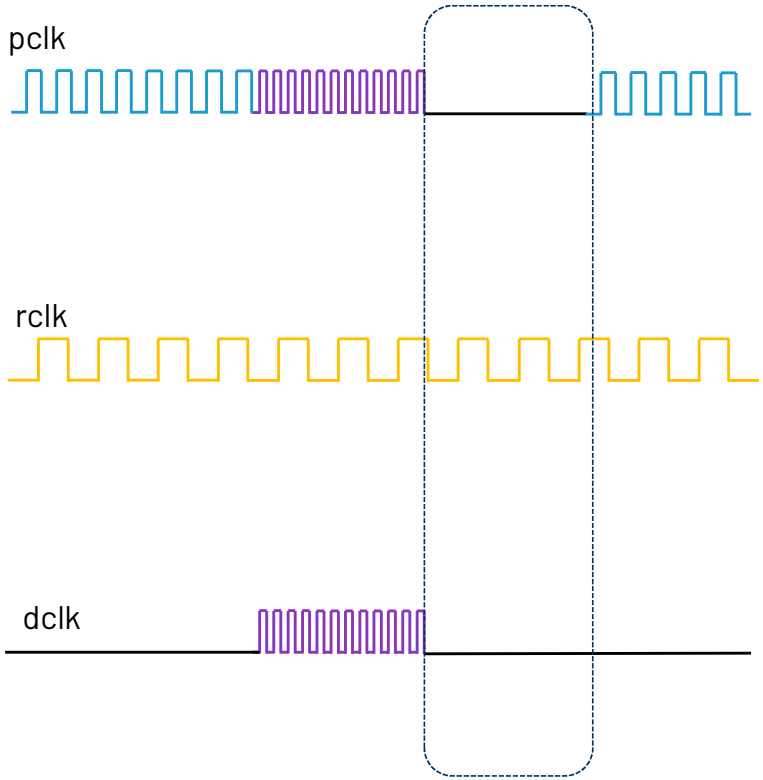
Multiple Clock Sources and Multiple Operation Modes



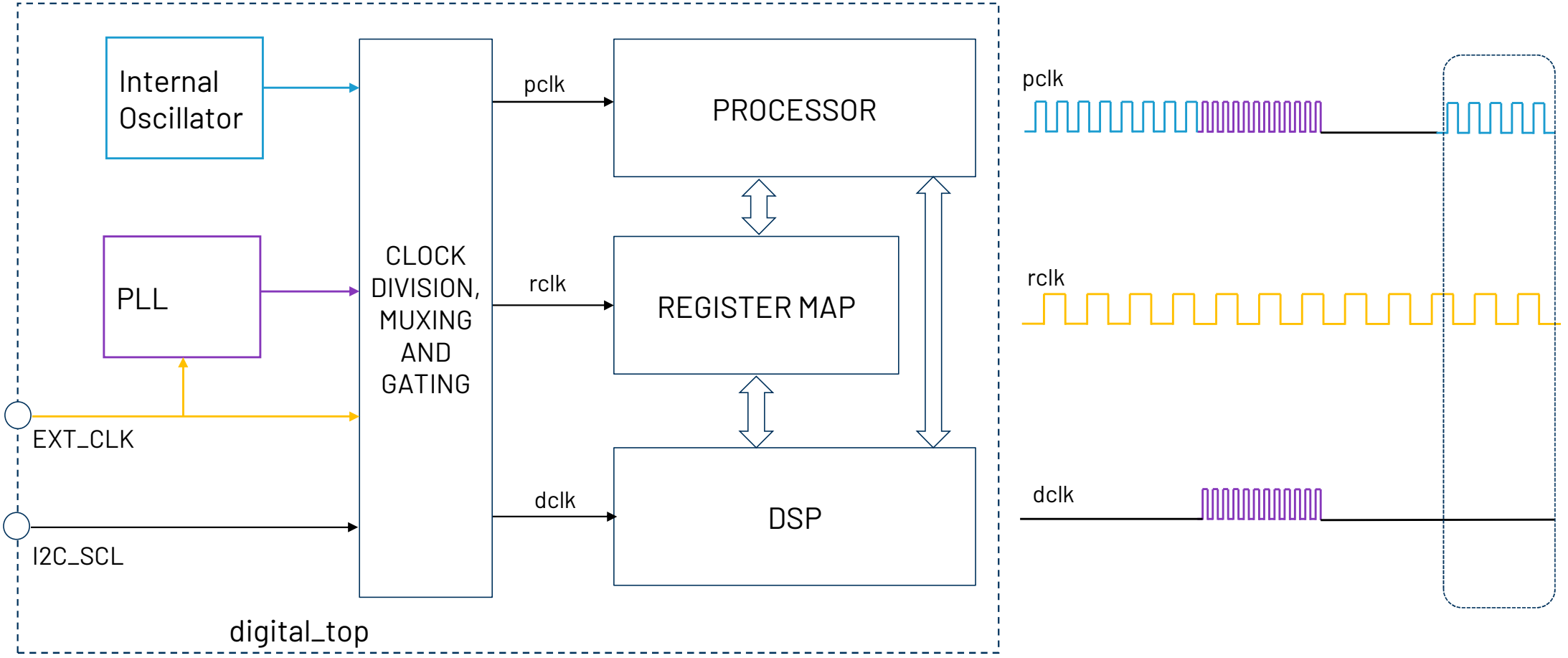


Multiple Clock Sources and Multiple Operation Modes





Multiple Clock Sources and Multiple Operation Modes



Introduction

The role of the Digital Designer in the Semiconductor Industry

Feasibility Study Stage

1 . Feasibility study. Meet specifications, estimate area and power for selected technology node (40nm, 22nm, etc)

2. Architecture definition of the digital system. Interaction with analog system

3. Micro-architecture definition of a digital block or subsystem

Design Stage

4. HDL design (e.g. SystemVerilog for synthesis)

5. Timing constraints definition (SDC file) and synthesis support (DFT Scan Insertion, etc.)

6. Power Intent deifinition (UPF file)

Sign-off Stage

7. Lint

8. Clock Domain Crossing (CDC) Sign-off. Tools: Spyglass, Litmus, etc.

9. Static Timing Analysis (STA) Sign-off. Tools: Tempus, PrimeTime, etc.

10. Power Analysis. Tools: Joules/Voltus, PimeTimePX, etc.

11. Power Intent Verification Sign-off. Tools: Conformal LP,etc

Introduction. Digital sims: RTL vs GLS

RTL simulations:

Inputs: RTL design, which is the HDL (SystemVerilog) description of the circuit

No cell delays are modelled

After running synthesis/PnR, a netlist connecting digital cells from the Standard Cell Library is generated

The **digital standard cell** from standard cell libraries is the minimum abstraction level used in digital design.

Spice/Spectre transistor level simulations are time consuming, complex to run/debug.

→ Std cells: Functional behaviour is modeled with Verilog. Delay of timing arcs through std cell is in .lib file. After synthesis/PnR, delay for all signal transitions taking into account the loads and routing R/C parasitics is annotated in SDF file (Standatd Delay Format)

GLS simulations:

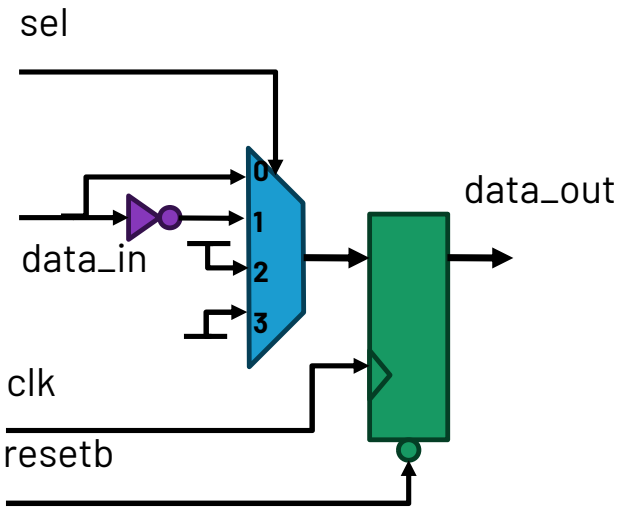
Inputs: PNR output netlist connecting standard cells

std cell library functional models in Verilog (OR gate function, AND gate, Flip-flop, etc)

SDF file, from timing extracting of PNR

Synthesis example

schematic



RTL input

```
 1 module m1 (
 2   input wire        clk,
 3   input wire        resetb,
 4
 5   input wire [1:0] sel,
 6   input wire [7:0] data_in,
 7
 8   output reg [7:0] data_out
 9 );
10
11 reg [7:0] data_out_nx;
12
13 always @(*)
14 begin
15
16   case ( sel )
17     2'd0: data_out_nx = data_in;
18     2'd1: data_out_nx = ~data_in;
19     2'd2: data_out_nx = 8'hFF;
20     default:
21           data_out_nx = 8'h00;
22   endcase
23
24 end
25
26 always @( posedge clk or negedge resetb )
27 begin
28
29   if ( !resetb )
30   begin
31     data_out <= 8'h00;
32   end
33   else
34   begin
35     data_out <= data_out_nx;
36   end
37
38 end
39
40 endmodule
41
```

gates netlist output

```
 2 // Generated by Cadence Genus(TM) Synthesis Solution 18.11-s009_1
 3 // Generated on: Feb 24 2020 10:57:46 CET (Feb 24 2020 09:57:46 UTC)
 4
 5 // Verification Directory fv/m1
 6
 7 module m1(clk, resetb, sel, data_in, data_out);
 8   input clk, resetb;
 9   input [1:0] sel;
10   input [7:0] data_in;
11   output [7:0] data_out;
12   wire clk, resetb;
13   wire [1:0] sel;
14   wire [7:0] data_in;
15   wire [7:0] data_out;
16   wire n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7;
17   wire n_8, n_9, n_10, n_11, n_12, n_13, n_14, n_15;
18   wire n_16, n_17;
19   DFFRQX2M data_out_reg_3_(.RN (resetb), .CK (clk), .D (n_17), .Q
20         (data_out[3]));
21   DFFRQX2M data_out_reg_5_(.RN (resetb), .CK (clk), .D (n_8), .Q
22         (data_out[5]));
23   DFFRQX2M data_out_reg_4_(.RN (resetb), .CK (clk), .D (n_11), .Q
24         (data_out[4]));
25   DFFRQX2M data_out_reg_1_(.RN (resetb), .CK (clk), .D (n_13), .Q
26         (data_out[1]));
27   DFFRQX2M data_out_reg_0_(.RN (resetb), .CK (clk), .D (n_16), .Q
28         (data_out[0]));
29   DFFRQX2M data_out_reg_7_(.RN (resetb), .CK (clk), .D (n_12), .Q
30         (data_out[7]));
31   DFFRQX2M data_out_reg_6_(.RN (resetb), .CK (clk), .D (n_10), .Q
32         (data_out[6]));
33   DFFRQX2M data_out_reg_2_(.RN (resetb), .CK (clk), .D (n_9), .Q
34         (data_out[2]));
35   OAI221X1M p7781A442__8780(.A0 (n_0), .A1 (sel[0]), .B0 (data_in[3]),
36         .B1 (n_15), .C0 (n_14), .Y (n_17));
37   OAI221X1M p7781A__4296(.A0 (n_1), .A1 (sel[0]), .B0 (data_in[0]), .B1
38         (n_15), .C0 (n_14), .Y (n_16));
39   OAI221X1M p7781A448__3772(.A0 (n_5), .A1 (sel[0]), .B0 (data_in[1]),
40         .B1 (n_15), .C0 (n_14), .Y (n_13));
41   OAI221X1M p7781A445__1474(.A0 (n_7), .A1 (sel[0]), .B0 (data_in[7]),
42         .B1 (n_15), .C0 (n_14), .Y (n_12));
43   OAI221X1M p7781A444__4547(.A0 (n_6), .A1 (sel[0]), .B0 (data_in[4]),
44         .B1 (n_15), .C0 (n_14), .Y (n_11));
45   OAI221X1M p7781A446__9682(.A0 (n_4), .A1 (sel[0]), .B0 (data_in[6]),
46         .B1 (n_15), .C0 (n_14), .Y (n_10));
47   OAI221X1M p7781A443__2683(.A0 (n_3), .A1 (sel[0]), .B0 (data_in[2]),
48         .B1 (n_15), .C0 (n_14), .Y (n_9));
49   OAI221X1M p7781A447__1309(.A0 (n_2), .A1 (sel[0]), .B0 (data_in[5]),
50         .B1 (n_15), .C0 (n_14), .Y (n_8));
51   NAND2BXLM p214748365A__6877(.AN (sel[1]), .B (sel[0]), .Y (n_15));
52   NAND2BXLM p214748365A449__2900(.AN (sel[0]), .B (sel[1]), .Y (n_14));
53   INVX2M Fp7718A451(.A (data_in[7]), .Y (n_7));
54   INVX2M Fp7718A454(.A (data_in[4]), .Y (n_6));
55   INVX2M Fp7718A450(.A (data_in[1]), .Y (n_5));
56   INVX2M Fp7718A452(.A (data_in[6]), .Y (n_4));
57   INVX2M Fp7718A453(.A (data_in[2]), .Y (n_3));
58   INVX2M Fp7718A455(.A (data_in[5]), .Y (n_2));
59   INVX2M Fp7718A(.A (data_in[0]), .Y (n_1));
60   INVX2M Fp7718A456(.A (data_in[3]), .Y (n_0));
61 endmodule
62
63
```

Flip-flops

Mux

Inverter

Session1 – Digital Design with System Verilog

1. Introduction
2. **HDL description of digital circuits in SystemVerilog**
3. Finite State Machines in SystemVerilog
4. Basic Digital Blocks

Lab 1

RTL Description with SystemVerilog

In Verilog , signals were described as *wire* or *reg* data types, depending if they were assigned in an “assign” statement or in procedural blocks (“always” statements)

SystemVerilog supports more data types (wire, reg, logic, bit, etc) . In synthesizable design, for simplicity, **we will only use *logic***, which can be used in procedural blocks (*always*, *always_ff*, *always_comb*) and *assign* statements.

RTL Description with SystemVerilog

While in Design Verification / Testbenches we can make use of full System Verilog capabilities, in Digital Designs we will use a subset of SystemVerilog: Synthesizable SystemVerilog.
Our designs need to represent a realizable circuit.

In Digital Designs for synthesis:

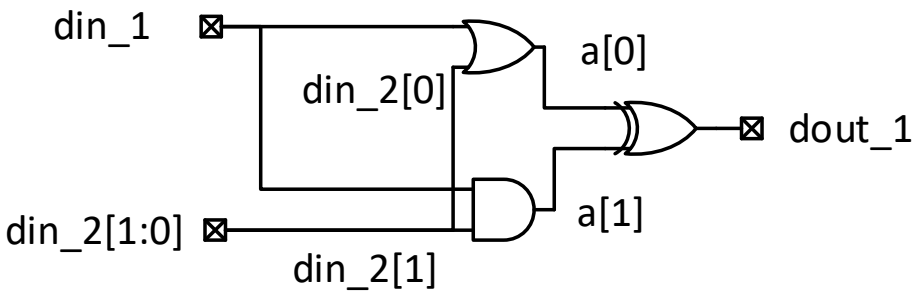
- we can't use *initial* statements
- a signal can't be assigned (receive a value) in two different procedural blocks. Only in one procedural block
- Delays (e.g. #5ns) should be avoided. They will be ignored during synthesis.
- When writing combinational logic, the output signal needs to be assigned in all *if/else* or *case* branches. Otherwise we will be inferring a latch unintentionally. A good practice is to have a default assignment before the *if* or *case.*

The key to writing RTL is

1. identify the flip-flops in the circuit that we want to implement.
2. write flip-flops using the provided template (always_ff...). Recommended: flops with asynchronous reset
3. write the combinational logic that will drive the input to those flip-flops.

Combinational Logic

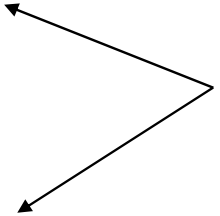
```
module example_circuit(
  input              din_1,  // Input signal, 1 bit
  input        [1:0] din_2,  // Input signal, 2 bit vector

  output logic       dout_1 // Output, 1 bit
);

logic [1:0] a;

always_comb a[0]   = din_1 || din_2[0];       // 1 bit OR
always_comb a[1]   = din_1 && din_2[1];       // 1 bit AND
always_comb dout_1 = a[0] ^ a[1];             // 1 bit XOR

endmodule
```



Combinational Logic (II)

Note *always_comb* supports multiple assignments within begin end. Equivalent to “always” continuous assignment in Verilog

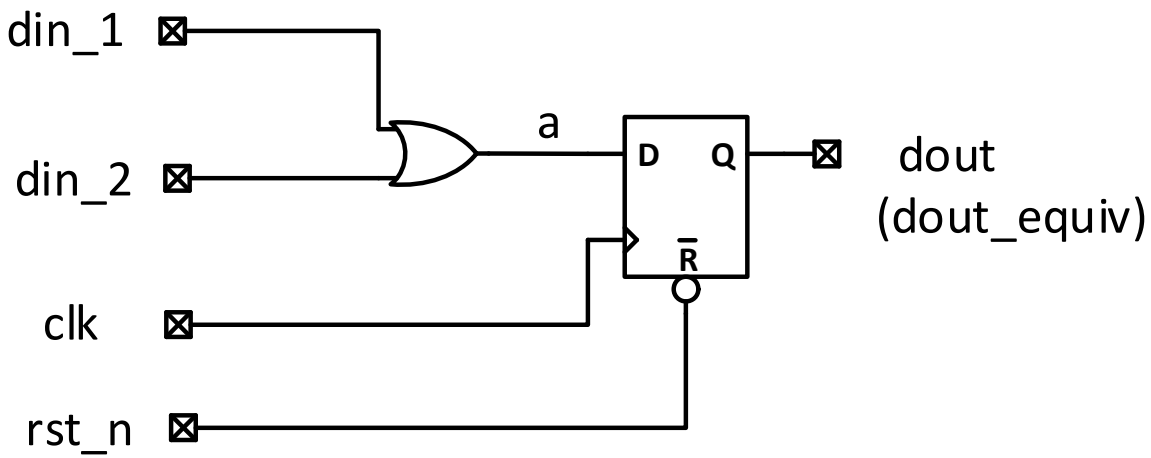
```
module example_circuit(
  input              din_1,  // Input signal, 1 bit
  input        [1:0] din_2,  // Input signal, 2 bit vector

  output logic       dout_1, // Output, 1 bit
  output logic       dout_2 // Output, 1 bit

);

logic [1:0] a;
logic [1:0] b;


always_comb  a[0]   = din_1 || din_2[0]; // 1 bit OR
always_comb  a[1]   = din_1 && din_2[1]; // 1 bit AND
always_comb  dout_2 = a[0] ^ a[1];       // 1 bit XOR


always_comb begin
  b[0]   = din_1 || din_2[0];       // 1 bit OR
  b[1]   = din_1 && din_2[1];       // 1 bit AND
  dout_2 = b[0] ^ b[1];             // 1 bit XOR
end

endmodule
```



Equivalent logic

Sequential logic. Defining a Flip-Flop

```
module example_circuit(
  input              clk,    // Clock signal
  input              rst_n,  // Asynchronous reset. Active low

  input              din_1,
  input              din_2,

  output logic       dout,
  output logic       dout_equiv
);

logic   a;


always_comb a  = din_1 || din_2;

always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    dout <= 1'b0;
  else
    dout <= a;


// Equivalent logic. Apply input comb logic to ff in always_ff statement

always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    dout_equiv <= 1'b0;
  else
    dout_equiv <= din_1 || din_2;

endmodule
```



Note that when defining the output of a FF we use the non-blocking statement (<=) , while for the output of combinational logic we use the blocking statement (=).

Good practices: Avoid using clocks or resets as data

Clock signal : drives CK pin of registers

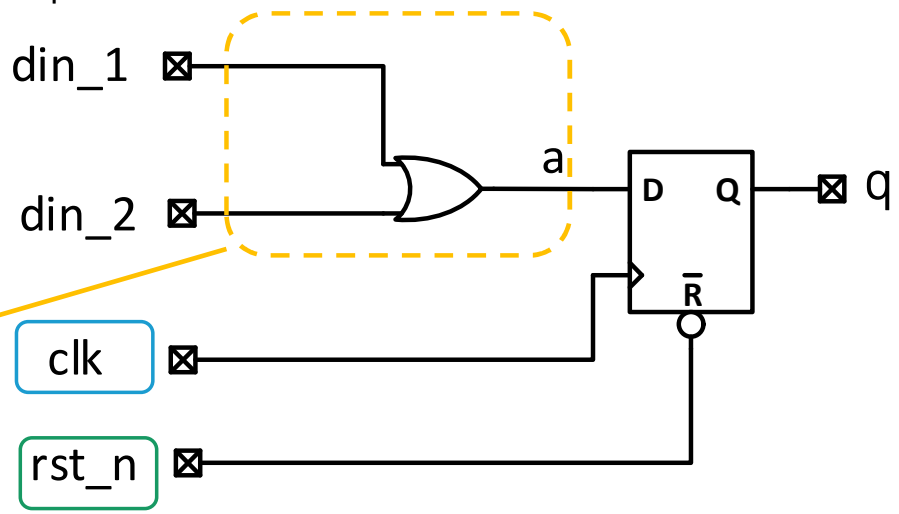
Data signal: drives synchronous pins, like D pin of a flop, D pin of a latch, EN pin of a ICG cell, output ports

Reset signal: drives Asynchronous Reset/Set pins of a flop.

Flip-flop template in Verilog:

```
always_ff @(posedge clk or negedge rst_n)
    if(!rst_n)
        q <= 1'b0;
    else begin
        ...
        q <= ...
    end
```

Combo logic cone to D input:
No clock / reset signals
should be used here!



Otherwise, extra work in SDC constraints, Lint waivers,
CDC sign-off constraints/waivers would be required

Barrel shifter

We can shift a vector by n bits using the logic shift operator

- data << n fill LSB bits with 0's
- data >> n fill MSB bits with 0's

Or the arithmetic shift operator, when the data is signed and we want to implement sign extension

- data <<< n (is equivalent to data << n)
- data >>> n fill MSB bit with sign bit of data

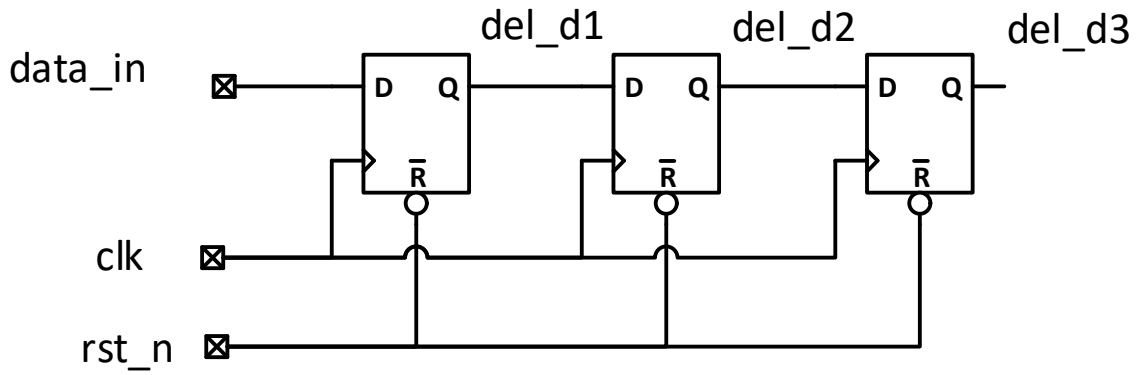
```
////////////////////////////////////////////////////////////////////////////////
// Barrel shifter.  Logic Shift
////////////////////////////////////////////////////////////////////////////////

logic [15:0] sample_in;
logic [15:0] sample_out;
logic  [2:0] n_shift;

always_comb sample_out = sample_in >> n_shift; // Logic shift: MSB bits filled with 0
```

```
////////////////////////////////////////////////////////////////////////////////
// Barrel shifter. Arithmetic Shift
////////////////////////////////////////////////////////////////////////////////

logic signed [15:0] sample_in;    // Signed, 2's complement
logic signed [15:0] sample_out;   // Signed, 2's complement
logic         [2:0] n_shift;

always_comb sample_out = sample_in >>> n_shift; // Arithmetic shift: MSB bits filled with sample_in[15]  (sign bit)
```

Shift Register

Shift register: delay a signal by a number of cycles

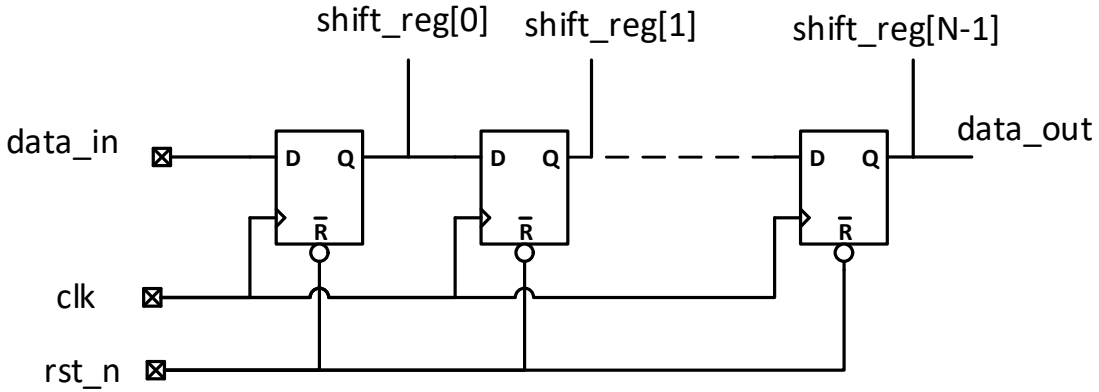
```
logic data_in;
logic del_d1;
logic del_d2;
logic del_d3;


always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n) begin
    del_d1 <= 1'b0;
    del_d2 <= 1'b0;
    del_d3 <= 1'b0;
  end
  else begin
    del_d1 <= data_in;
    del_d2 <= del_d1;
    del_d3 <= del_d2;
  end
```



Shift Register (II)

Shift register: delay a signal by N cycles. When parameter is local to a module use *localparam*.
A shift register can also be used to parallelize incoming serial data. Example: capture data from SPI miso pin.

```
module shift_register(
  input               clk,
  input               rst_n,

  input               data_in,
  output logic        data_out
);


localparam N = 8;

logic [N-1:0] shift_reg;

always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    shift_reg <= 'b0;
  else
    shift_reg <= {shift_reg[N-2:0],data_in};

always_comb data_out = shift_reg[N-1];

endmodule
```



Parameterized counter with enable, load and clear

In this case the parameter can be changed when instantiating this module. Use *parameter*.

```
module counter #(
  parameter CNT_VAL_W = 3,
  parameter DEFAULT_VAL = 3'h2
)(
 input                          clk,
 input                          rst_n,
 input                          count_en,
 input                          clear,
 input                          load_en,
 input        [CNT_VAL_W-1:0] load_val,

 output logic [CNT_VAL_W-1:0] counter_val
);


always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    counter_val <= 'h0;
  else if (clear)
    counter_val <= 'h0;
  else if (load_en)
    counter_val <= load_val;
  else if (count_en && (counter_val != 'h0))
    counter_val <= counter_val - 1'b1;
```

Instantiate parameterized module

Instantiate using default parameter values in counter.sv

```
logic       clk;
logic       rst_n;
logic       tb_count_en;
logic       tb_clear;
logic       tb_load_en;
logic [2:0] tb_load_val;
logic [2:0] counter_val;

// Instantiate module counter in testbench
counter u_counter(
 .clk        (clk),
 .rst_n      (rst_n),
 .count_en   (tb_count_en),
 .clear      (tb_clear),
 .load_en    (tb_load_en),
 .load_val   (tb_load_val),

 .counter_val(counter_val)    // output
);
```

Instantiate overwriting default parameter values in counter.sv

```
logic       clk;
logic       rst_n;
logic       tb_count_en;
logic       tb_clear;
logic       tb_load_en;
logic [7:0] tb_load_val;
logic [7:0] counter_val;

// Instantiate module counter in testbench
counter #(
  .CNT_VAL_W(8),
  .DEFAULT_VAL(8'h4)
) u_counter_2(
 .clk        (clk),
 .rst_n      (rst_n),
 .count_en   (tb_count_en),
 .clear      (tb_clear),
 .load_en    (tb_load_en),
 .load_val   (tb_load_val),

 .counter_val(counter_val)    // output
);
```

Multidimensional Arrays

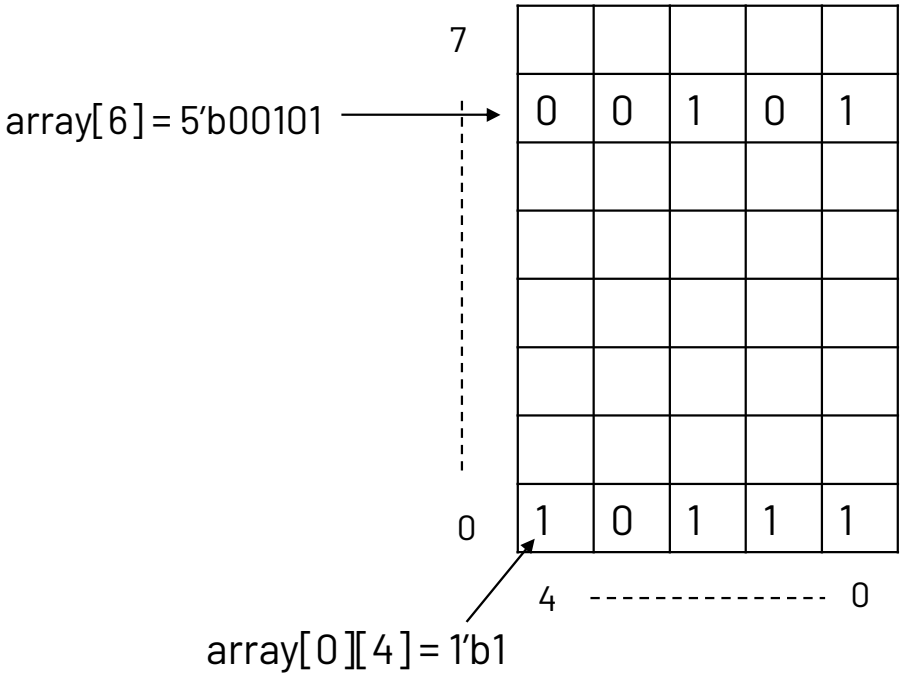
Multidimensional arrays can be described in SystemVerilog as:

logic [NUM_ROWS-1:0][NUM_COL-1] array;

Example:

logic [7:0][4:0] array;

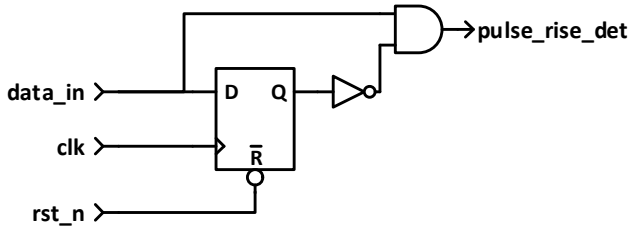
that represents 8 words of 5 bits per word



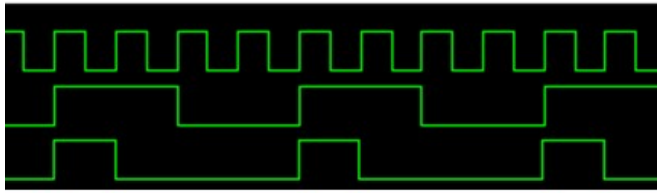
Useful circuits

Edge Detector Pulse Generator. Useful when we want to detect data transition event instead of level value.
Example: external interrupt 0->1 and keeps at 1 for multiple cycles:
we want to send one interrupt pulse to processor, not multiple interrupts

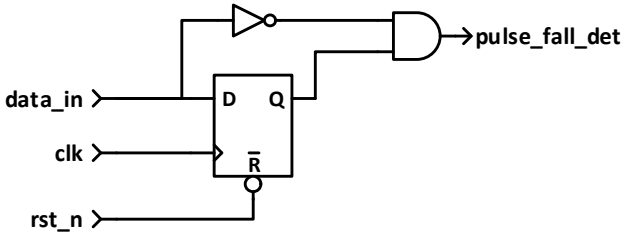
→ Rising edge detector



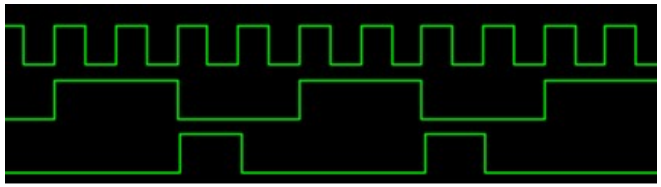
clk
data_in



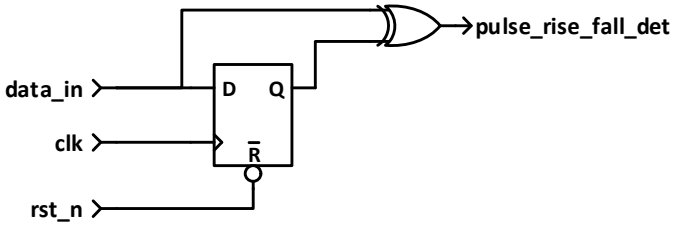
→ Falling edge detector



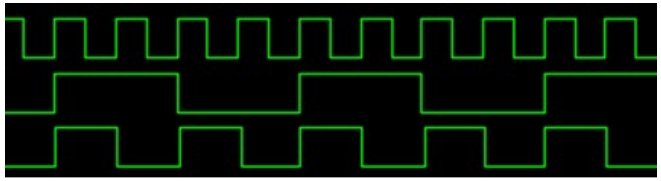
clk
data_in



→ Both Edges detector



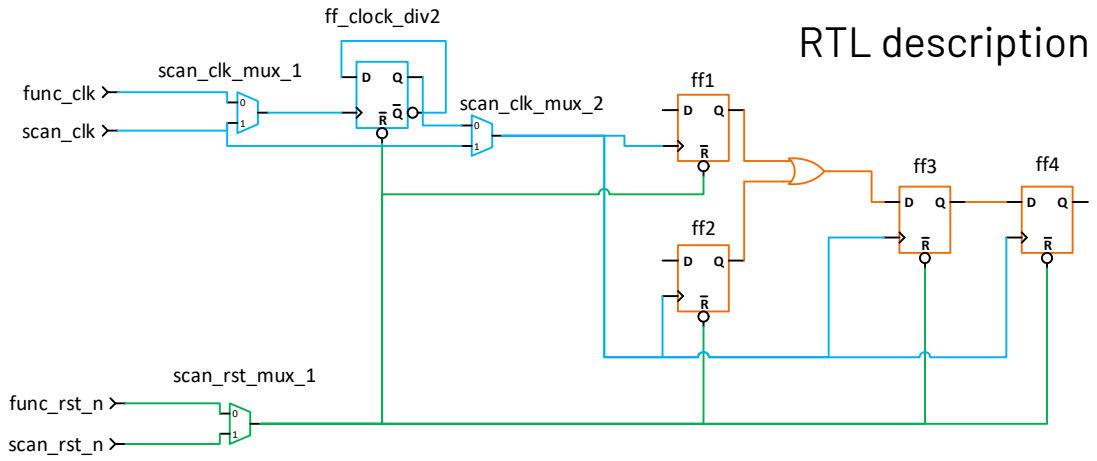
clk
data_in



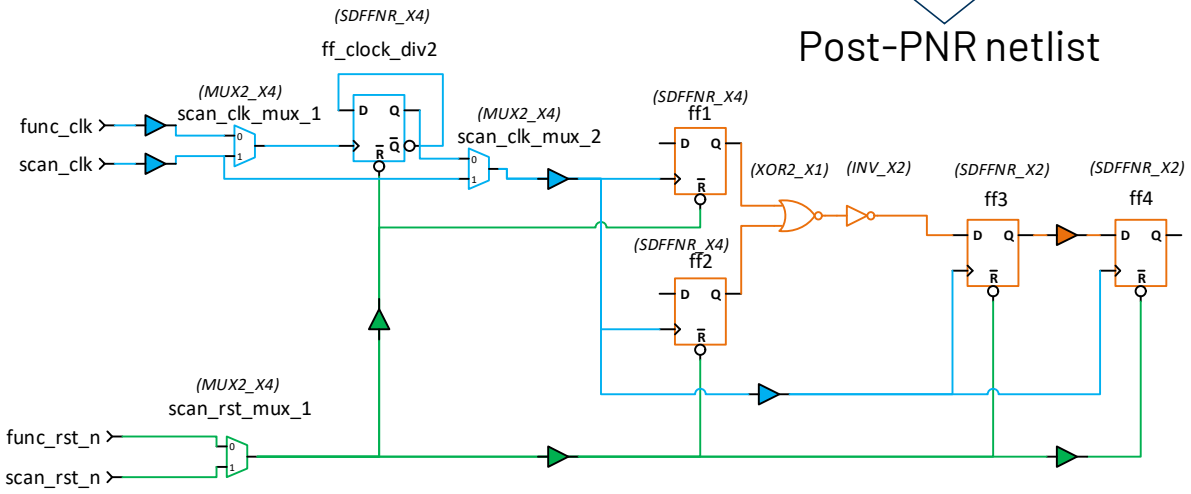
Data Path vs Reset Path vs Clock Path

In the **RTL description** of the circuit, we need to clearly differentiate
- data path elements
- clock path elements
- reset path element

so data, clock and reset signals are not combined unintentionally



The **implemented post-PNR netlist** will connect std cells that have been mapped based on the RTL description and timing constraints.
Also, extra buffers are added by synthesis and PnR tools to meet timing, max transition, max load, max fanout, etc.



Session1 – Digital Design with System Verilog

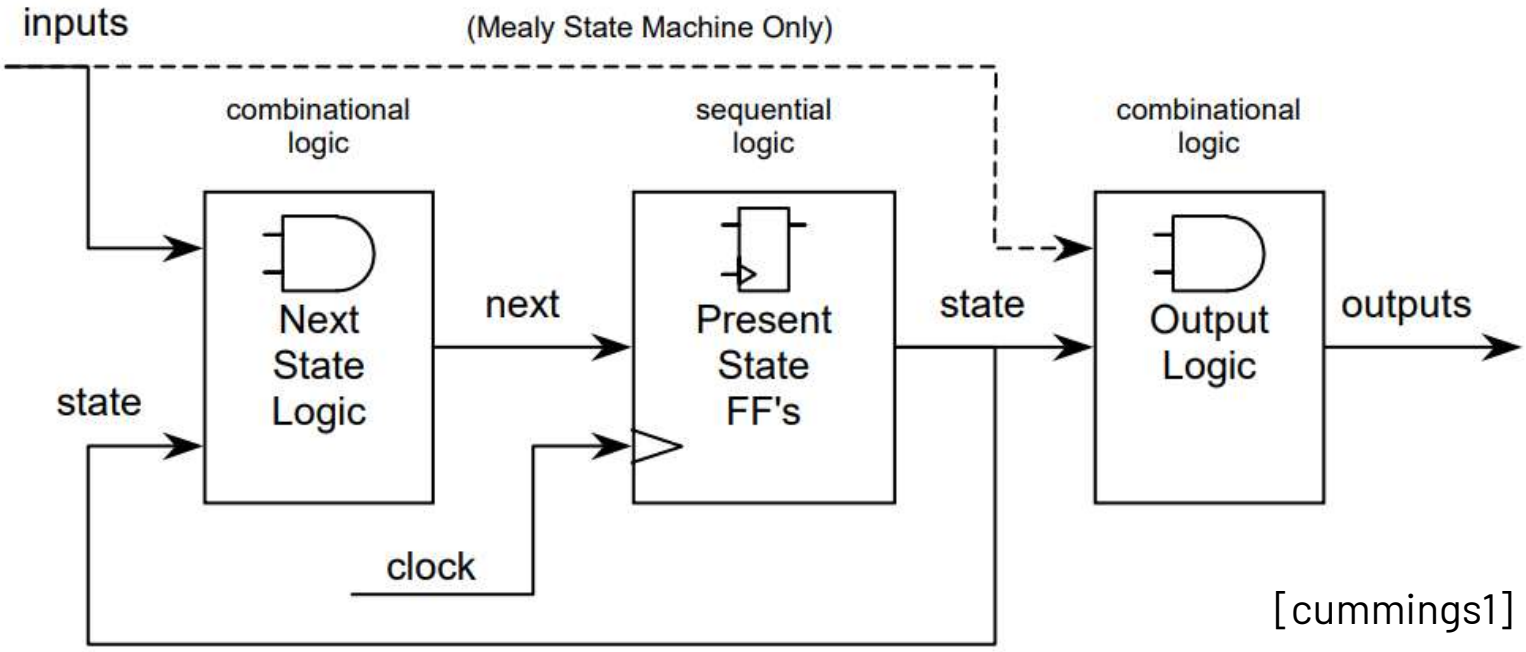
1. Introduction
2. HDL description of digital circuits in SystemVerilog
3. **Finite State Machines in SystemVerilog**
4. Common Digital Blocks

Lab 1

Finite State Machine (FSM). Classification

FSMs can be classified in:

- Moore State Machine: the outputs are only a function of the present state
- Mealy State Machine: the outputs are a function of the preset state and the input

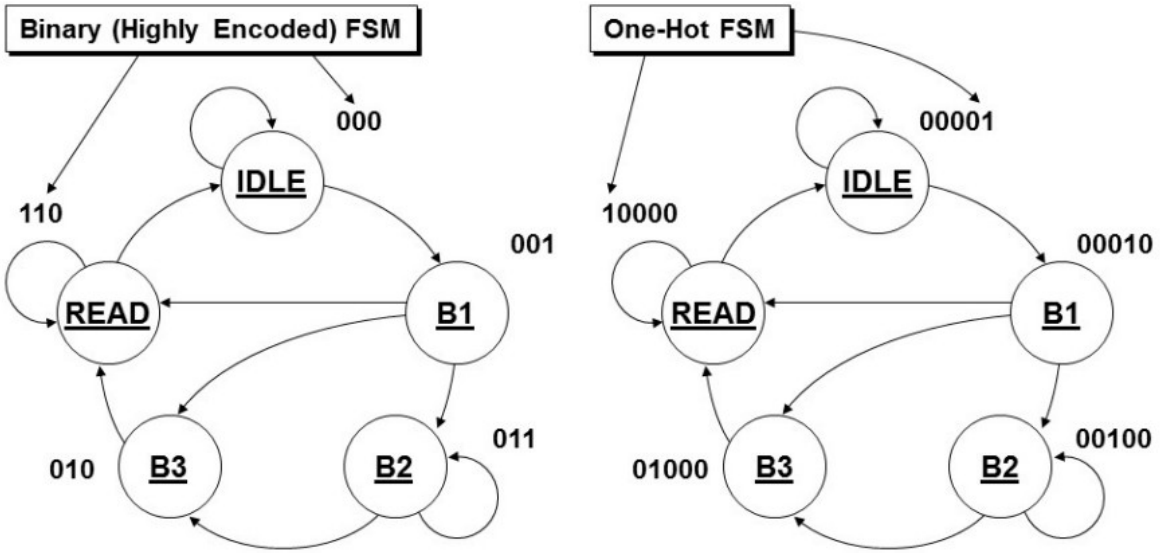


[cummings1]

Finite State Machine (FSM). State Encoding

The states can also be encoded in different ways:

- Binary encoding (default)
- Gray encoding (only one bit changes from current state to next state)
- One-Hot encoding (only one bit set to 1)
- Custom encoding. Example: in simple Moore FSMs, outputs can directly be some state bits.

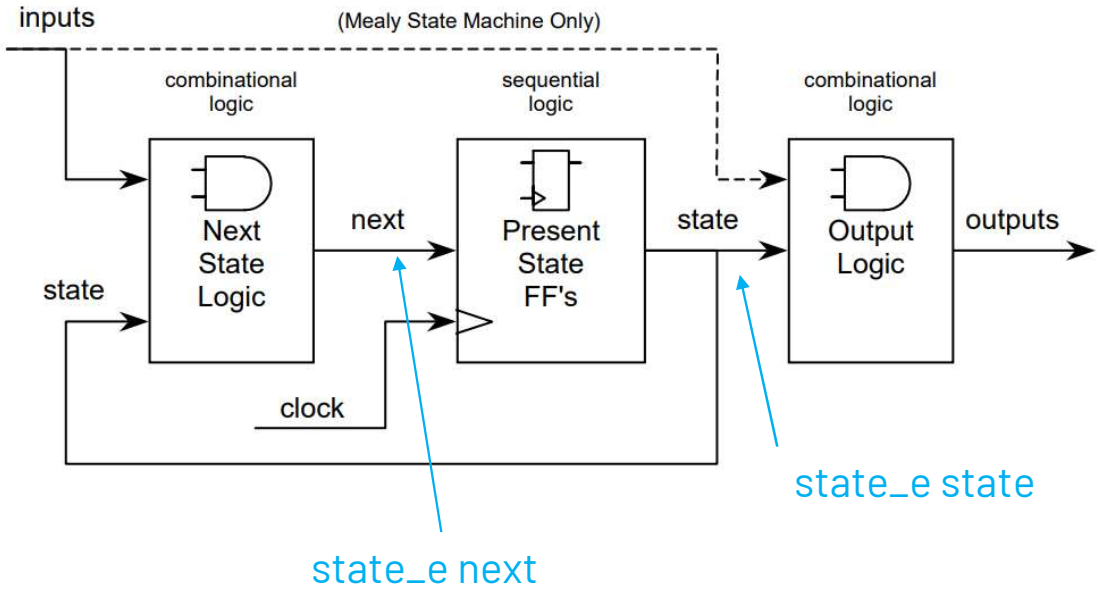


[cummings2]

Finite State Machine (FSM). State Encoding in Verilog

Abstract enumerated types (synthesis chooses state encoding)

```
typedef enum {    IDLE,
                  READ,
                  DLY,
                  DONE} state_e;
state_e state, next;
```



[cummings1]

Binary encoded enumerated types

```
typedef enum logic [1:0] {  IDLE    = 2'b00,
                            READ    = 2'b01,
                            DLY     = 2'b10,
                            DONE    = 2'b10} state_e;
state_e state, next;
```

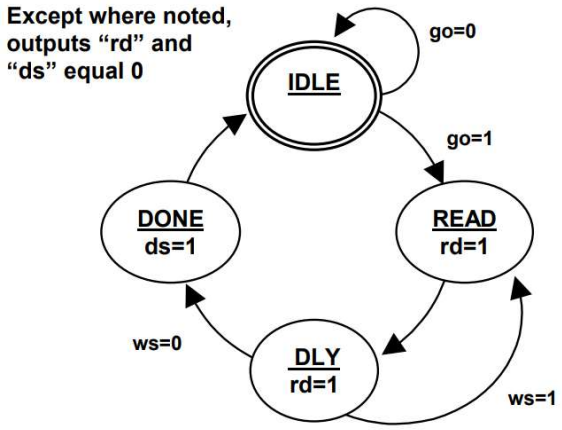
FSM Coding Styles

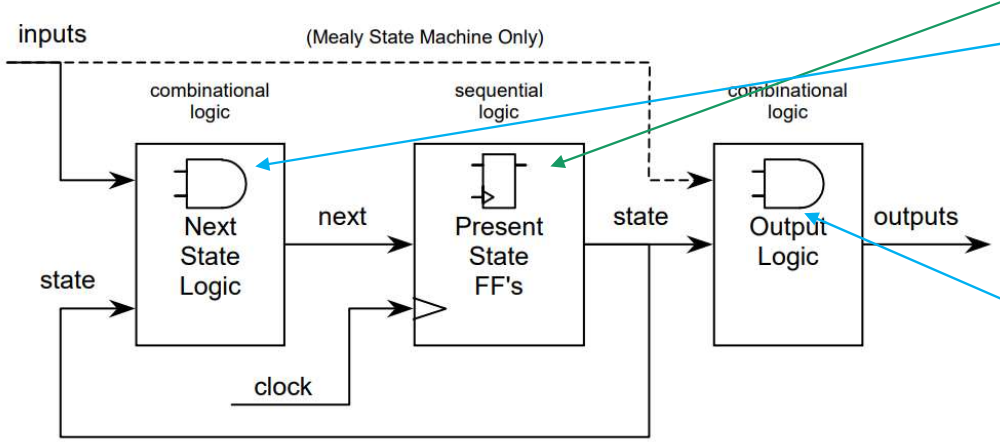
- Non-Registered Outputs
  - Separate next_state and output logic
  - Combine next_state and output logic in same always_comb

- Registered Outputs
  - 3 always blocks
  - 2 always blocks : State Encoded Outputs

FSM #1: Moore, Non-Registered Outputs

Style 1: Separate always_comb for next state logic and output logic



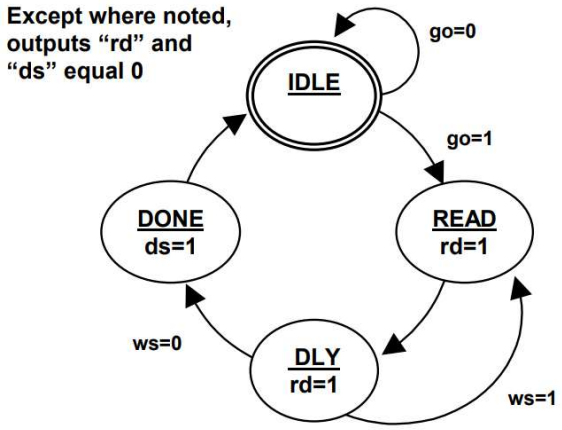


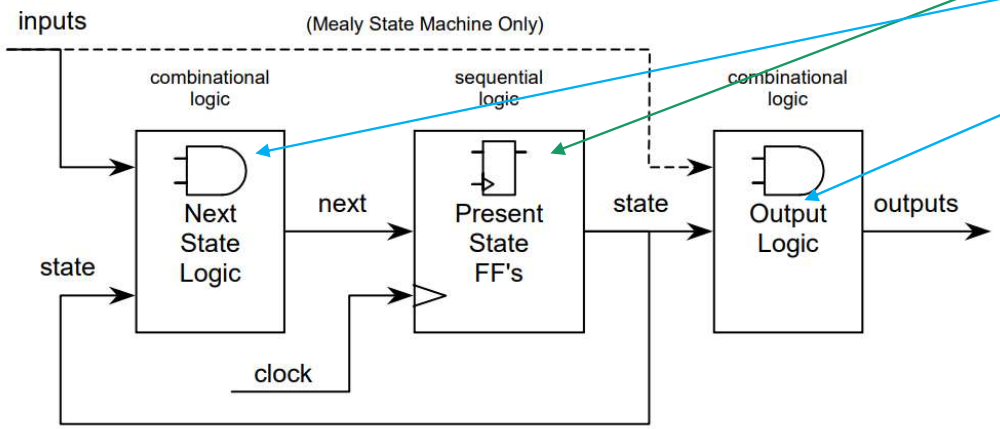

```
module fsm (
 input        clk,
 input        rst_n,
 input        go,
 input        ws,
 output logic ds,
 output logic rd
);

typedef enum logic [1:0] {
  IDLE = 2'b00,
  READ = 2'b01,
  DLY  = 2'b10,
  DONE = 2'b11} state_e;

state_e state, next;

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n)   state <= IDLE;
  else          state <= next;

always_comb begin
  next = IDLE;
  case (state)
    IDLE: if (go) next = READ;
          else    next = IDLE;

    READ: next = DLY;

    DLY: if (ws) next = READ;
         else    next = DONE;

    DONE: next = IDLE;

    default: next = IDLE;
  endcase
end

always_comb rd = (state==READ || state==DLY);
always_comb ds = (state==DONE);
```

[cummings1]

FSM #2: Moore, Non-Registered Outputs

Style 2: Combined next state logic and output logic in same always_comb



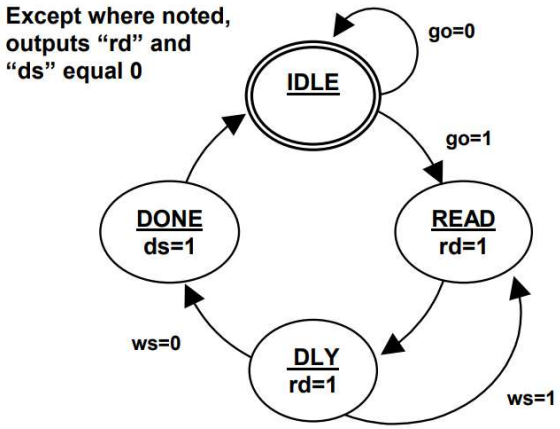


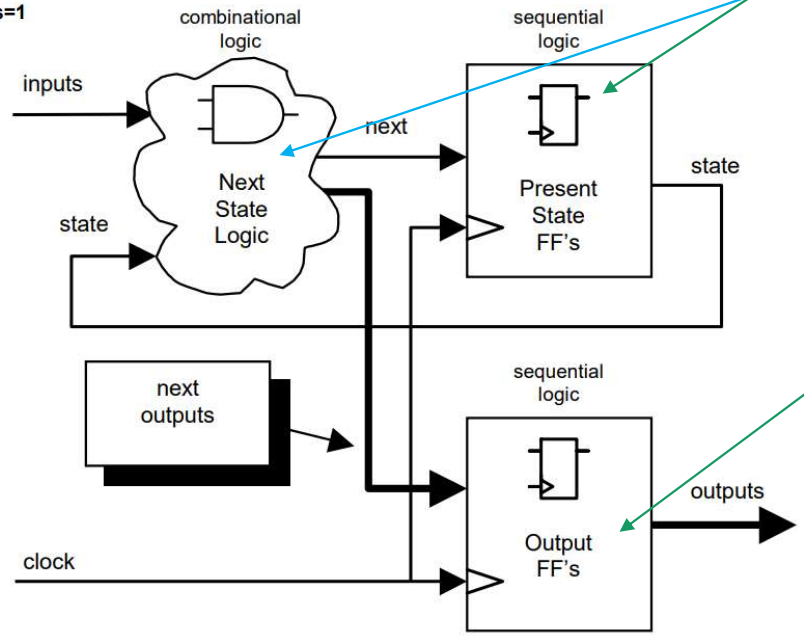

```
module fsm (
 input        clk,
 input        rst_n,
 input        go,
 input        ws,
 output logic ds,
 output logic rd
);

typedef enum logic [1:0] {
  IDLE = 2'b00,
  READ = 2'b01,
  DLY  = 2'b10,
  DONE = 2'b11} state_e;

state_e state, next;

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n)   state <= IDLE;
  else          state <= next;

always_comb begin
  next = IDLE;
  ds   = 1'b0;
  rd   = 1'b0;
  case (state)
    IDLE: if (go) next = READ;
          else    next = IDLE;

    READ: begin rd   = 1'b1;
                next = DLY;
          end

    DLY:  begin rd = 1'b1;
                if (ws) next = READ;
                else    next = DONE;
          end

    DONE: begin ds   = 1'b1;
                next = IDLE;
          end
    default: next = IDLE;
  endcase
end

endmodule
```

[cummings1]

FSM #3: Moore, Registered Outputs

Style 3: Two separated always_ff for state and output registers



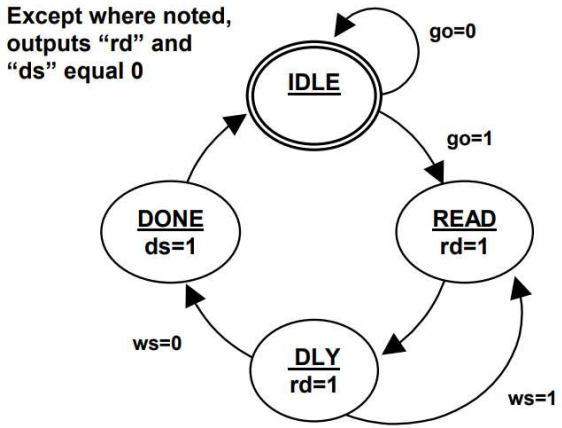


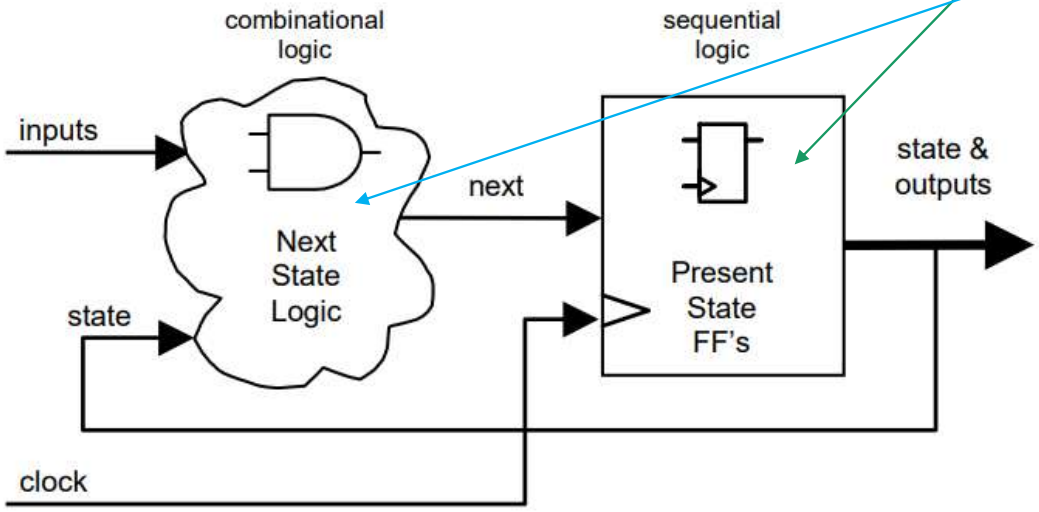
[cummings1]

```
module fsm (
 input        clk,
 input        rst_n,
 input        go,
 input        ws,
 output logic ds,
 output logic rd);

typedef enum logic [1:0] {
  IDLE = 2'b00,
  READ = 2'b01,
  DLY  = 2'b10,
  DONE = 2'b10} state_e;
state_e state, next;

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n)   state <= IDLE;
  else          state <= next;

always_comb begin
  next = IDLE;
  case (state)
    IDLE: if (go) next = READ;
          else    next = IDLE;
    READ: next = DLY;
    DLY: if (ws) next = READ;
         else    next = DONE;
    DONE: next = IDLE;
    default: next = IDLE;
  endcase
end

always @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    ds <= 1'b0;
    rd <= 1'b0;
  end
  else begin
    ds <= 1'b0;
    rd <= 1'b0;
    case (state)
      IDLE: if (go) rd <= 1'b1;
      READ: rd <= 1'b1;
      DLY:  if (ws) rd <= 1'b1;
            else    ds <= 1'b1;
      default: begin
                 ds <= 1'b0;
                 rd <= 1'b0;
               end
    endcase
  end
endmodule
```

FSM #4: Moore, Registered Outputs

Style 4: State encoded outputs. In simple FSM, states can be encoded so particular bits take the value of an output for that state





[cummings1]

```
module fsm (
 input        clk,
 input        rst_n,
 input        go,
 input        ws,
 output logic ds,
 output logic rd);

typedef enum logic [2:0] {
 // state bits =    x_ds_rd
          IDLE = 3'b0_0_0
          READ = 3'b0_0_1,
          DLY  = 3'b1_0_1,
          DONE = 3'b0_1_0} state_e;
state_e state, next;

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) state <= IDLE;
  else        state <= next;

always_comb begin
  next = IDLE;
  case (state)
    IDLE: if (go) next = READ;
          else    next = IDLE;

    READ:         next = DLY;

    DLY:  if (ws) next = READ;
          else    next = DONE;

    DONE:         next = IDLE;

    default:      next = IDLE;
  endcase
end

always_comb {ds,rd} = state[1:0];

endmodule
```

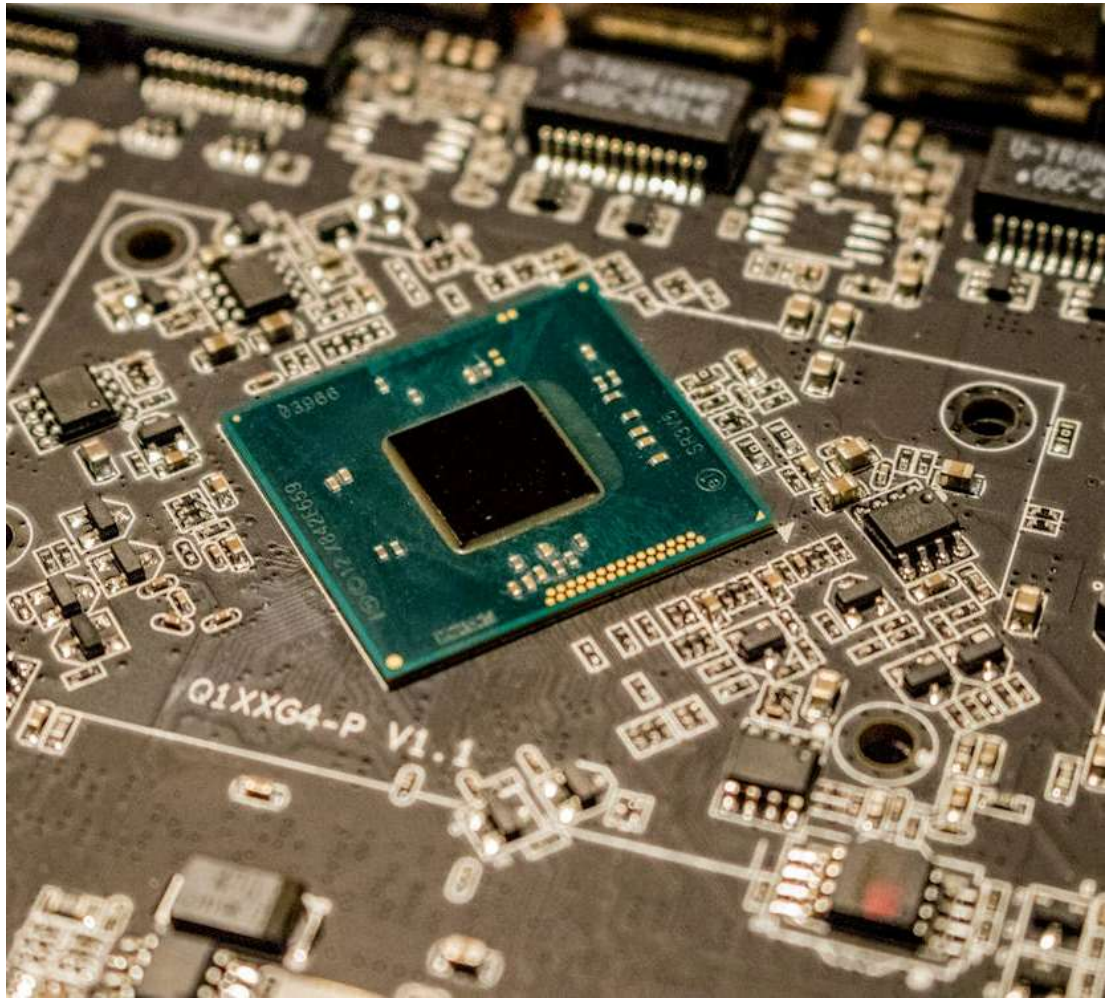
Session4 – Digital Design with System Verilog

1. Introduction
2. HDL description of digital circuits in SystemVerilog
3. Finite State Machines in SystemVerilog
4. **Common Digital Blocks**

Lab 1

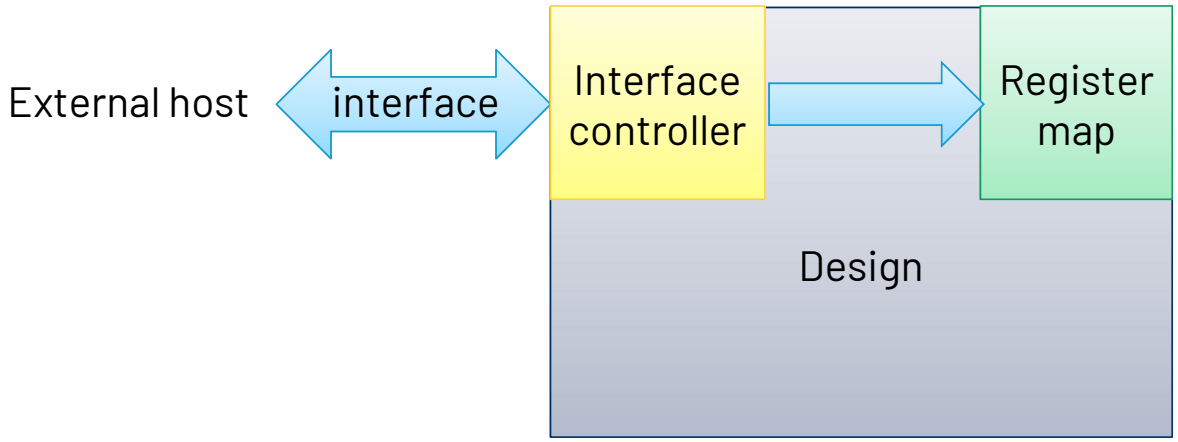
Common digital blocks

- This section introduces digital blocks commonly found in any digital IC design
  - Configuration interface
  - Register map
  - Sequencers
  - Buffers
  - Microcontroller
  - Peripherals



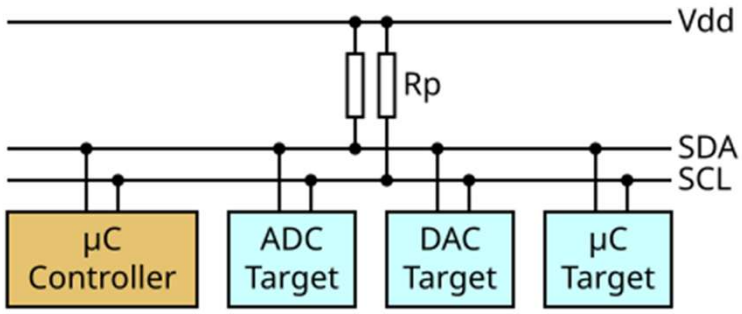
Configuration interface

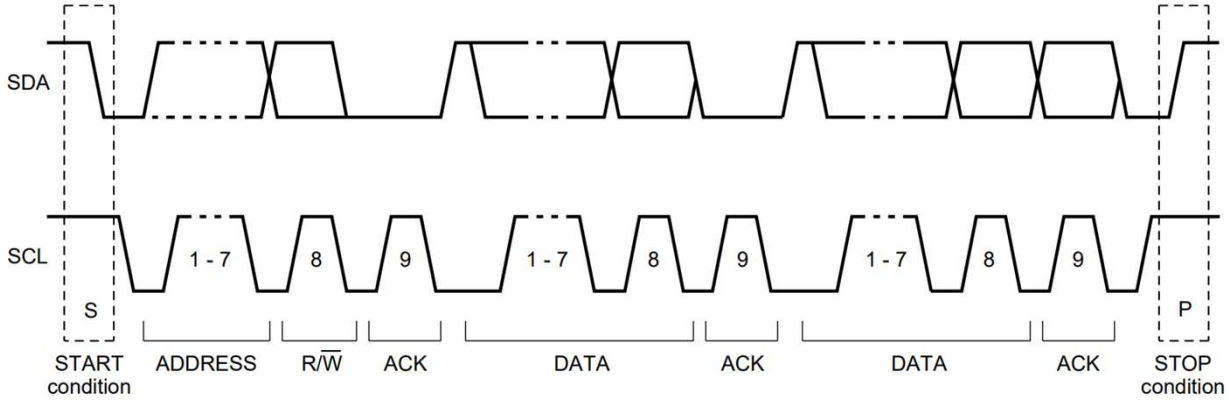
- Allows configuration of IC parameters from an external host
- Most common interfaces
  - I2C (Inter-Integrated Circuit bus)
  - SPI (Serial Peripheral Interface)
  - UART (Universal Asynchronous Receiver-Transmitter)
- Provides external access to the register map



Configuration interface: I2C

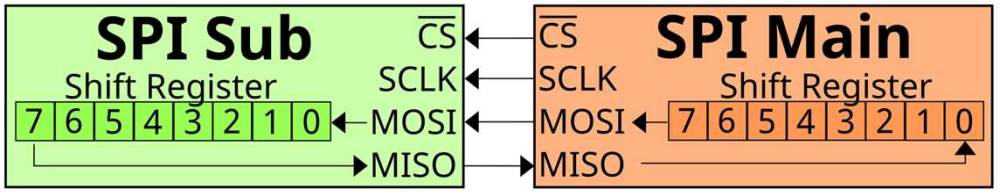
- Serial bus, supports multiple devices
- 2 lines: SCL (clock), SDA (data)
- Read/write messages
- Start/Stop condition, ACK, NACK



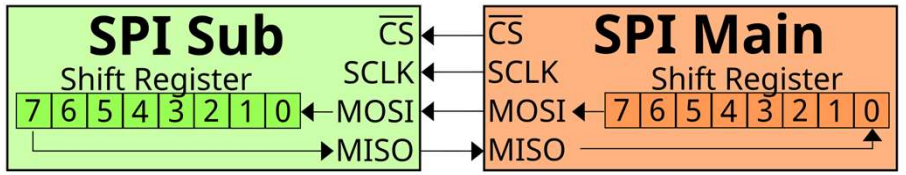


Configuration interface: SPI

- Serial interface, point to point
  - SCLK, MOSI, MISO may be shared across devices
  - CS unique per device
- 4 lines
  - SCLK: clock
  - MOSI: Master Output Slave Input
  - MISO: Master Input Slave Output
  - CS: Chip Select

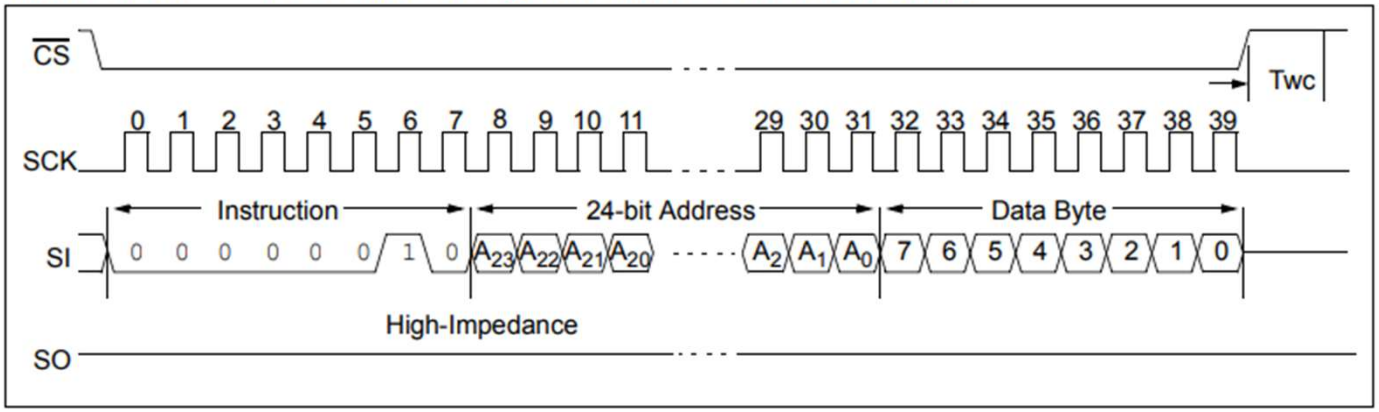


Configuration interface: SPI



- No specification for frame format (application specific)

FIGURE 2-2C: BYTE WRITE SEQUENCE WITH 24-BIT ADDRESSING



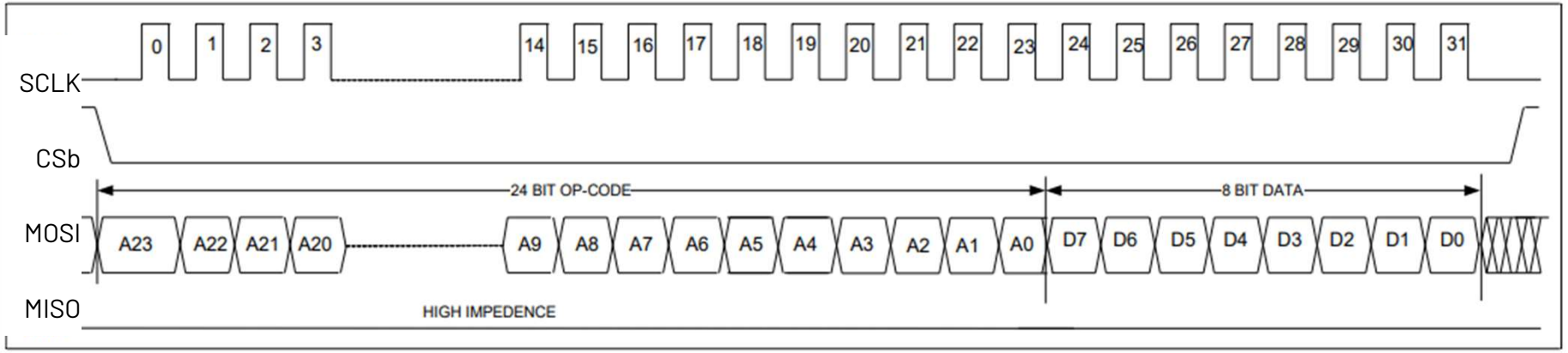


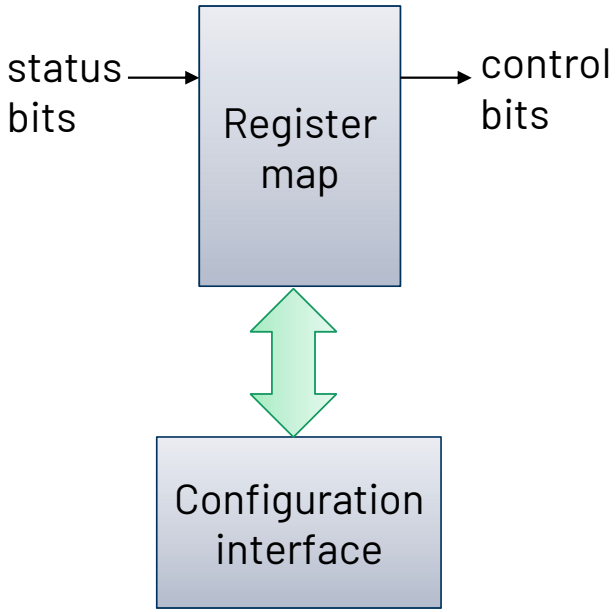
Figure 4: Single-Byte Write

Register map

- Bank of registers to control different sections of design
- Accessed externally via configuration interface
- Usually grouped in 8 or 16 bit registers
- Registers can be read-only (status bits) or read-write (control bits)

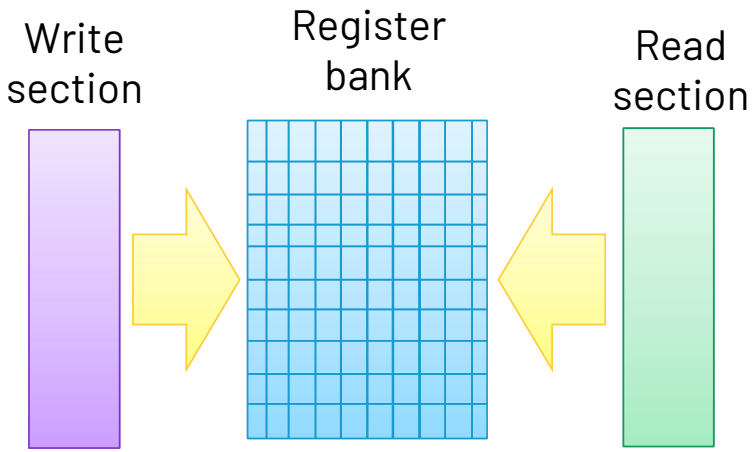
AD9364 Register Map Reference Manual

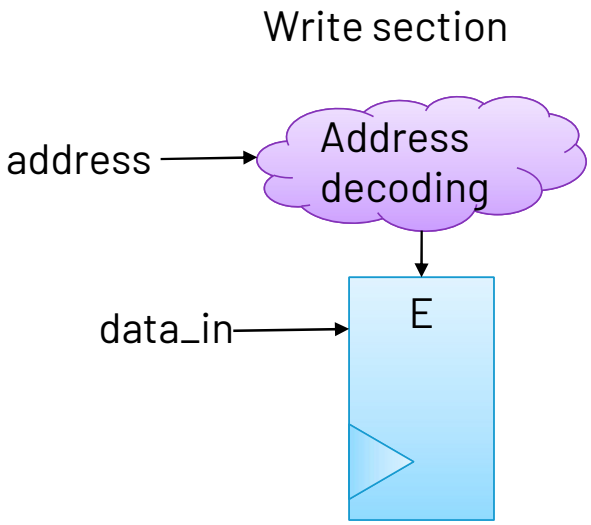
Register Address	Name	D7	D6	D5	D4	D3	D2	D1	D0	Default	R/W
0x000	SPI Configuration	Must be 0	3-Wire SPI	LSB First	Open		LSB First	3-Wire SPI	Must be 0	0x00	R/W
0x001	Multichip Sync and Tx Mon Control	Open		Tx Monitor Enable	Open	MCS RF Enable	MCS BBPLL enable	MCS Digital CLK Enable	MCS BB Enable	0x00	R/W
0x002	Tx Enable and Filter Control	Open	Tx Enable	THB3 Enable and Interp[1:0]		THB2 Enable	THB1 Enable	Tx FIR Enable and Interpolation[1:0]		0x5F	R/W
0x003	Rx Enable and Filter Control	Open	Rx Enable	RHB3 Enable and Decimation[1:0]		RHB2 Enable	RHB1 Enable	Rx FIR Enable and Decimation[1:0]		0x5F	R/W
0x004	Input Select	Must be 0	Tx Output	Rx Input [5:0]						0x00	R/W
0x005	RFPLL Dividers	Tx VCO Divider[3:0]				Rx VCO Divider[3:0]				0x00	R/W
0x006	Rx Clock and Data Delay	DATA_CLK Delay[3:0]				Rx Data Delay [3:0]				0x00	R/W
0x007	Tx Clock and Data Delay	FB_CLK Delay[3:0]				Tx Data Delay [3:0]				0x00	R/W

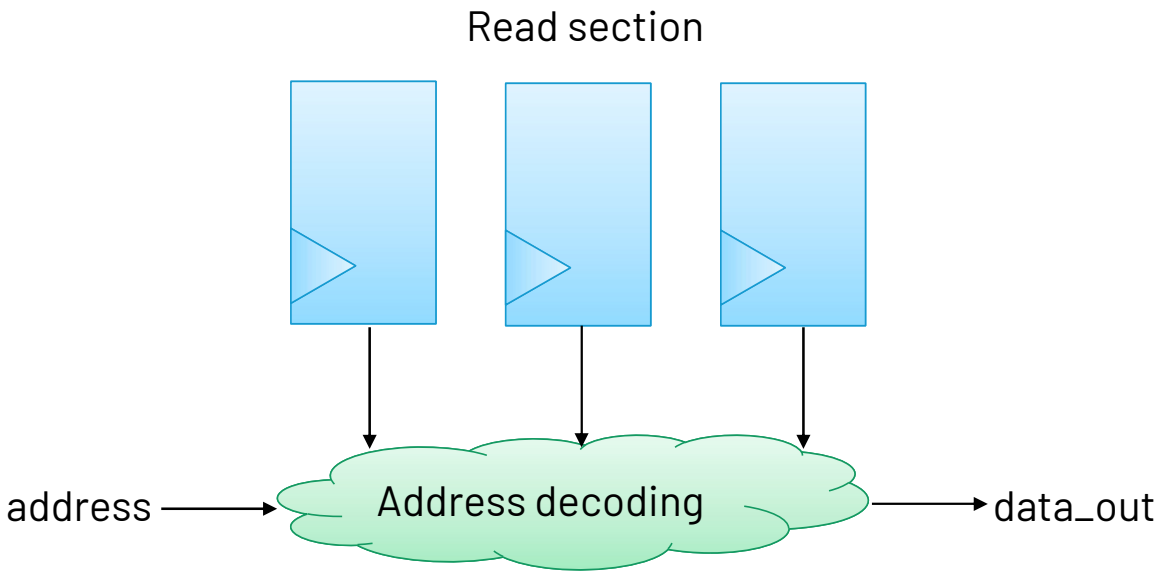


Register map

- Write section
  - Address decoding logic to enable writing input data to each register
- Read section
  - Address decoding logic to select output data from all all registers

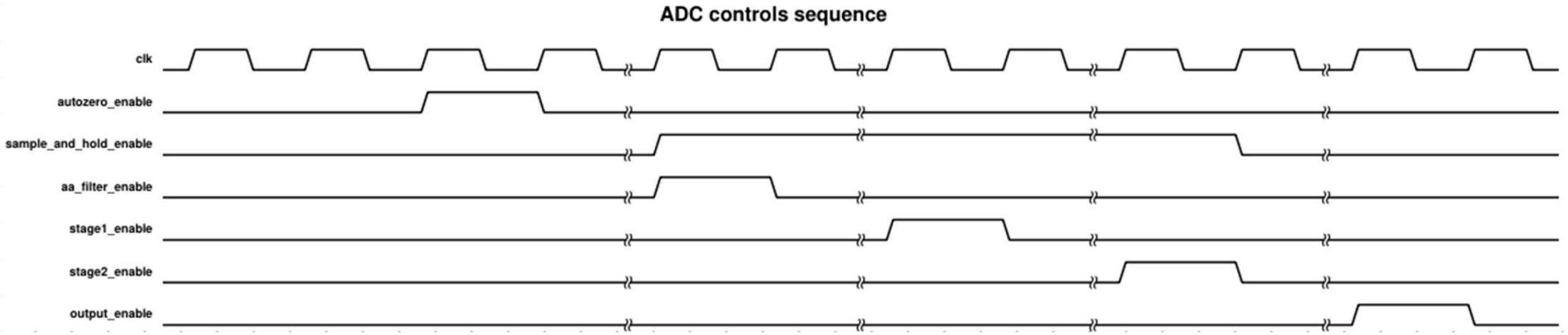






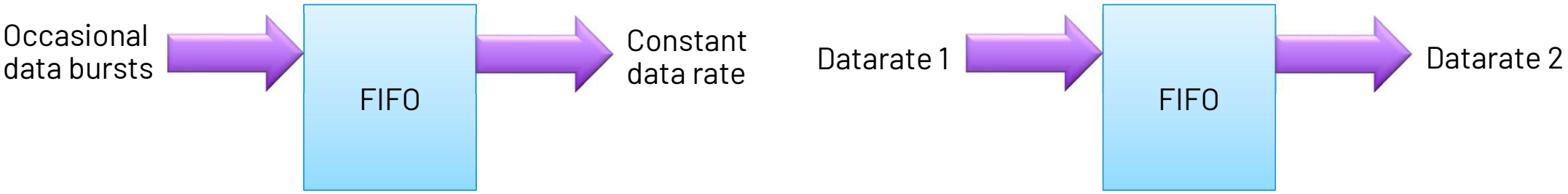
Sequencer

- Manage the sequence of operations, producing finely timed control signals
- Common examples: ADC controllers, RAM, OTP memory controllers
- Typically implemented with state machines, counters
- Usually simple design since there's no feedback path involved



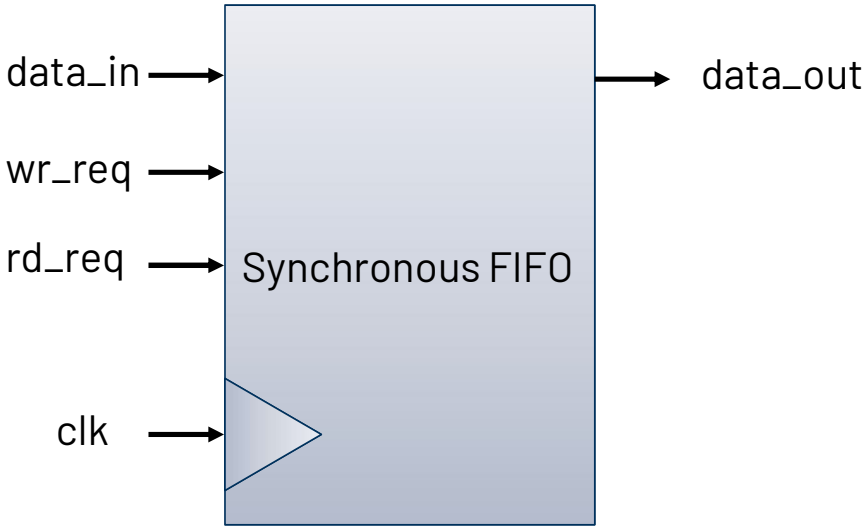
Buffers/FIFOs

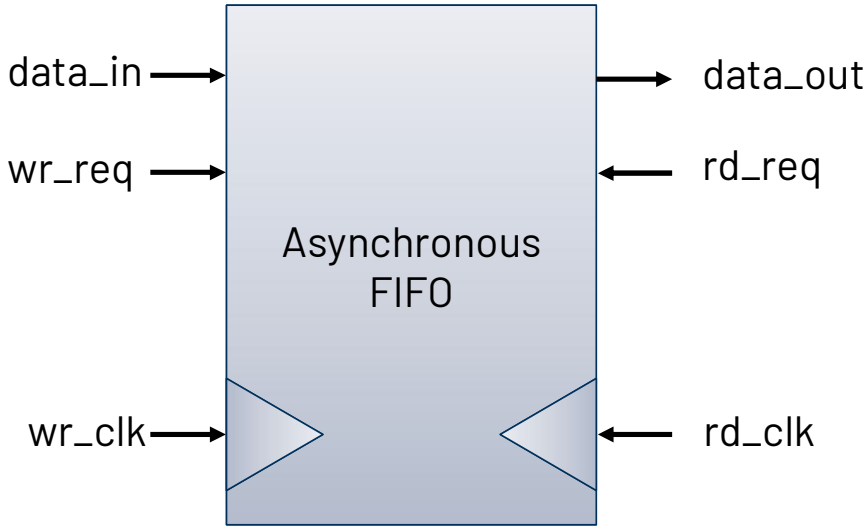
- Temporary data storage to balance different data rates
- Used in data streaming, communication interfaces, signal processing
- Example
  - IP video streaming
    - Video transmitted in irregular IP packets, must be output constantly
  - HDMI/DisplayPort audio buffering
    - Audio transmitted in video blanking regions, must be output constantly
  - ADC data windowing
    - Accelerometer data sampled in ADC written to a FIFO buffer for window processing



Buffers/FIFOs

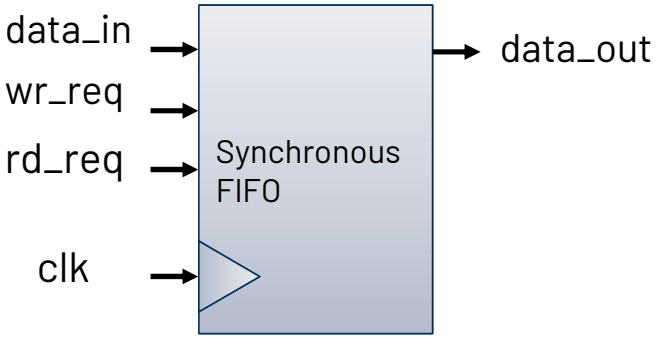
- 2 timing categories
  - Synchronous: common clock for write, read
  - Asynchronous: independent write, read clocks

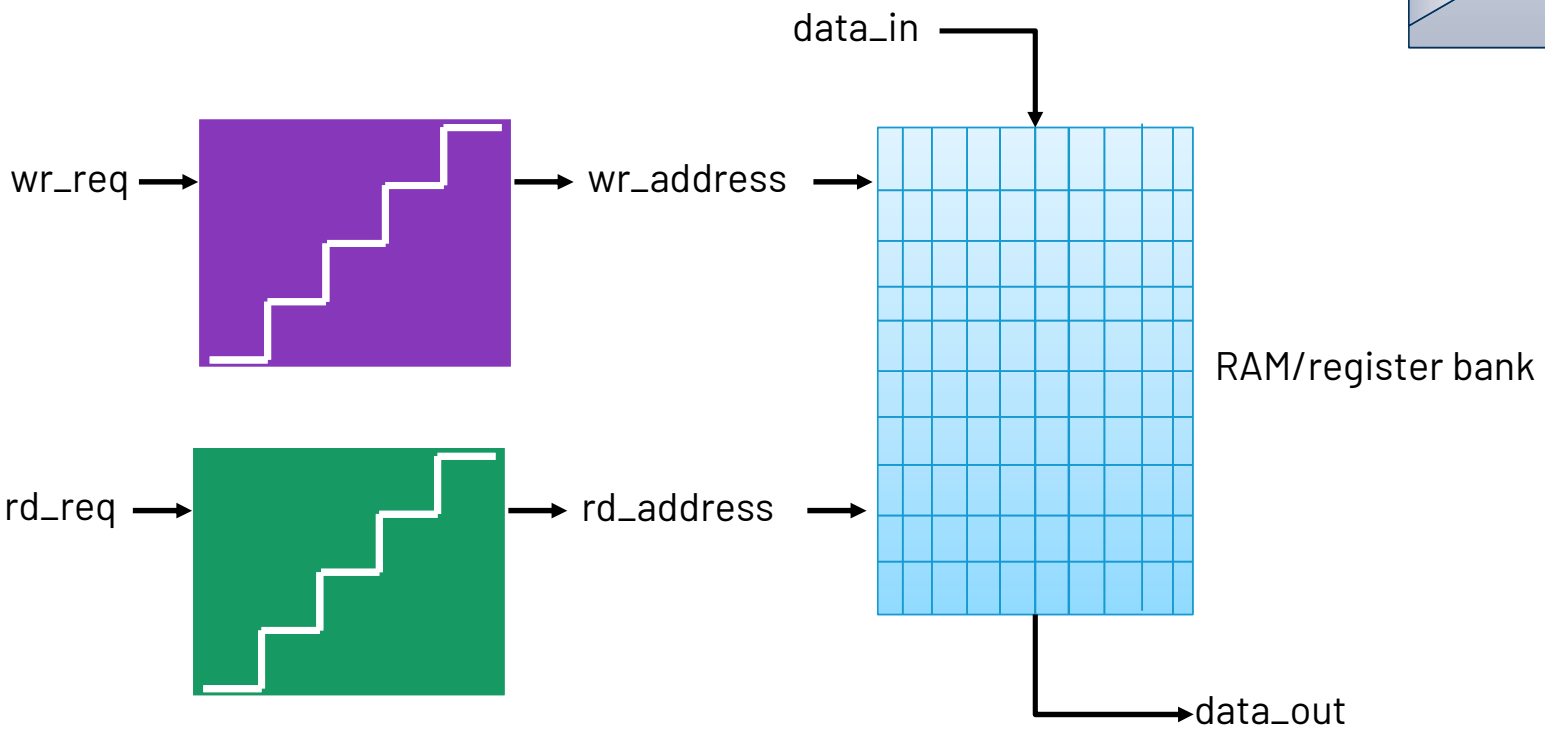




Synchronous FIFO design

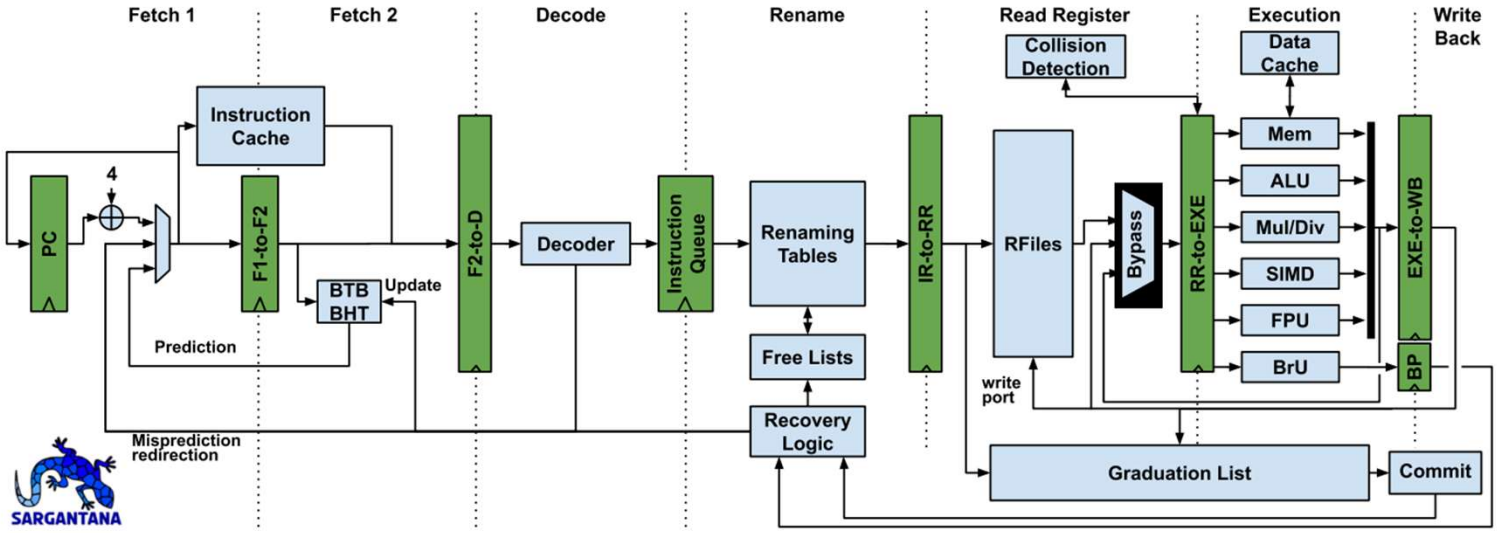
- Central block: RAM memory macro or register bank
- Separate write, read pointers: counters





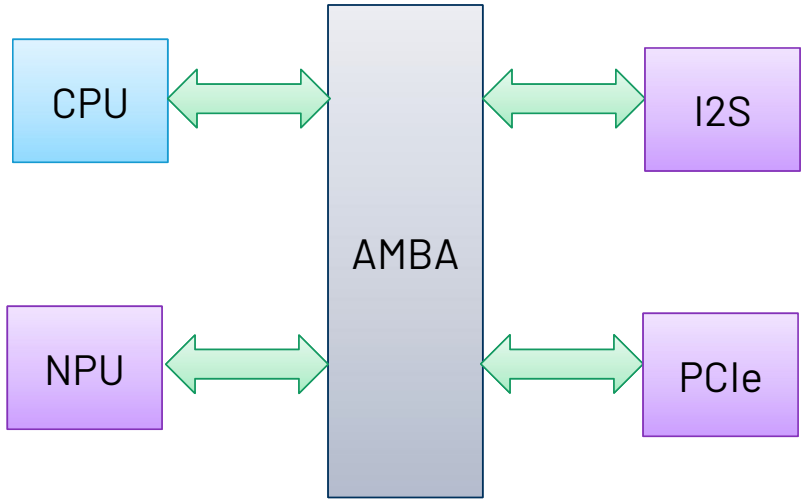
Microcontroller

- Provides control and processing capabilities
- Can run firmware to perform complex tasks and control other blocks
- Usually associated to a bus matrix (eg AMBA) to communicate with peripherals or other blocks
- Common examples: ARM Cortex, RISC-V



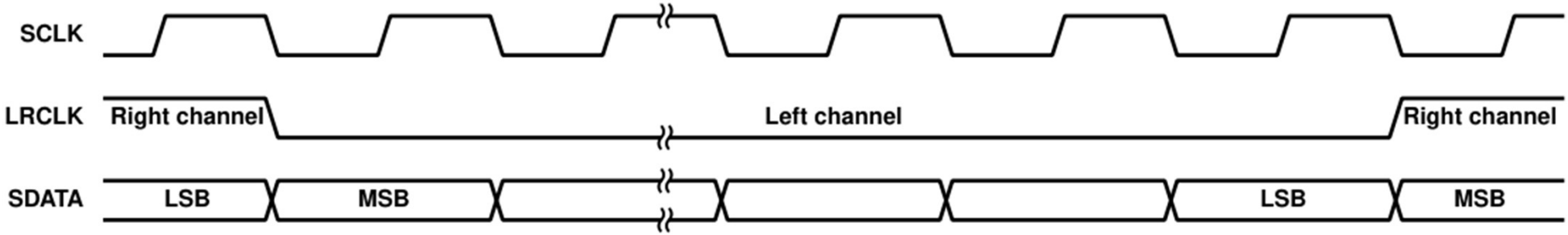
Peripherals

- Provide additional functionality and interfaces to the design
- Usually connect to microcontroller or core with bus matrix
- Common examples
  - I2S (Inter-Integrated Circuit Sound)
  - Peripheral Component Interconnect express (PCIe)
  - Neural accelerator (NPU)



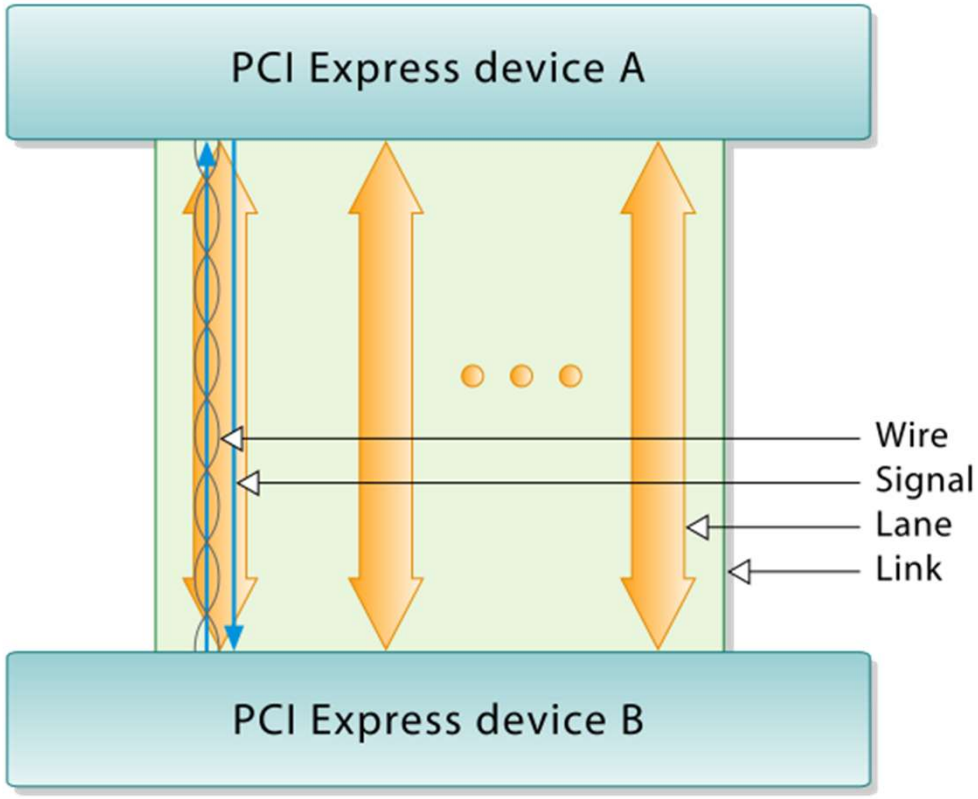
Peripherals: I2S

- I2S (Inter-Integrated Circuit Sound) provides audio serial connectivity between chips
  - SCLK serial clock
  - LRCLK left/right channel id
  - SDATA serial audio data
- Multiple stereo pairs support (add more SDATA lines, time demultiplexing in single SDATA)



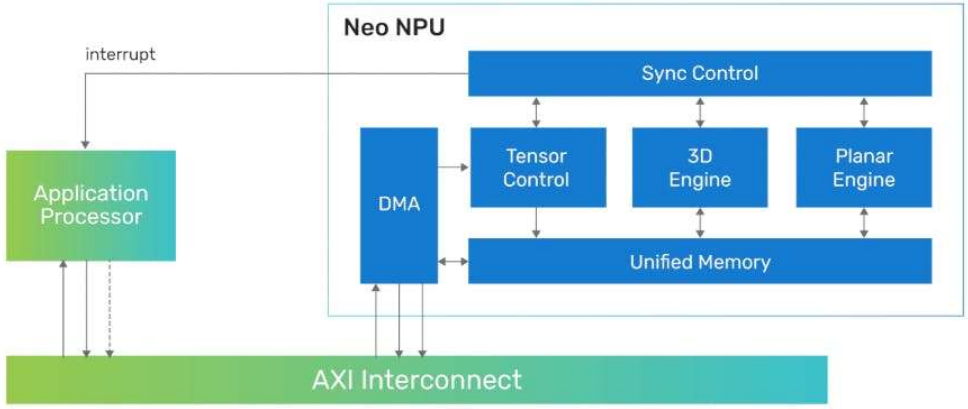
Peripherals: PCIe

- Peripheral Component Interconnect express (PCIe)
- Point to point connectivity, serial interface
- Multiple parallel lanes for increased bandwidth
  - Each lane is full duplex, up to 1GB/s
  - Clock is embedded in data coding



Peripherals: Neural accelerator

- Rise of AI, deep learning requires integrating neural network inference on chip (edge AI)
- Integrated, specialized implementation allows lower power, faster processing



References

[cummings1] C. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs", SNUG,2000

[cummings2] C. Cummings, "Finite State Machine (FSM) Deisgn & Synthesis using SystemVerilog - Part I", SNUG, 2019

Lab 1: RTL design

Lucas Valentin

analog.com

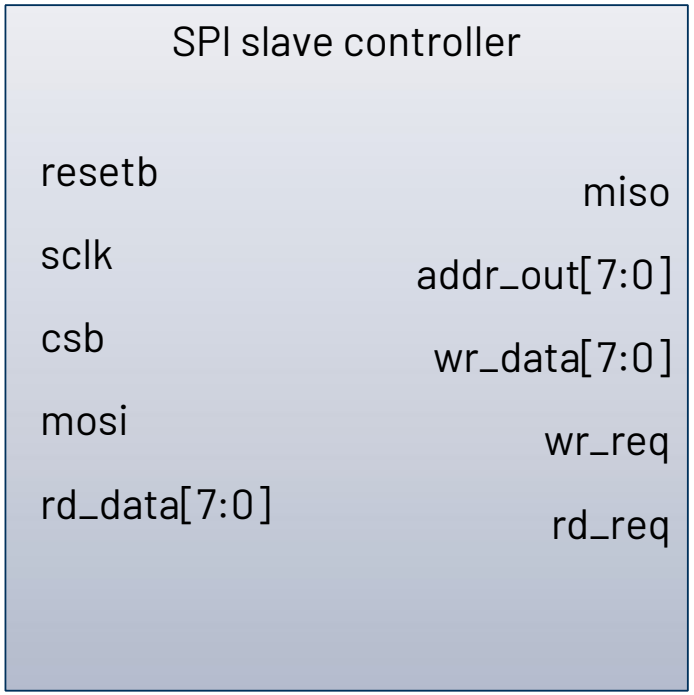
Lab 1: RTL design

- On the lab you will create 2 blocks found in many IC designs: a register map and an SPI slave controller
- Get the files at https://github.com/lucasval-vasic/vasic-uv-advanced-digital-design-lab1
- For each design a testbench has been created
  - Testbench will stimulate the design
  - Testbench will check design outputs, count errors, report pass/fail
- Lab goal is to get both simulations passing without errors
- Evaluation rubric

SPI compiles – 1 point	Register map compiles – 1 point
SPI no write errors – 2 points	Register map no write errors – 2 points
SPI no read errors – 2 points	Register map no read errors – 2 points

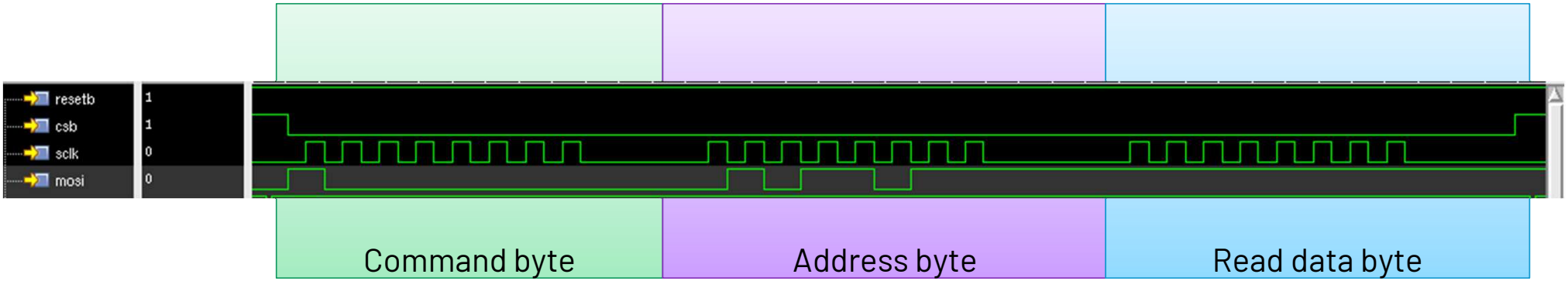
SPI slave controller

- Recommended implementation
  - FSM with multiple states to process each operation supported by controller
  - Synchronous design: single clock, all registers clocked off same edge

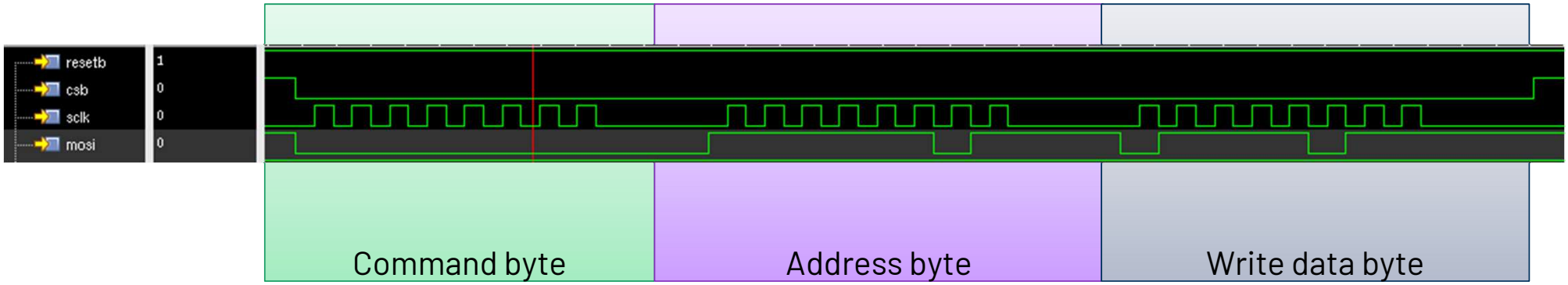


SPI frame

Read cycle

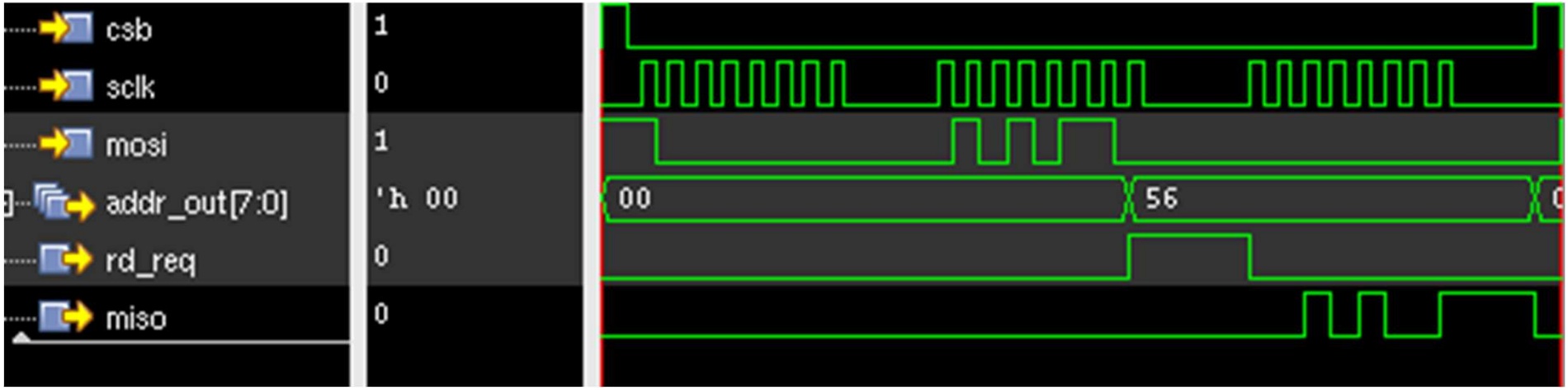


Write cycle



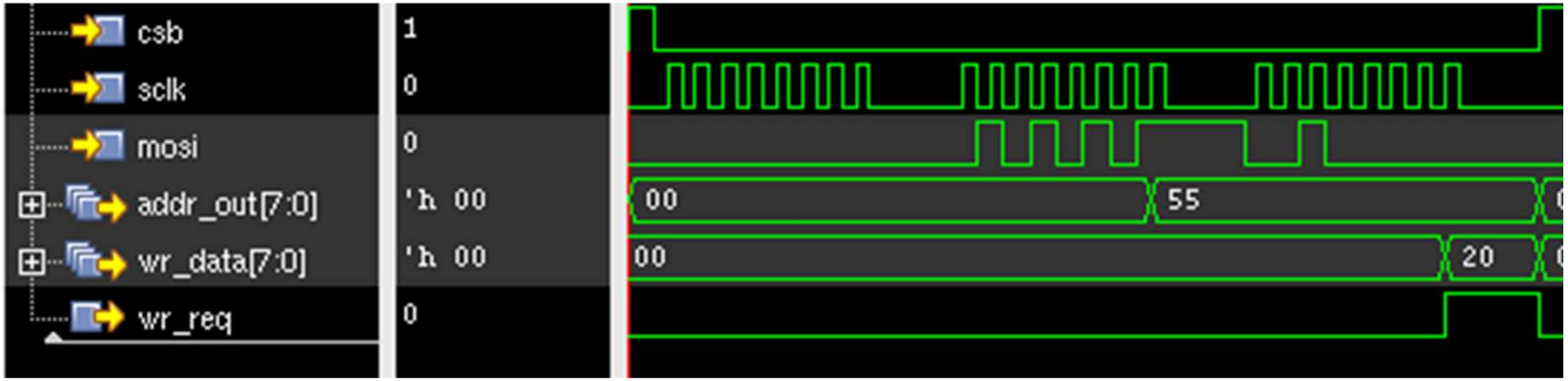
SPI slave controller simulation

- Testbench contains a 256 positions memory filled with random data
- Testbench will initially test read operations
  - Request SPI read from random address
  - Check SPI controller created a read request (address updated, rd_req asserted)
  - Check data sent by MISO line matches data in required address position



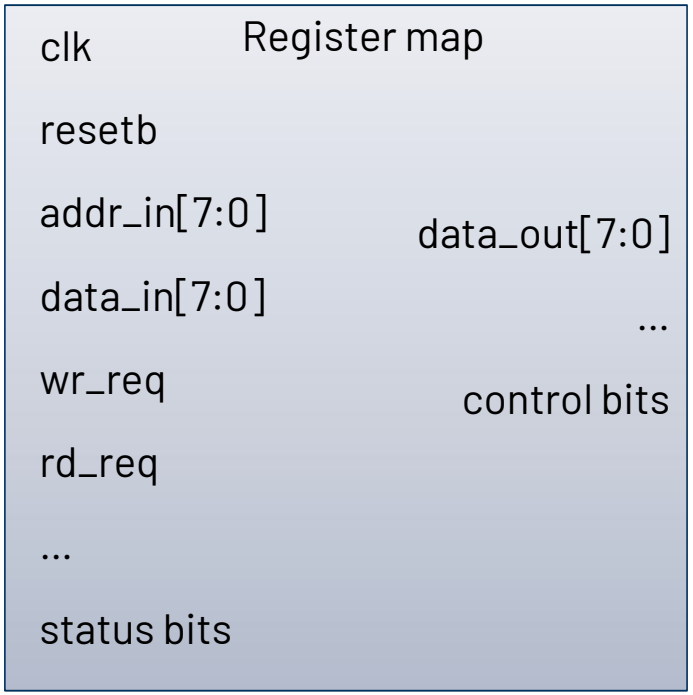
SPI slave controller simulation

- Finally testbench will check write operations
  - Request SPI write with random data
  - Check SPI controller creates a write request
  - Request SPI read from same address and check data has been updated
  - Note: wr_req is disabled when csb==1 (SPI inactive)



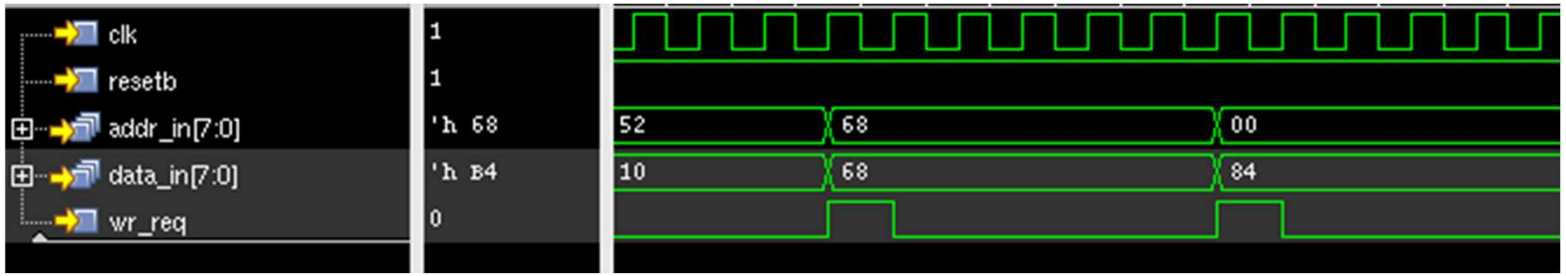
Register map

- Implement the register map from the table
- **Note:** status bits and control bits must be added as inputs and outputs respectively
- Read data must be registered
- Recommended implementation
  - Fully synchronous design (single clock, all registers clocked off same edge)



Register map simulation

- Testbench will first test the read-only registers
  - Stimulate status bits with random data, request read operation
- Then read-write registers will be tested
  - Initial defaults test: read default values
  - Then write random data, check control bits



Write cycle

Read cycle