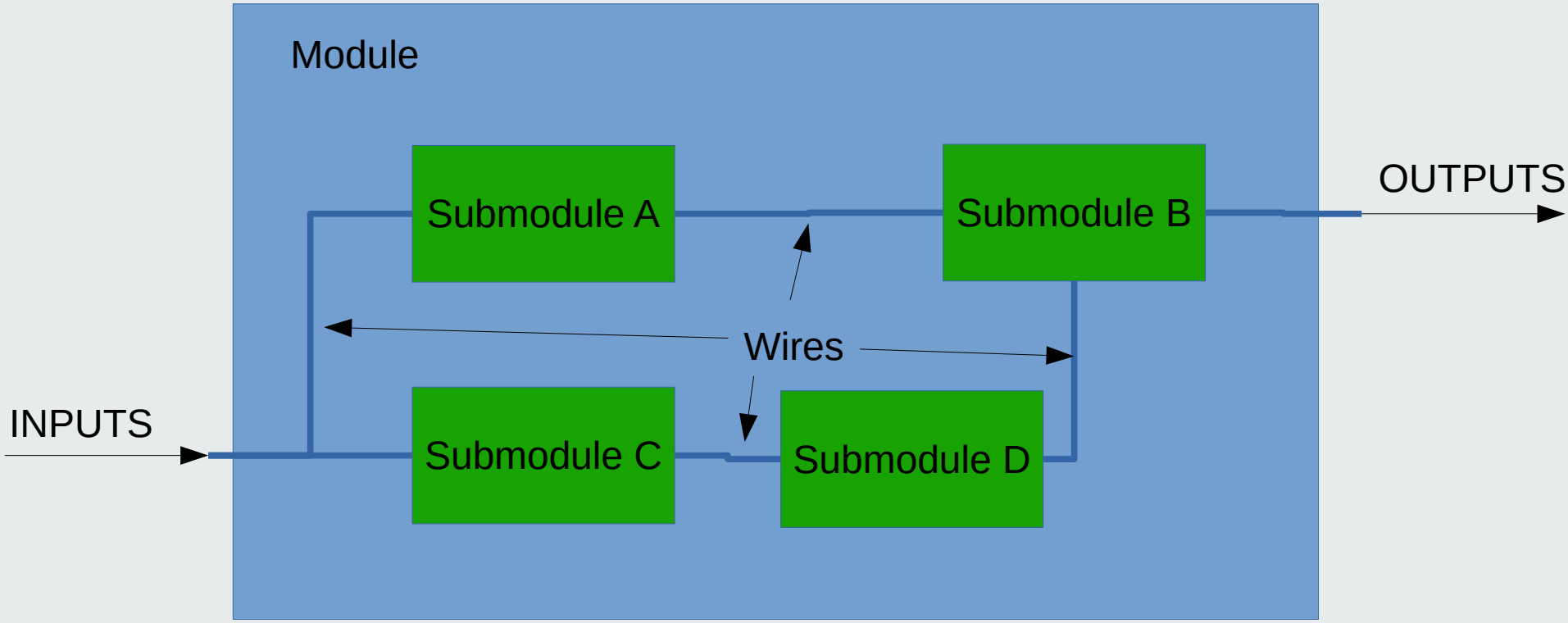
TÉCNICO LISBOA

Computer Electronics

Lecture 7: Digital Circuits using Verilog and IOb-Lib

Modular circuit design



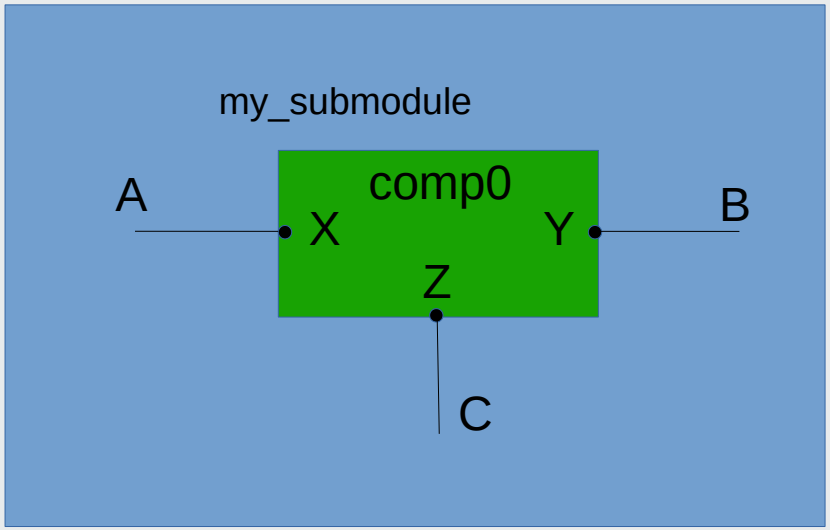
IOb-lib: module interface

```
module my_module (
  `INPUT(X, 8),   //declare 8-bit input
  `INPUT(Y, 8),
  `OUTPUT(Z, 16) //declare 16-bit output
  `INOUT(F, 32) //declare 32-bit bidirectional port
);
<module body>
endmodule
```

Instantiate sub-components

```
module my_module
(
 //module ports
...
);
...
   my_submodule comp0
   (
      .X(A),
      .Y(B),
      .Z ( C),
      .F (D)
   )
...
endmodule
```

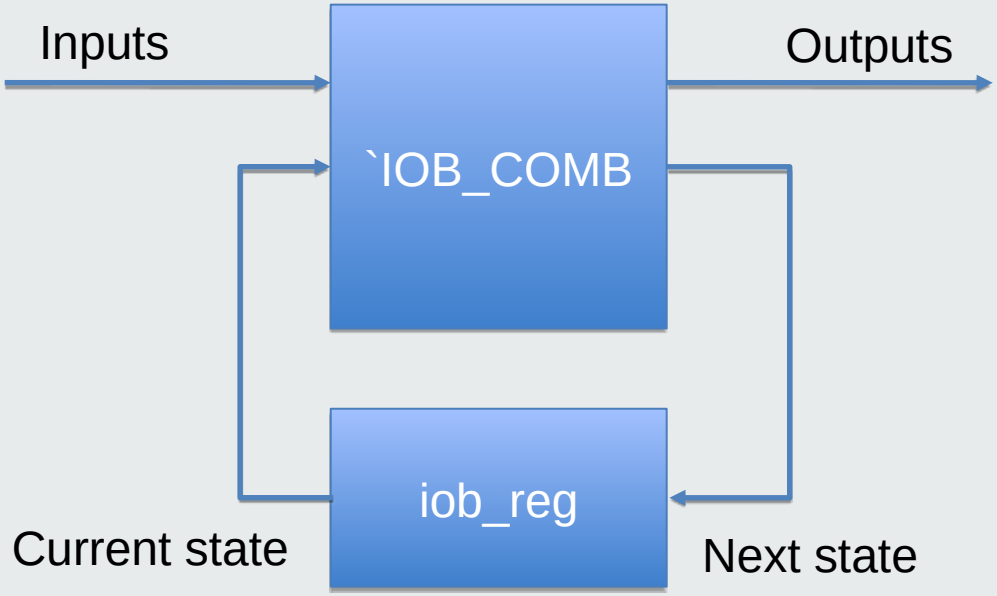
my_module



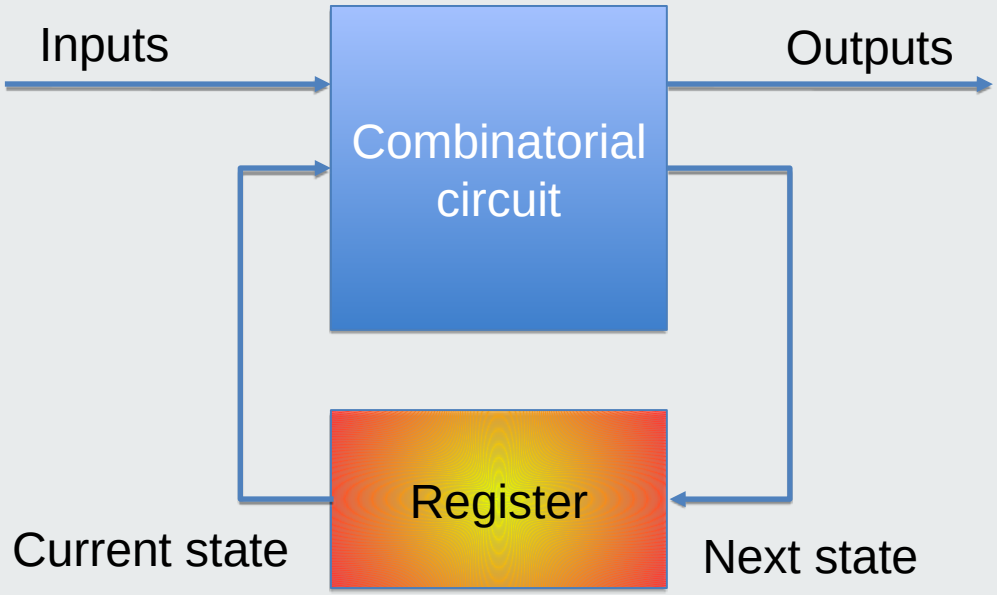
Wires and variables

- Circuits are blocks connected by wires that carry digital signals (0 or 1)
- So we need to declare and use wires
- Verilog describes two types of wires:
  - Type *wire:* used for connections and assign statements
  - Type *reg*: user for operation results within *always* and *initial* processes
- Using IOb Verilog macros:
  - Type wire: `IOB_WIRE(NAME, WIDTH)
  - Type reg: `IOB_VAR(NAME, WIDTH)
- Signed versions:
  - Type wire: `IOB_WIRE_SIGNED(NAME, WIDTH)
  - Type reg: `IOB_VAR_SIGNED(NAME, WIDTH)

Typical module: combinatorial circuit plus register



IOb-lib: registers!



IOb-lib: registers

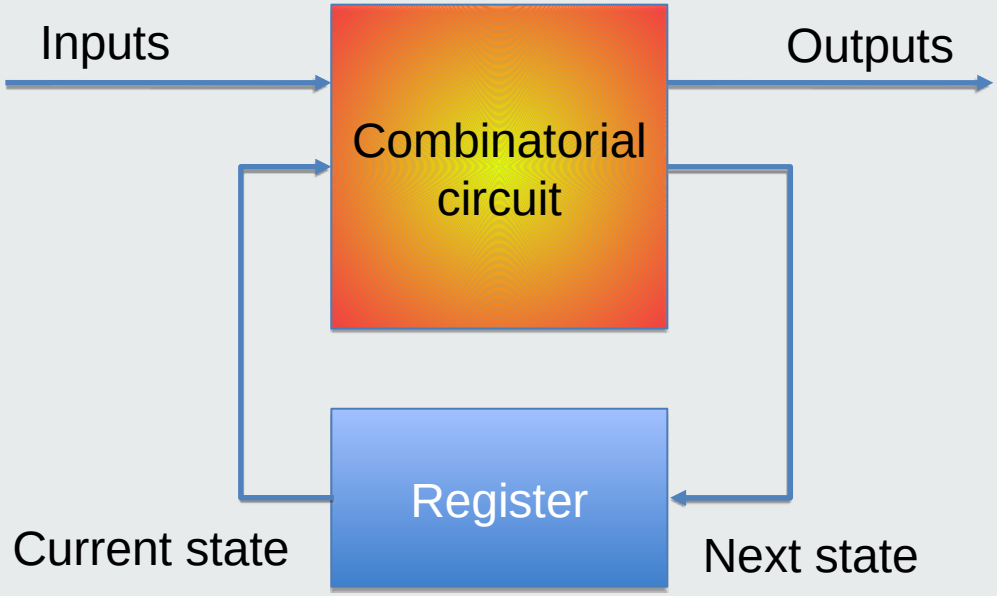
- IOb-lib users are not allowed to describe registers in Verilog (allowing tools to infer them)
- IOb-lib users must instantiate the iob_reg component

The iob_reg module

```
`timescale 1ns / 1ps

module iob_reg
  #(
    parameter DATA_W = 0,
    parameter RST_VAL = 0
    )
   (
    input                  clk,
    input                  arst,
    input                  rst,
    input                  en,
    input [DATA_W-1:0]      data_in,
    output reg [DATA_W-1:0] data_out
    );

   //prevent width mismatch
   localparam [DATA_W-1:0] RST_VAL_INT = RST_VAL;

   always @(posedge clk, posedge arst) begin
      if (arst) begin
         data_out <= RST_VAL_INT;
      end else if (rst) begin
         data_out <= RST_VAL_INT;
      end else if (en) begin
         data_out <= data_in;
      end
   end

endmodule
```

Exercise: instantiate a register

IOb-lib: combinatorial circuits



Combinatorial blocks

```
module my_module
(
 //interface signals
);
...
   `IOB_VAR(zv,10);
   `IOB_VAR(yv,10);
   `IOB_COMB begin
      //insert C-like expressions here: examples
      zv = s? X: Y;
      yv = a*(b+c);
   end
`IOB_VAR2WIRE(zv, z)
`IOB_VAR2WIRE(yv, y)
...
endmodule
```

WARNING: make sure you have no feedback loops

`IOB_COMB is defined in iob_lib.vh:
`define IOB_COMB always @*

What is always @*?

The verilog explanation is cumbersome

The **iob-lib** explanation is trivial:

1) `IOB_COMB describes a combinatorial block
2) LHS are computed variables using wires and other variables
3) RHS are inputs: wires or other variables
4) computed variables need to be converted to wires to be used for interconnection

Order of statements

- In Verilog the order of statements is **irrelevant** since we are merely listing components
- However:
  - Signal declarations must precede their use
  - Inside a combinatorial block, the order of statements is important because it is ***described procedurally***
  - Only the variables computed in a combinatorial block matters and this computation may be affected by the order of statements

Combinatorial block: if statements

```
module my_module
(
 //interface signals
);
...
   `IOB_VAR(zv,10);
   `IOB_COMB begin
   if (s)
      zv = x;
   else
      zv = y;
   end
`IOB_VAR2WIRE(zv, z)
...
endmodule
```

```
module my_module
(
 //interface signals
);
...
   `IOB_VAR(zv,10);
   `IOB_COMB begin
   zv = y;
   if (s)
      zv = x;
   end
`IOB_VAR2WIRE(zv, z)
...
endmodule
```

Combinatorial block: for loops

```
module my_module
(
 //interface signals
);
…
   integer  i;
   `IOB_VAR(c,9);
   `IOB_VAR(s,8);
   `IOB_COMB begin

   for (i=0; I<N; i=i+1)
       s[i] = a[i] ^b[i] ^c[i];
       c[i+1] = a[i]&b[i] | ( c[i] & (a[i] ^b[i] ));
   end
`IOB_VAR2WIRE(y, s)
`IOB_VAR2WIRE(co, c[8])
...
endmodule
```