TEL AVIV UNIVERSITY אוניברסיטת תל אביב

Iby and Aladar Fleischman Faculty of Engineering Tel Aviv University
הפקולטה להנדסה ע"ש איבי ואלדר פליישמן אוניברסיטת תל-אביב

# Advanced Architecture Lab-3 Briefing

School of Electrical Engineering

Faculty of Engineering

Tel Aviv University

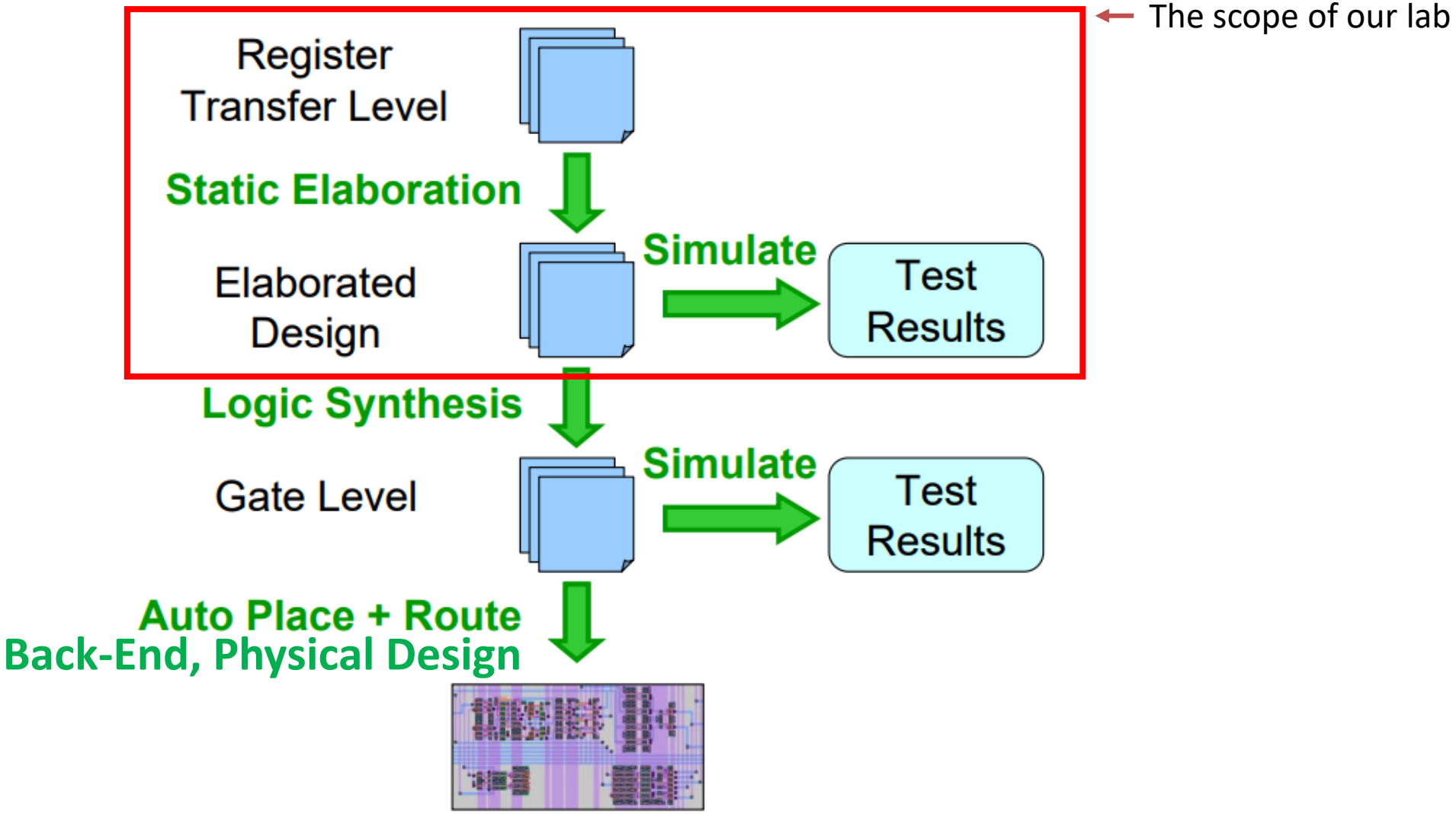
# Verilog

- The HDL we will use in our lab is Verilog
- Verilog's capabilities:
  - Design and Verification of digital circuits
  - Verification of mixed-signal or analog circuits
- Established in 1984; kept proprietary until 1991;
  - 1995: Verilog-95: first IEEE-standardized version!
  - 2001: Verilog-2001: significantly upgraded! ← We use this
  - 2005: Verilog-2005: insignificant upgrades
  - 2009: Verilog was merged into SystemVerilog
- http://www.asic-world.com/verilog is the best guide

# Verilog is not a programming language

- It is a hardware description language
- The code describes an underlying circuit
- All the code is **continuously active in parallel**, not like C which is sequential
- Verilog syntax is very similar to the C syntax

# Design Flow


The scope of our lab
Register Transfer Level
Static Elaboration
Elaborated Design
Simulate
Test Results
Logic Synthesis
Gate Level
Simulate
Test Results
Auto Place + Route
Back-End, Physical Design

# Hardware Data types

- **Single Bit:**

  `reg  r1` – r1 keeps a value, which can be changed on another assignment

  `wire w1` – w1 is a net, which must be constantly driven by a signal

- **Busses and multiple bit registers** (MSB is on the left, unsigned by default)

```
reg  [3:0] r1
wire [3:0] w1
```

- **Signed busses and multiple bit registers** (two's complement)

```
reg  signed [3:0] r1
wire signed [3:0] w1
```

# Virtual Data types

`integer` – 32-bit two's complement integer value

`real` – 64-bit (double precision) IEEE floating point value

`realtime` – used for storing time as floating point

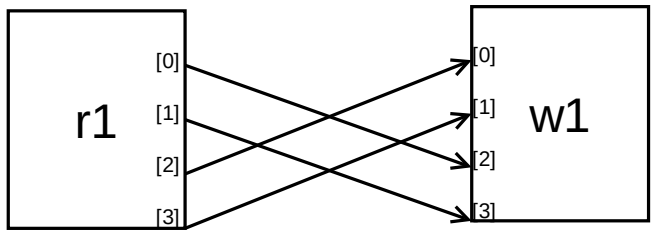
`time` – 64 bit unsigned value for the simulation time counting

`genvar` - positive integer values. The index variable used in a `generate` loop must be declared as a genvar

# Operators

- Reduction means “tree”:
  - `^A[2:0]` is a xor-tree of 3 bits - returns 1 bit, equivalent to `A[2]^A[1]^A[0]` and to `^{A[2],A[1],A[0]}`
- Bitwise:
  - `A & B` performs bitwise and between buses A and B
- Logical
  - `A == B && B==C` returns only one bit!
- Arithmetic
  - `A >>> 3` is arithmetic right shift of A by 3 bits
  - Comparison `A < B` respects whether A and B are signed.
- Concatenation operator

```
w1 = {r1[1:0],r1[3:2]};
```




| | | |
|---|---|---|
| ! | logical negation | logical |
| ~ | negation | bit-wise |
| & | reduction AND | reduction |
| \| | reduction OR | reduction |
| ~& | reduction NAND | reduction |
| ~\| | reduction NOR | reduction |
| ^ | reduction XOR | reduction |
| ~^ or ^~ | reduction XNOR | reduction |
| + | unary (sign) plus | arithmetic |
| - | unary (sign) minus | arithmetic |
| { } | concatenation | concatenation |
| {{ }} | replication | replication |
| * | multiply | arithmetic |
| / | divide | arithmetic |
| % | modulus | arithmetic |
| + | binary plus | arithmetic |
| - | binary minus | arithmetic |
| << | shift left | shift |
| >> | shift right | shift |
| > | greater than | relational |
| >= | greater than or equal to | relational |
| < | less than | relational |
| <= | less than or equal to | relational |
| == | case equality | equality |
| != | case inequality | equality |
| & | bit-wise AND | bit-wise |
| ^ | bit-wise XOR | bit-wise |
| \| | bit-wise OR | bit-wise |
| && | logical AND | logical |
| \|\| | logical OR | logical |
| ?: | conditional “ternary if-the-else” | conditional |

# Signal values

- single bit values: **0, 1, Z, X**
- multiple bits : use the pattern [Width]'[Radix][Value]
  - For example: `4'b1100`
  - Number of **bits** binary fill the string 1100
- Other available radices `'h` `'d` for hexadecimal and decimal
- The bit width of the value must match the width of the wire/reg
- Default representation radix is decimal:
  - Example: all following unsigned numbers are equal:
  - Example: all following signed numbers are equal:

| | | | |
|---|---|---|---|
| X=12; | X=4'hc; | X=4'b1100; | X=4'd12; |

| | | | |
|---|---|---|---|
| X=-1; | X=4'hf; | X=4'b1111; | X=-4'd1; |
| | X=-4'h1 | X=-4'b0001 | |

# Module Definition

- Module describes a synchronous/combinational circuit
- Good practice: write each module in its own .v source file
- There must be one **top module** that interfaces with the external system

```
module Fifo(in,push,out);
   parameter N=2;
   input in;
   input push;
   output out;
   //some logic goes here
endmodule
```

**Parameters:** are set once per module instance, they are not changed during operation as opposed to inputs/outputs

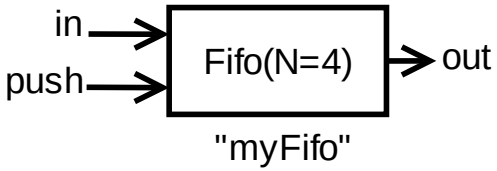
**Input/Output ports:** are ports through which the signals are fed in/out of the module.

# Module Instantiation

- Design hierarchy is constructed by instantiating modules in other modules
- An instance of a module is a use of this module in another, higher-level module
- **Single instance:**

```
reg in, push;
wire out;
Fifo #(4) myFifo(in, push, out);
```

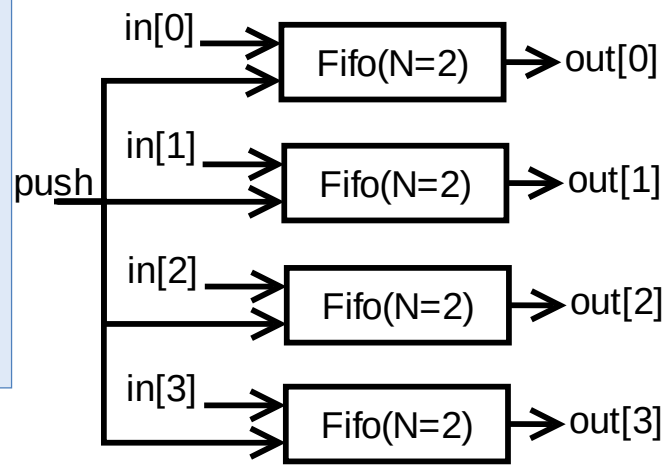
Module name  Param.  Instance name  Wirings



# Module Instantiation

- **Array of instances**

```
reg [3:0] in;
reg push;
wire [3:0] out;
Fifo fifoArr[3:0](in, {4*{push}}, out);
```

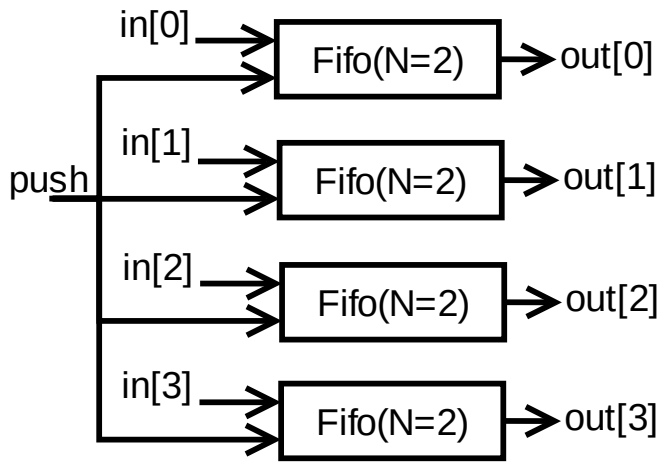


# Generate block

```
reg [3:0] in;
reg push;
wire [3:0] out;
genvar i;
generate
 for ( i = 0; i < 4; i = i+1 )
  begin : fifoArr
   Fifo f(in[i], push, out[i]);
  end
endgenerate
```

**genvar** is a datatype used for the generate block indexing (virtual) It is not part of the circuit.

**generate** syntax enables usage of **loops**, **if-else**, **case**

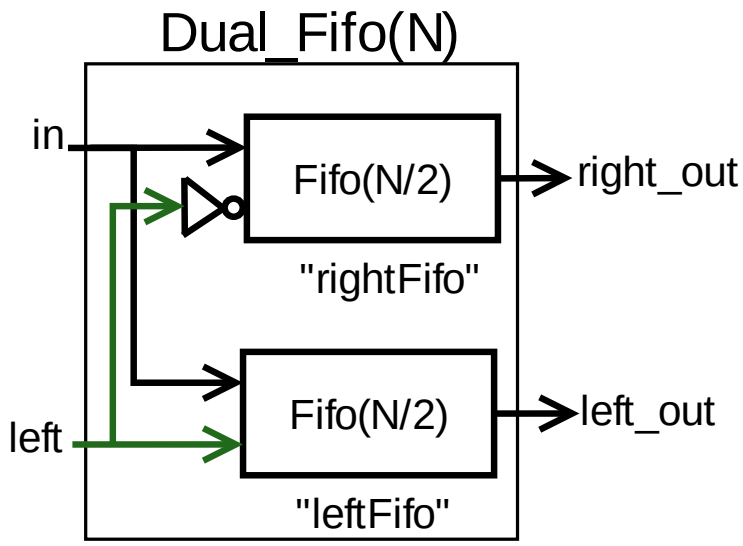


# Static elaboration

- The following features of Verilog are “virtual” and are used only during static elaboration:
  1. Parameters
  2. Generate blocks
  3. Genvars

# Conditional Hierarchy Construction

```
module Dual_Fifo(in, left, left_out, right_out);
     parameter N=4;
     input in;
     input left;
     output left_out;
     output right_out;
     generate
       if (N == 1) begin
          Fifo #(1) rightFifo(in, ~left, right_out);
          Fifo #(1) leftFifo(in, left, left_out);
       end else begin
          Fifo #(N/2) rightFifo(in, ~left, right_out);
          Fifo #(N/2) leftFifo(in, left, left_out);
       end
     endgenerate
endmodule
```




Pay attention:

- Only **`generate`** allows a conditioned instantiation (i.e. if-else, loops, cases).
- `in, left, left_out, right_out` are **`wire`**s by default, act as internal signals of the Dual_Fifo
- More **`wire`**s/**`reg`**s can be defined for internal usage within the Dual_Fifo module.
- Specifying a wire in multiple terminals – effectively short-circuits these terminals (see the `in` wire)

# Combinational logic

- Continuous assignment

```
assign result_w = (select) ? b : a;
```

Every destination of continuous assignment must be declared as **wire**

- Behavioral always@ block

```
always @(a or b or select)
begin
    if (select == 1)
        result_r = b;
    else
        result_r = a;
end
```

**Sensitivity list:** the always@ block will be triggered only when these signals change

In combinational logic "always @" statements, you must not leave if statements without else as well as you must not leave switch statements without a default. Otherwise, a **latch will be inferred** in order to keep the previous value of "result". An inferred latch is a common design mistake.

Every destination of always block must be declared as **reg**, however it won't be necessarily synthesized into a register hardware.

# Combinational logic summary

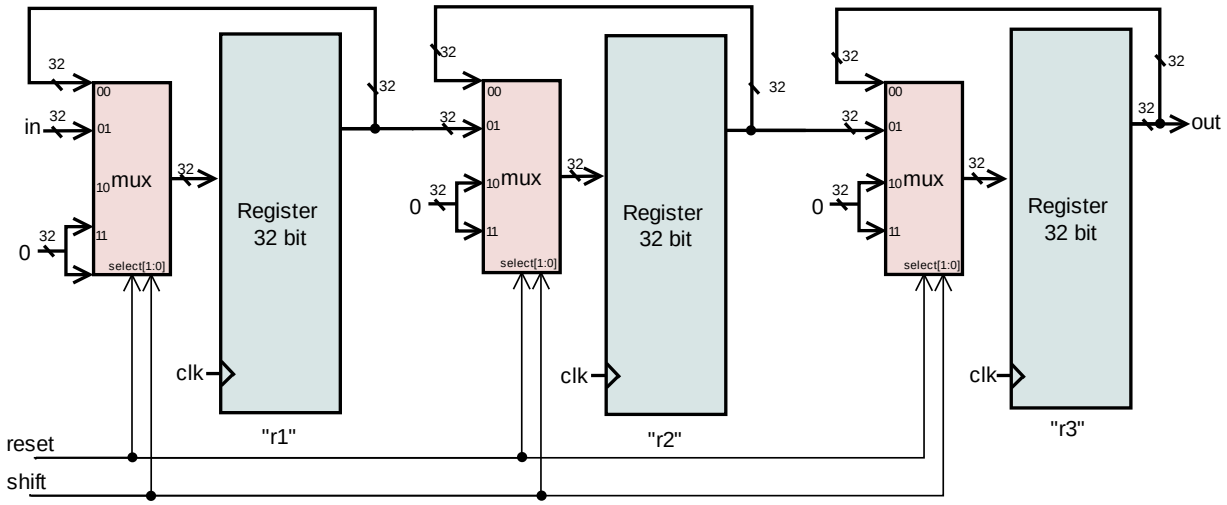
- **`assign`** (single liner) describes signal assignments to a **`wire`** element
- Tip: use **`assign w1=`**“some important Boolean expression” and then use w1 as a predicate in other places in your code.
- **`always@`** (block) describes signal assignments to **`reg`** element(s)
- Don’t write **`assign`** inside an **`always@`** block, only outside the block
- In combinational logic:
  - never leave an **`if`** without **`else`**
  - never leave a **`case`** without **`default`**

# Synchronous Logic Example: 32 shift registers

```
input reset, shift, clk;
input [31:0] in;
output [31:0] out;
reg [31:0] r1;
reg [31:0] r2;
reg [31:0] r3;

always@(posedge clk) begin
        if (reset) begin
                r1 <= 0; r2 <= 32'b0; r3 <= 32'h0;
         end else if (shift)
                begin
                        r1 <= in;
                        r2 <= r1;
                        r3 <= r2;
                end
end
assign out = r3;
```

Important to notice:

- **posedge** is a saved word
- **Reset** is the only way to initialize the registers!
- Make sure to have all registers initialized!
- Use **<=** for proper flip flop assignment.
- All **<=** assignments are parallel, although written sequentially



# Synchronous Logic – Blocking vs Non Blocking

```
module blocking (clk,a,c);
        input clk,a;
        output c;
        reg c,b;

        always @ (posedge clk)
        begin
                b = a;
                c = b;
        end

endmodule
```

Simple Thumb rule:

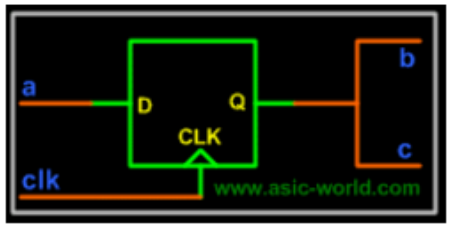
- In combinational-logic
  always@(*)
  use =

- In synchronous-logic
  always @(posedge clk)
  use <=

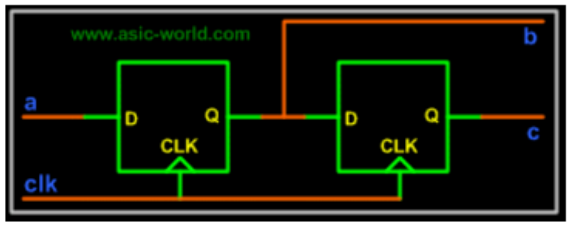
sequential "execution" of code

parallel "execution" of code

```
module nonblocking (clk,a,c);
        input clk,a;
        output c;
        reg c,b;

        always @ (posedge clk)
        begin
                b <= a;
                c <= b;
        end

endmodule
```



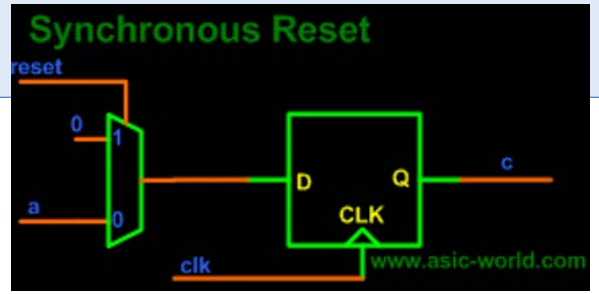

Synthesized products



# Synchronous Logic – Reset

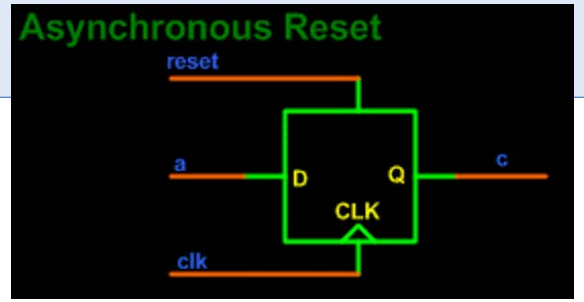
- Synchronous

```
always @(posedge clk)
begin
if(reset == 1)
        begin
        //initializations
        end
else
        begin
        //actual stuff
        end
end
```



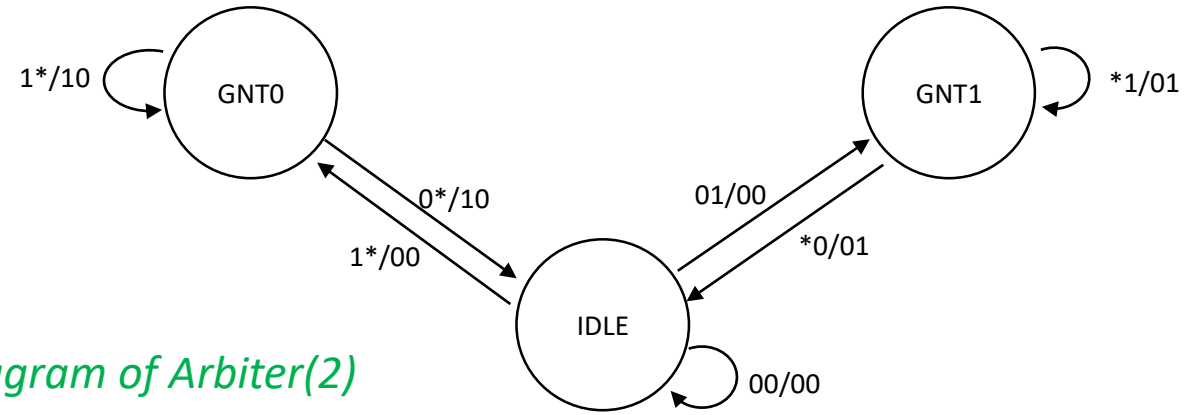

- Asynchronous

```
always @(posedge clk, posedge reset)
begin
if(reset == 1)
        begin
        //initializations
        end
else
        begin
        //actual stuff
        end
end
```

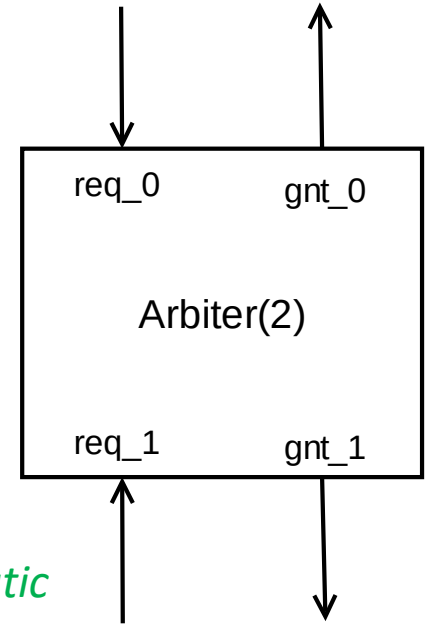


# Synchronous Logic – State Machines

- Example: **Arbiter(N)** is a circuit that helps resolving multiple requests to a shared resource. Arbiter(2) of 2 requestors can be implemented as a state machine with 2 inputs and 2 outputs, with a following functionality:
  - When req_0 is asserted, gnt_0 is outputted
  - When req_1 is asserted, gnt_1 is outputted
  - When both req_0 and req_1 are asserted then gnt_0 is asserted.
    i.e. higher priority is given to req_0.
  - After one of the grant signals was asserted, its owner needs to go low to allow next arbitration.



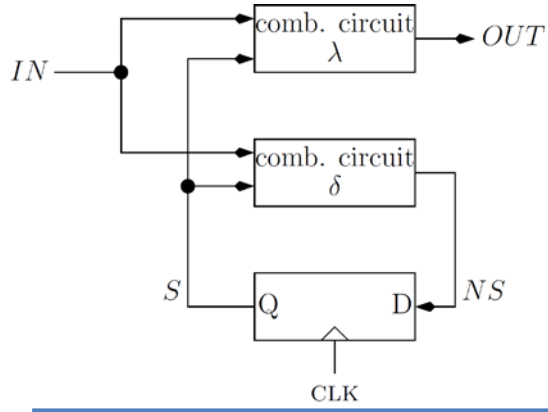

*State diagram of Arbiter(2)*



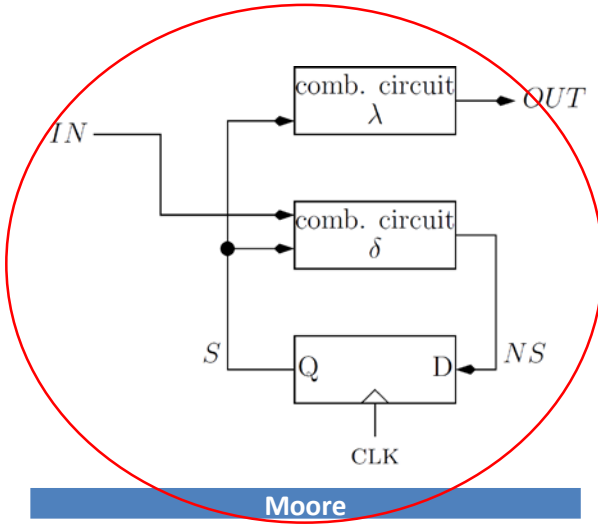

*Arbiter(2) schematic*

# Synchronous Logic – State Machines

```
module Arbiter (clk,reset,req_0,req_1,gnt_0,gnt_1);
                input   clk,reset,req_0,req_1;
                output  gnt_0,gnt_1;

                localparam SIZE = 3;
                localparam IDLE  = 3'b001,GNT0 = 3'b010,GNT1 = 3'b100 ;
                reg   [SIZE-1:0] state;// Internal state

                //------------ Transition function ----------//
                always @ (posedge clk)
                begin
                                if (reset == 1'b1)
                                                state <=  IDLE;
                                else
                                                case(state)
                                                                IDLE :          if (req_0 == 1'b1)
                                                                                                state <= GNT0;
                                                                                else if (req_1 == 1'b1)
                                                                                                state <= GNT1;
                                                                                else
                                                                                                state <= IDLE;

                                                                GNT0 :          if (req_0 == 1'b1)
                                                                                                state <= GNT0;
                                                                                else
                                                                                                state <= IDLE;
                                                                GNT1 :          if (req_1 == 1'b1)
                                                                                                state <= GNT1;
                                                                                else
                                                                                                state <= IDLE;
                                                                default :       state <= IDLE;
                                                endcase
                end

                //------------ Output function ----------//
                assign gnt_0 = (state == GNT0) ? 1 : 0;
                assign gnt_1 = (state == GNT1) ? 1 : 0;

endmodule
```




Mealey




Moore

# Test Bench for Combinational Circuits

```
module FA_tb();
reg a, b, ci, correct, loop_skipped;
wire sum, co;
integer ai, bi, cii;
FA uut (a, b, ci, sum, co);
initial begin
correct = 1; loop_skipped = 1;
#1 for( ai=0; ai<=1; ai=ai+1 ) begin
  for( bi=0; bi<=1; bi=bi+1 ) begin
    for( cii=0; cii<=1; cii=cii+1 ) begin
      loop_skipped = 0;
      a = ai[0]; b = bi[0]; ci = cii[0];
      #5 correct = correct & (a + b + ci == {co,sum});
    end
  end
end
if (correct && ~loop_skipped)
  $display("Test Passed - %m");
else
  $display("Test Failed - %m");
$finish;
end
endmodule
```

Delay notation must be either followed by a Verilog command or terminated by a semicolon ;

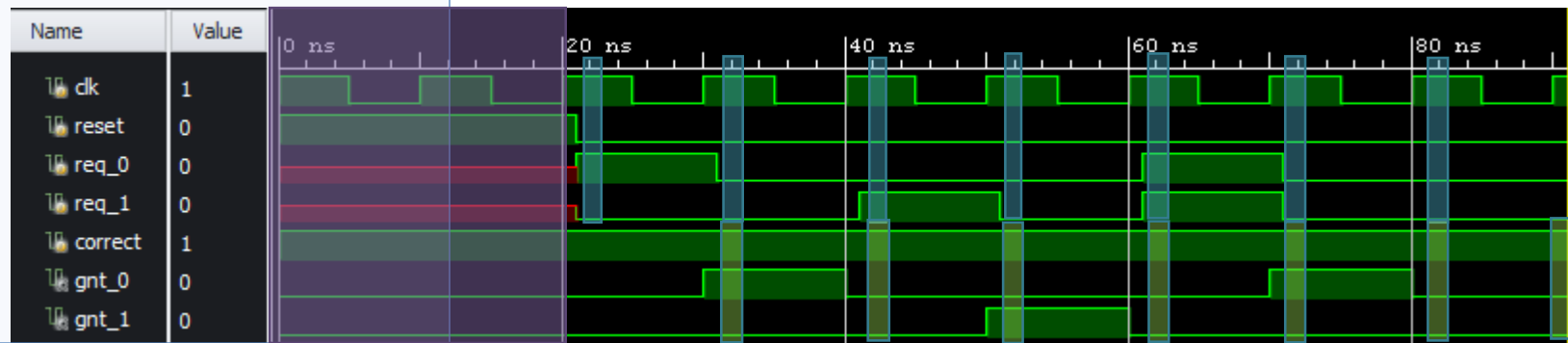
- Integer data types, for loops, $display and other "$" functions are virtual creatures (not a part of the hardware).
- UUT/DUT is a generic name for Unit/Design Under Test instance
- `initial` - Same as always@ block, but performed only once
- `initial` - Could be thought of as a main() function
- `initial` - Useful in test benches
- Check the correctness against expected values. Don't just look at the waveform diagrams and say "seems fine to me".
- #5 creates a delay of 5 time units
- Print the test summary!

# Test Bench for Synchronous Circuits

```
module Arbiter_tb();
reg clk, reset, req_0, req_1, correct;
wire gnt_0, gnt_1;
Arbiter uut (clk, reset, req_0, req_1, gnt_0, gnt_1);
initial begin
clk = 1; reset = 1; correct = 1; #1;
#20 reset = 0; req_0 = 1; req _1 = 0;
#10 correct = correct & (gnt_0 == 1 && gnt_1 == 0);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
    req_0 = 0; req_1 = 1;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 1);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
    req_0 = 1; req_1 = 1;
#10 correct = correct & (gnt_0 == 1 && gnt_1 == 0);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
if (correct==1)
  $display("Test Passed");
else
  $display("Test Failed");
$finish;
end
always #5 clk = ~clk;
endmodule
```

Tips:

- Reset for a couple of cycles and create a 1ns shift after posedge
- Force test-inputs during the clock cycle for the to be sampled in the following posedge.
- Check the outputs of the current clock cycle



# Timescale

- **`timescale 1 ns / 1 ps**
- In this example:
  - 1ns is the basic time unit when using # (delay) notation. #10 will be equal to 10ns.
  - 1ps is the resolution, which means that #(1.001) is the maximal precision you can use in the delay notation

# Functions and Tasks

Notes:

- Use tasks and functions to write a particular code once and invoke it in many locations.
- Don't forget to specify input/output bit-width

- **Function** cannot return more than one output, cannot contain delays, or call tasks. Can call other functions.
- Define

```
function myfunction;
      input a, b, c, d;
      begin
         myfunction = ((a+b) + (c-d));
      end
endfunction
```

- Call

```
 W1 = myfunction (a,b,c,d);
```

- **Task** can drive many outputs, can call other functions and tasks
- Define

```
task convert;
   input [7:0] temp_in;
   output [7:0] temp_out;
   begin
     temp_out = (9/5) *( temp_in + 32);
   end
endtask
```

- Call

```
  convert (temp_a, temp_b);
```

# Additional Important Lessons

- Apply Logic on data lines only. CLOCK or RESET gating must be done carefully to avoid glitches!
- Use `#10` (delays) and `initial` blocks only in test-benches! Not in the designs
- `reg` for output is a good practice - cuts the critical path, helps timing analysis. (Otherwise, the module fed by your module will extend the critical path)
- Every synchronous always block – must first reset ALL the registers if the reset is active. Otherwise: do all the logic.
- Don’t use `initial` blocks for initialization

# Running NCVERILOG

- `ncverilog +access+rw *.v`
  - Runs all the Verilog sources in the current directory. Performs the test benches.
- `simvision waves.vcd`
  - Opens a GUI, then choose signals and drag them right to the timeline view
- `/bin/tcsh`
  - If your shell doesn't recognize ncverilog