Verilog introduction

Embedded and Ambient Systems Lab

Purpose of HDL languages

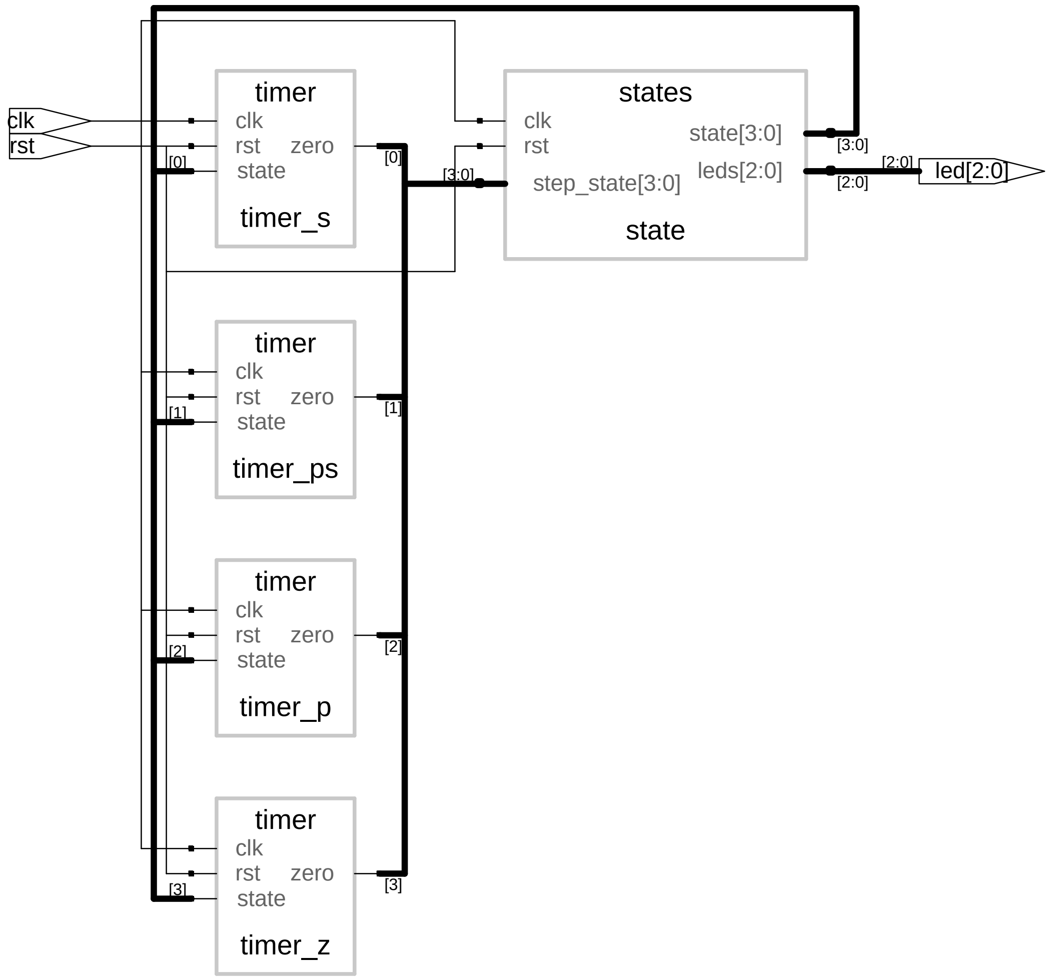
- Modeling hardware behavior
  - Large part of these languages can only be used for simulation, not for hardware generation (synthesis)
  - Synthesizable part depends on the actual synthesizer
- Replace graphical, schematic based design method (which very time consuming)
- RTL (Register Transfer Level) level description
  - Automatic hardware synthesis
  - Increase productivity

HDL languages

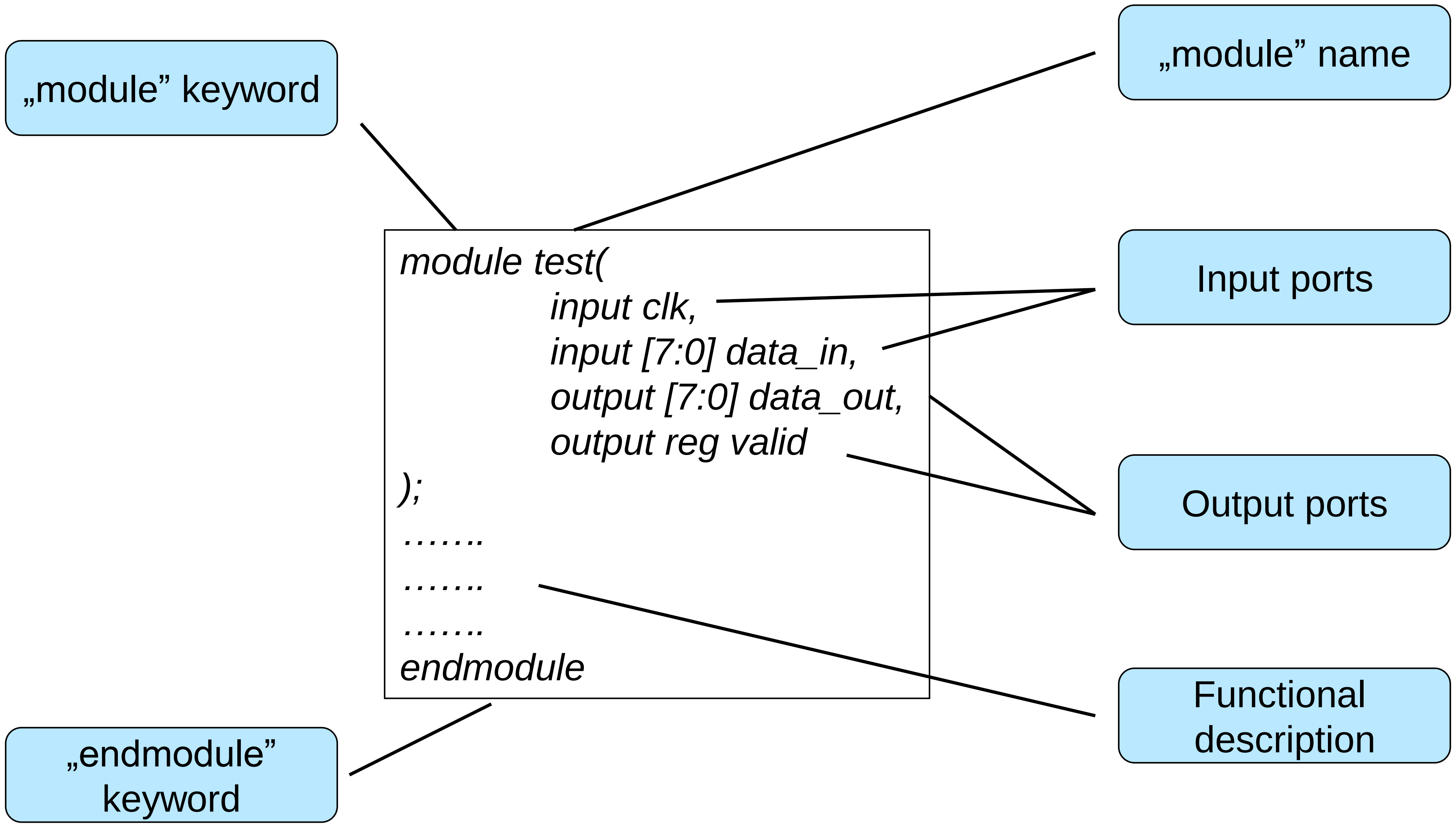
- Modular languages
- HDL module
  - Input and output port definitions
  - Logic equations between the inputs and the outputs
- Unlike software programming languages, NOT a sequential language
  - Describes PARALLEL OPERATIONS

Modules

- Building blocks to design complex, hierarchical systems
- Hierarchical description, partitioning



Verilog: module (2001)



```
module test(
        input clk,
        input [7:0] data_in,
        output [7:0] data_out,
        output reg valid
);
…….
…….
…….
endmodule
```

Verilog Syntax

- Comments (like C)
  - // one line
  - /* */ multiple lines
- Constants
  - <bit width><'base><value>

5'b00100:	00100	decimal value: 4, 5 bit wide
8'h4e:	01001110	decimal value: 78, 8 bit wide
4'bZ:	ZZZZ	high impedance state

Bit operations

- ~, &, |, ^, ~^ (negate, and, or, xor, xnor)
- Bitwise operator on vectors, e.g.:
  - 4’b1101 & 4’b0110 = 4’b0100
- If the operand widths are not equal, the smaller one is extended with zeros
  - 2’b11 & 4’b1101 = 4’b0001
- (Logic operators: !, &&, ||)

Bit reduction operators

- Operates on all bits of a vector, the output is a single bit
- &, ~&, |, ~|, ^, ~^ (and, nand, or, nor, xor, xnor)
  - &4'b1101 = 1'b0
  - |4'b1101 = 1'b1
  - Typical usage scenarios:
    - Parity check

Comparison

- Same as in C
- Equal, not-equal
  - ==, !=
  - ===: equality considering „Z”, „X”
  - !==: not-equal considering „Z”, „X”
- Comparison
  - <, >, <=, >=

Arithmetic

- Same as in C
- Operators: +, -, *, /, %
  - Not all of them is synthesizable
    - Typically division, modulo are only synthesizable when the second operator is power of 2
  - Negative numbers in twos-complement code

Other operators

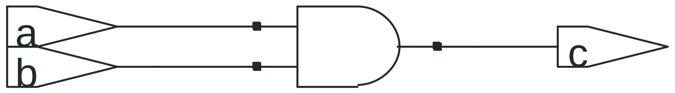
- Concatenate: {}
  E.g.:
  - {4’b0101, 4’b1110} = 8’b01011110
- Shift:
  - <<, >>
- Bit selection
  - Selected part has to be constant
  - data[5:3]

Data types

- wire
  - Behaves like a real wire (combinatorial logic)
  - Declaration of an 8 bit wire: wire [7:0] data;
- reg
  - After synthesis it can translate into
    - Wire
    - Latch
    - Flip-flop
  - E.g.: reg [7:0] data;

Assign

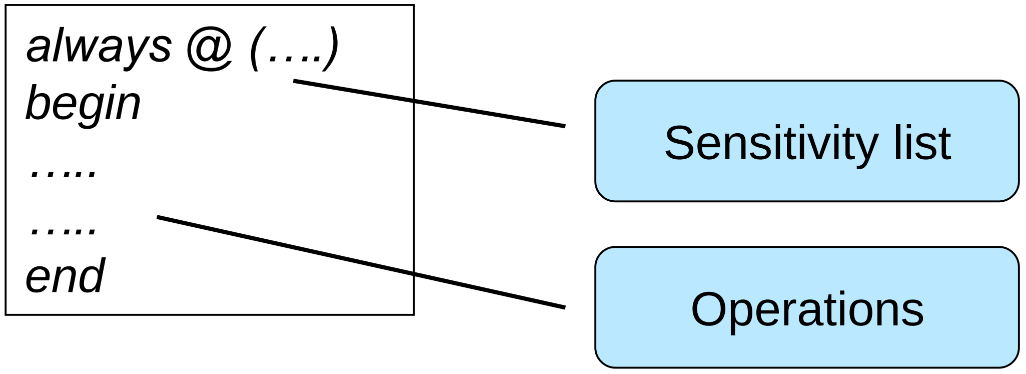
- Assign can be used only on wire types
- Continuous assignment
  - Left operand continuously gets a new value
- E.g.
  - assign c = a & b;



- A wire can be driven by only one assign statement
- Multiple assigns operate parallel to each other
- Can be used to describe combinatorial logic

Always block

- Syntax:



- A reg type variable should be written only in one always block
- The sensitivity list cannot contain the outputs (left-side variables) of the always block
- Assign cannot be used within an always block
- Multiple always blocks are executed in parallel

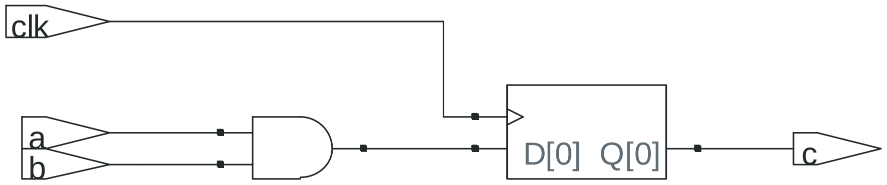
Always – assignments

- Blocking: =
  - Blocks the execution of operations after it till it is executed -> sequential operation (don’t use it unless really necessary)
- Nonblocking: <=
  - Nonblocking assignments are executed in parallel -> hardware-like operation
- **Always use nonblocking assignment**

Always – Flip Flop

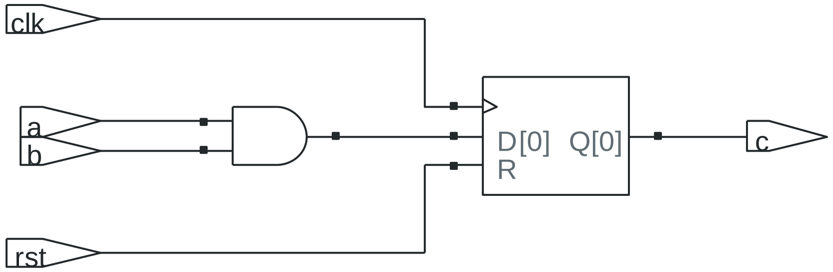
- Flip Flop: edge sensitive storage element

```
always @ (posedge clk)
            c <= a & b;
```



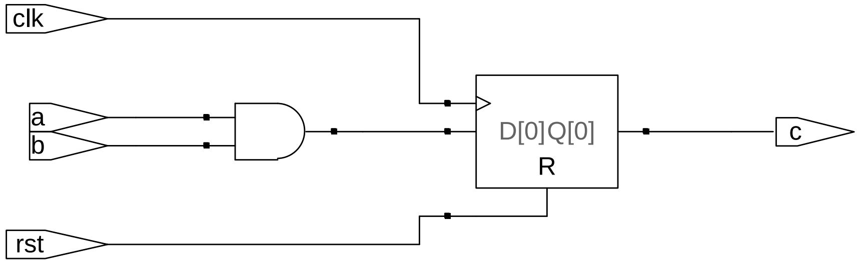
- Synchronous reset

```
always @ (posedge clk)
if (rst)
            c <= 1'b0;
else
            c <= a & b;
```



- Asynchronous reset

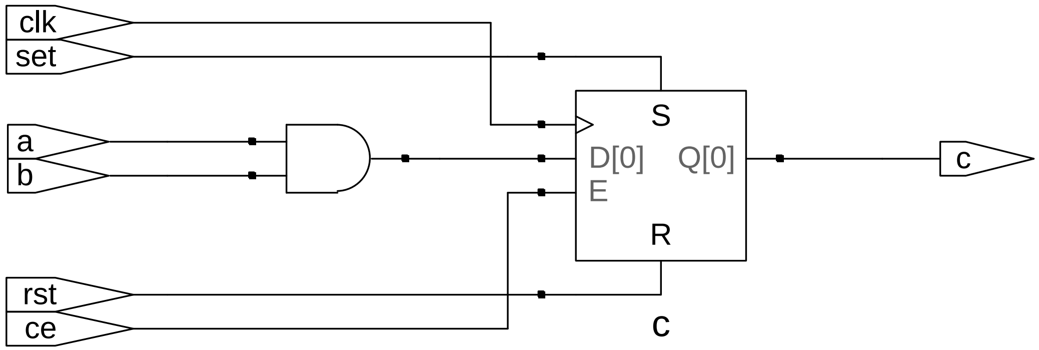
```
always @ (posedge clk, posedge rst)
if (rst)
            c <= 1'b0;
else
            c <= a & b;
```



Always – Flip Flop

- In Xilinx FPGAs
  - Reset and set can be synchronous or asynchronous
  - Priority in synchronous case:
    - reset, set, ce
- Asynchronous example:

```
always @ (posedge clk, posedge rst,
posedge set)
if (rst)
        c <= 1'b0;
else if (set)
        c <= 1'b1;
else if (ce)
        c <= a & b;
```

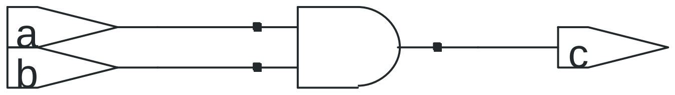


Always – comb. logic

- Result is continuously calculated – if any of the inputs changes the output immediately changes

```
always @ (a, b)
   c <= a & b;
```

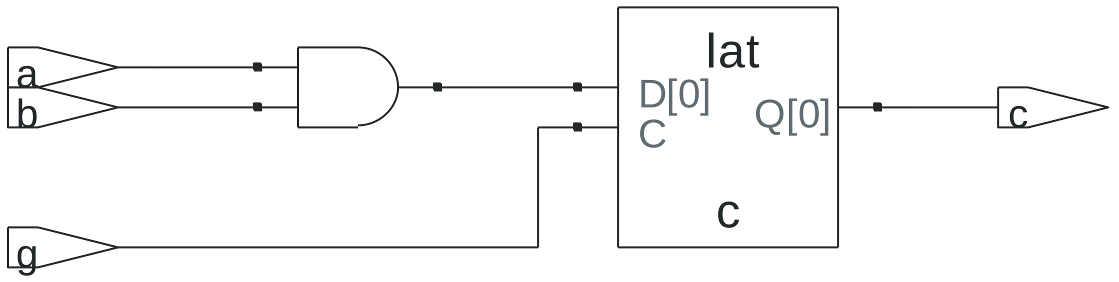
```
always @ (*)
   c <= a & b;
```



Always – latch

- Latch: level sensitive storage element
  - as long as the „gate” input is ‘1’, the input is sampled into the latch
  - If the „gate” input is ‘0’, the previously sampled value is kept

```
always @ (*)
If (g)
   c <= a & b;
```

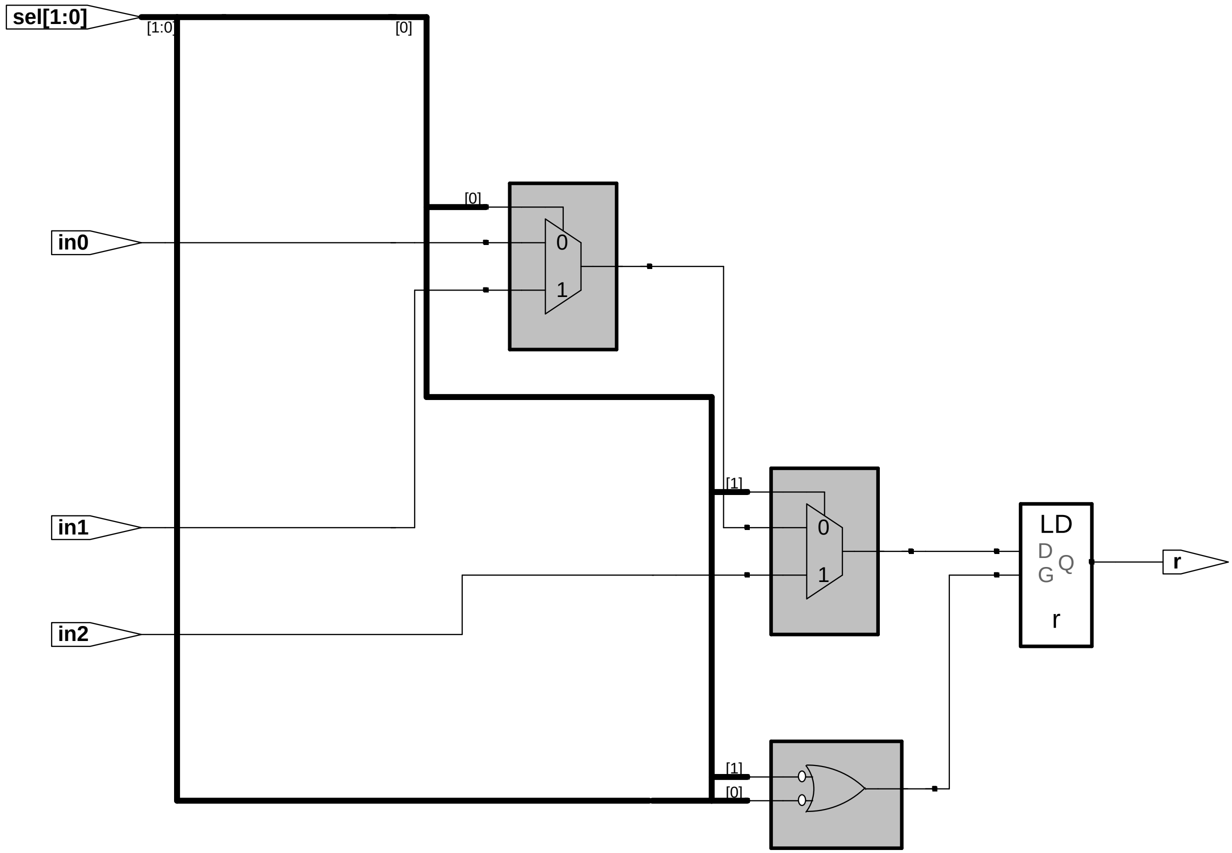


Always – latch error

- Using latch is typically a bad idea; it can be generated by wrong code
  - Not full if or case statements
  - Synthesizers typically give a warning

```
always @ (*)
case (sel)
   2'b00: r <= in0;
   2'b01: r <= in1;
   2'b10: r <= in2;
endcase
```

```
always @ (*)
if (sel==0)
   r <= in0;
else if (sel==1)
   r <= in1;
else if (sel==2)
   r <= in2;
```

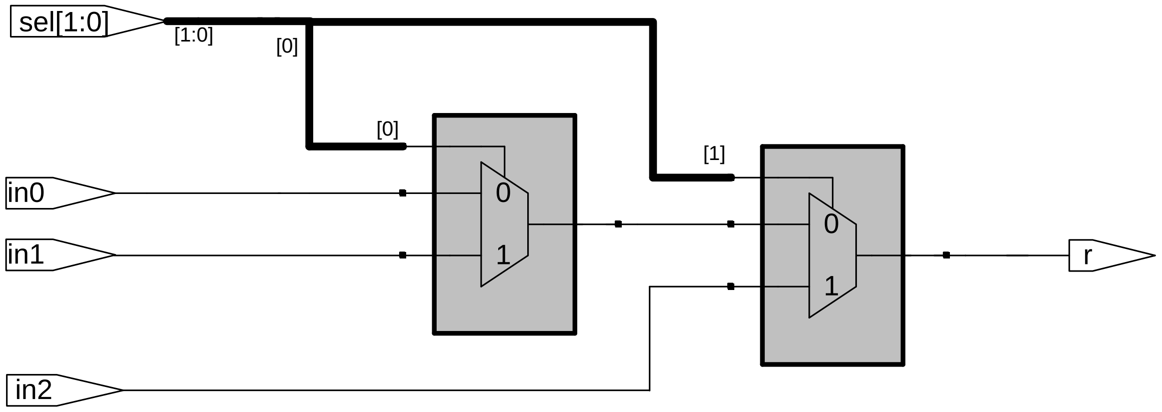


Always – correct if/case

- Correct code using combinatorial if/case

```
always @ (*)
case (sel)
   2'b00: r <= in0;
   2'b01: r <= in1;
   2'b10: r <= in2;
   default: r <= 'bx;
endcase
```

```
always @ (*)
if (sel==0)
   r <= in0;
else if (sel==1)
   r <= in1;
else
   r <= in2;
```

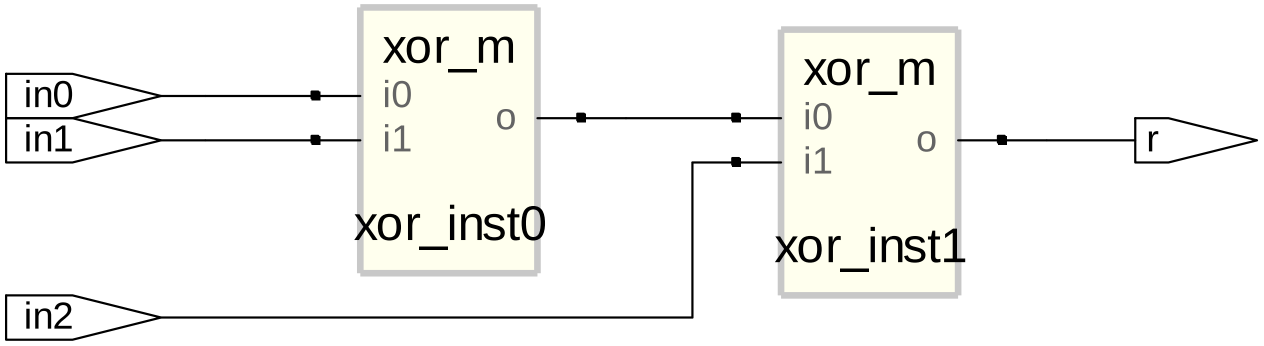


Structural description

- Creating hierarchy: connecting modules

```
module top_level (input in0, in1, in2, output r);

wire xor0;
xor_m xor_inst0(.i0(in0), .i1(in1), .o(xor0));
xor_m xor_inst1(.i0(xor0), .i1(in2), .o(r));

endmodule
```

- Port – signal assignment based on the port names



Example – MUX (1.)

- 2:1 multiplexer

```
module mux_21 (input in0, in1, sel, output r);
assign r = (sel==1’b1) ? in1 : in0;
endmodule
```

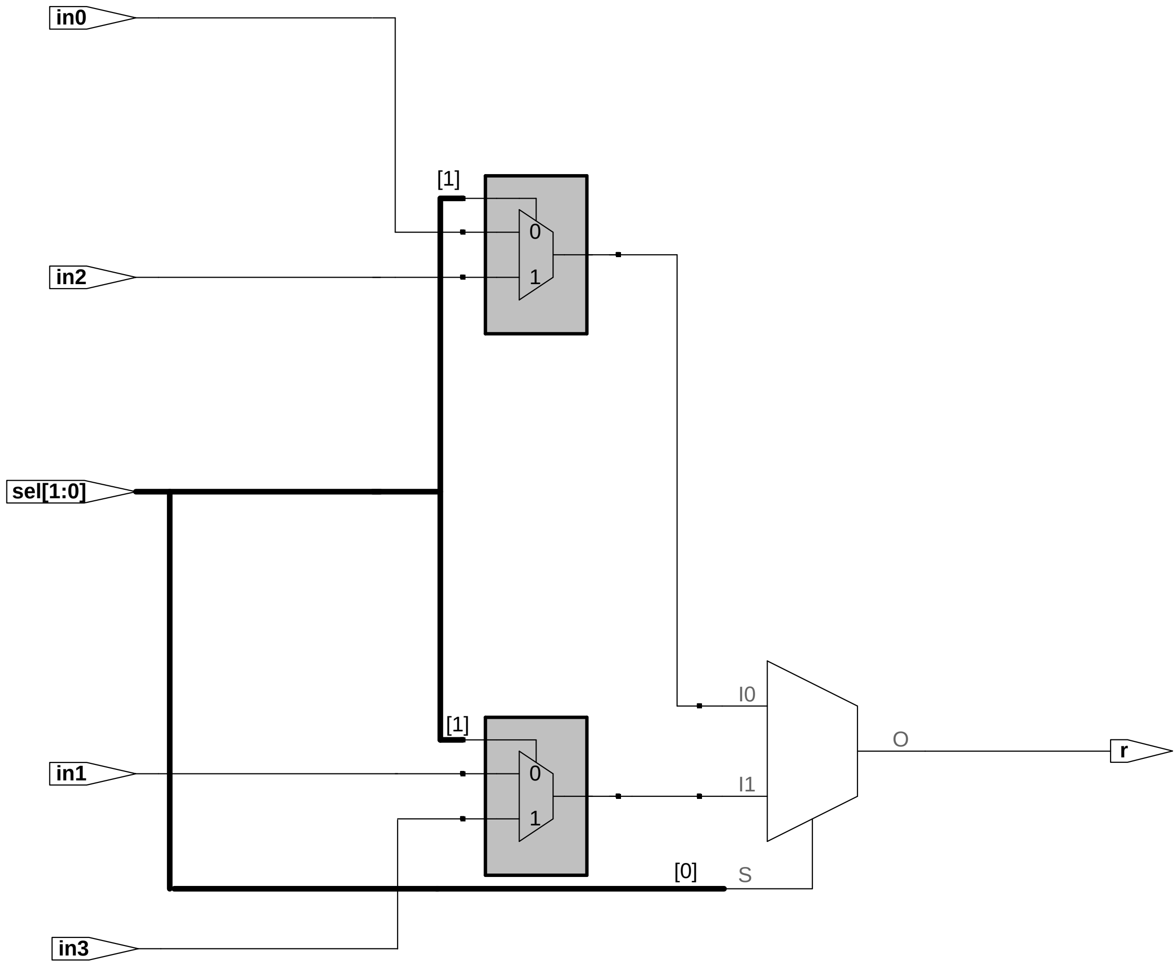
```
module mux_21 (input in0, in1, sel, output reg r);
always @ (*)
if (sel==1’b1) r <= in1;
else           r <= in0;
endmodule
```

```
module mux_21 (input in0, in1, sel, output reg r);
always @ (*)
case(sel)
    1’b0:      r <= in0;
    1’b1:      r <= in1;
endmodule
```

Example – MUX (2.)

- 4:1 multiplexer

```
module mux_41 (input in0, in1, in2, in3, input [1:0] sel, output reg r);
always @ (*)
case(sel)
      2'b00: r <= in0;
      2'b01: r <= in1;
      2'b10: r <= in2;
      2'b11: r <= in3;
endcase
endmodule
```



Example – Shift register

- 16 bit deep shift register (e.g. for delaying a value)

```
module shr (input clk, sh, din, output dout);

reg [15:0] shr;
always @ (posedge clk)
if (sh)
      shr <= {shr[14:0], din};

assign dout = shr[15];

endmodule
```

Example – Counter

- Binary counter with synchronous reset, clock enable, load and direction inputs

```
module m_cntr (input          clk, rst, ce, load, dir,
                input  [7:0] din,
               output [7:0] dout);

reg [7:0] cntr_reg;
always @ (posedge clk)
if (rst)
      cntr_reg <= 0;
else if (ce)
      if (load)
               cntr_reg <= din;
      else if (dir)
               cntr_reg <= cntr_reg – 1;
      else
               cntr_reg <= cntr_reg + 1;

assign dout = cntr_reg;

endmodule
```

Example – Secundum counter

- 50 MHz clock frequency, 1 sec = 50 000 000 clocks

```
module sec (input clk, rst, output [6:0] dout);

reg [25:0] clk_div;
wire tc;
always @ (posedge clk)
If (rst)
        clk_div <= 0;
else
        if (tc)
                clk_div <= 0;
        else
                clk_div <= clk_div + 1;

assign tc = (clk_div == 49999999);

reg [6:0] sec_cntr;
always @ (posedge clk)
If (rst)
        sec_cntr <= 0;
else if (tc)
        if (sec_cntr==59)
                sec_cntr <= 0;
        else
                sec_cntr <= sec_cntr + 1;

assign dout = sec_cntr;

endmodule
```

Tri-state lines

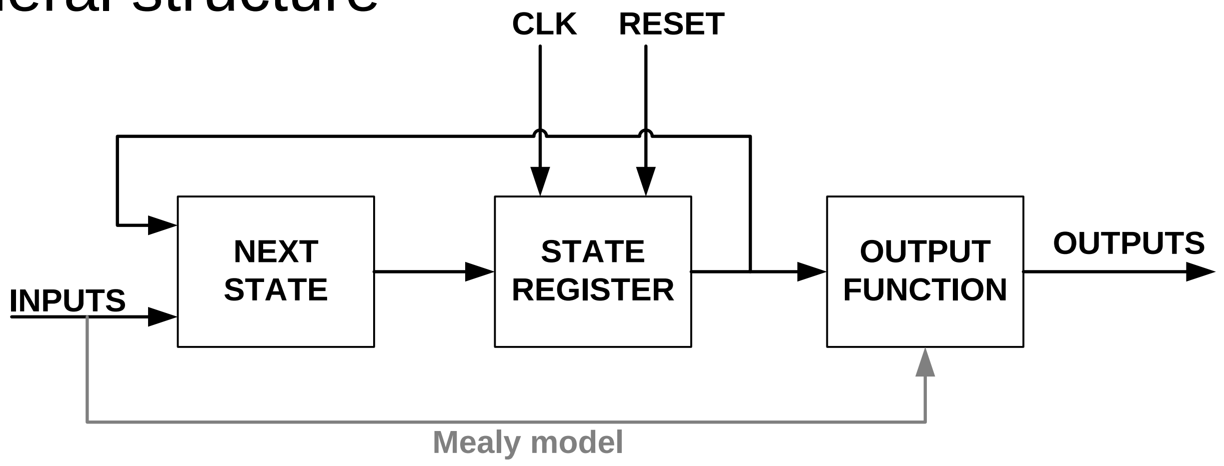
- Bi-directional buses, eg.
  - E.g. data bus of external memories

```
module tri_state (input clk, inout [7:0] data_io);

wire [7:0] data_in, data_out;
wire bus_drv;

assign data_in = data_io;
assign data_io = (bus_drv) ? data_out : 8'bz;

endmodule
```

- The bus drive enable signal is critical (bus_drv), take care when generating it

FSM – Finite State Machine

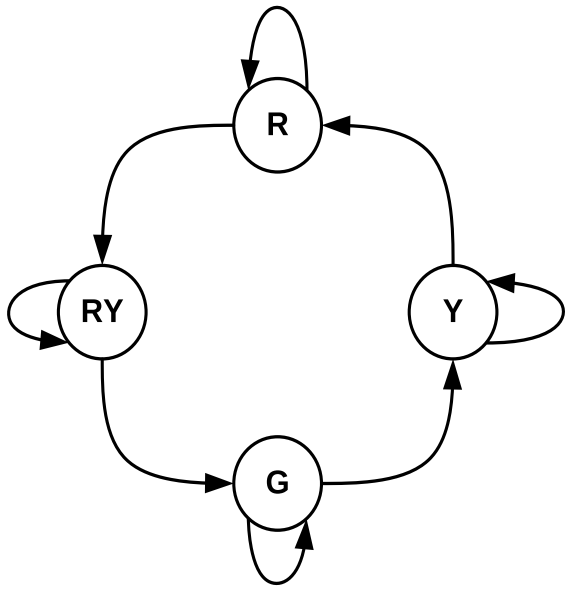
- FSM – to create complex control machines
- General structure



- State register: state variable
- Next state function: determines the next state (combinatorial logic)
- Output function: generates outputs
  - Moore: based on the state register
  - Mealy: based on the state registers and the current inputs

FSM example

- Traffic light (simple)
  - States: red, yellow, green, red-yellow (no blinking yellow)
  - Inputs: timers for the different states
  - Output: state



FSM example – Verilog (1)

```
module light(
        input              clk, rst,
        output reg [2:0] led);

parameter RED        = 2'b00;
parameter RY          = 2'b01;
parameter GREEN    = 2'b10;
parameter YELLOW  = 2'b11;

reg [15:0] timer;
reg [1:0] state_reg;
reg [1:0] next_state;

always @ (posedge clk)
if (rst)
        state_reg <= RED;
else
        state_reg <= next_state;
```

```
always @ (*)
case(state_reg)
        RED: begin
          if (timer == 0)
            next_state <= RY;
          else
            next_state <= R;
          end
        RY: begin
          if (timer == 0)
            next_state <= GREEN;
          else
            next_state <= RY;
          end
        YELLOW: begin
          if (timer == 0)
            next_state <= RED;
          else
            next_state <= YELLOW;
          end
        GREEN: begin
          if (timer == 0)
            next_state <= YELLOW;
          else
            next_state <= GREEN;
          end
        default:
          next_state <= 3'bxxx;
endcase
```

FSM example – Verilog (2)

```
always @ (posedge clk)
case(state_reg)
        RED: begin
          if (timer == 0)
            timer <= 500;       //next_state <= RY;
          else
            timer <= timer - 1;
          end
        RY: begin
          if (timer == 0)
            timer <= 4000;      //next_state <= GREEN;
          else
            timer <= timer - 1;
          end
        YELLOW: begin
          if (timer == 0)
            timer <= 4500;      //next_state <= RED;
          else
            timer <= timer - 1;
          end
        GREEN: begin
          if (timer == 0)
            timer <= 500;       //next_state <= YELLOW;
          else
            timer <= timer - 1;
          end
endcase
```

- Timer
  - Loads a new value when state changes
  - Down-counter
  - ==0: state change

```
always @ (*)
case (state_reg)
        RY   :                  led <= 3'b110;
        RED:                    led <= 3'b100;
        YELLOW:                 led <= 3'b010;
        GREEN:                  led <= 3'b001;
        default:                led <= 3’b100;
endcase

endmodule
```

Simulation

- Testbench creation: drive the input ports of the unit under test
  - Verilog Test Fixture
    - Generating inputs using Verilog
    - All synthesizable language elements can be used
    - Other constructs which are not synthesizable are also available (e.g. delay)
- Simulator
  - ISE Simulator (Isim)
  - Modelsim (MXE)

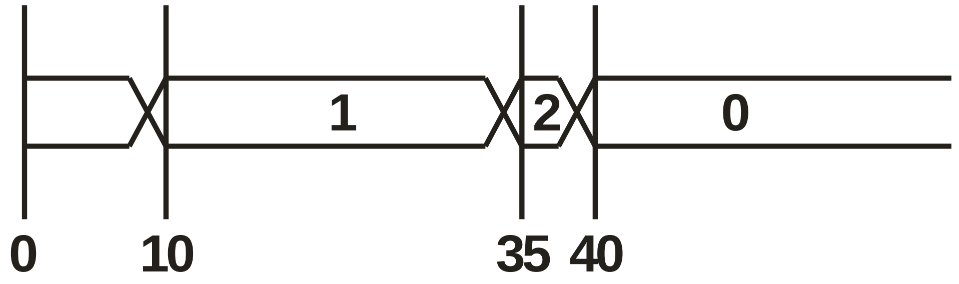
Verilog Test Fixture

- Test Fixture
  - Test Fixture is a Verilog module
  - The module under test is a sub-module of the test fixture
  - All Verilog syntax constructs can be used
  - There are non-synthesizable constructs
- Time base
  - ‘timescale 1ns/1ps
    - Time base is 1 ns
    - Simulation resolution: 1 ps

Test Fixture - initial

- „initial” block
  - Execution starts at time „0”
  - Executed once
  - „initial” blocks are executed in parallel with each other and with always blocks and assigns
- The delays are cumulative, e.g.

```
initial
begin
    a <= 0;
    #10 a <= 1;
    #25 a <= 2;
    #5   a <= 0;
end
```



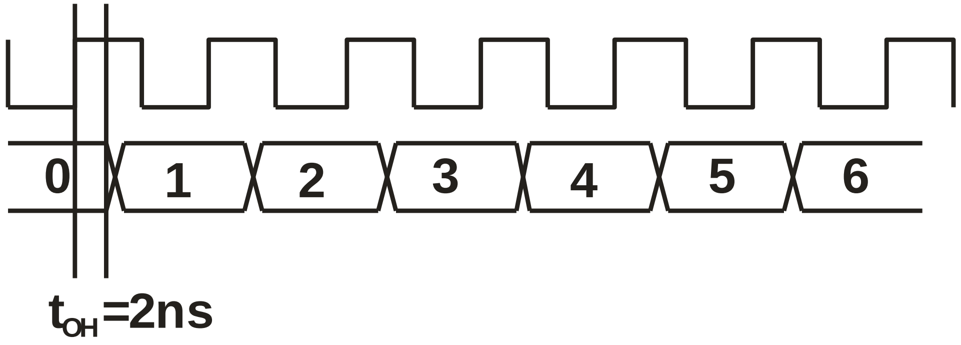
Test Fixture - always

- Generating clock

```
initial
    clk <= 1;

always #5
    clk <= ~clk;
```

- Clocked inputs (propagation time!)

```
initial cntr <= 0;
always @ (posedge clk)
    #2 cntr <= cntr + 1;
```

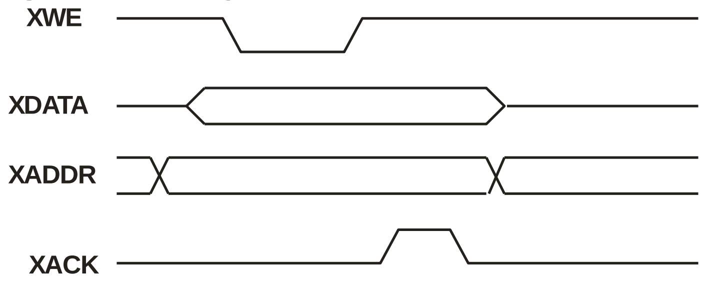


Task

- Declaration:
  - In the module which uses the task
  - In a different file (more modules can use the same task)
- Arbitrary number of inputs and outputs
- Can contain timing (delay)
- Variables declared in a task are local variables
- Global variables can be read or written by the task
- A task can call another task

Example - Task

- Simulating an asynchronous read operation



- Verilog code

```
task bus_w(input [15:0] addr, input [7:0] data);
begin
          xaddr <= addr;
     #5   xdata <= data;
     #3   xwe   <= 0;
     #10  xwe   <= 1;
          while (xack != 1) wait;
     #4   xdata <= 8'bz;
          xaddr <= 0;
end
endtask;
```

Example - Task

- „bus_w” is located in „tasks.v” file
- x* variables used by the task are global variables defined in the test fixture
- Using the task in a test fixture
  - 3 write cycles
  - 10 ns between them

```
`include “tasks.v”

initial
begin
            bus_w(16’h0, 8’h4);
     #10 bus_w(16’h1, 8’h65);
     #10 bus_w(16’h2, 8’h42);
end
```

File operations

- Reading data into an array

```
reg [9:0] input_data[255:0];
initial
     $readmemh(“input.txt”, input_data);
```

- Writing data into a file

```
integer file_out;
wire res_valid;
wire [16:0] res;

initial
     file_out =$fopen(“output.txt");

always @ (posedge clk)
if (out_valid)
     $fwrite(file_out, "%d \n", res);
```