Digital System Design

Hajar Falahati

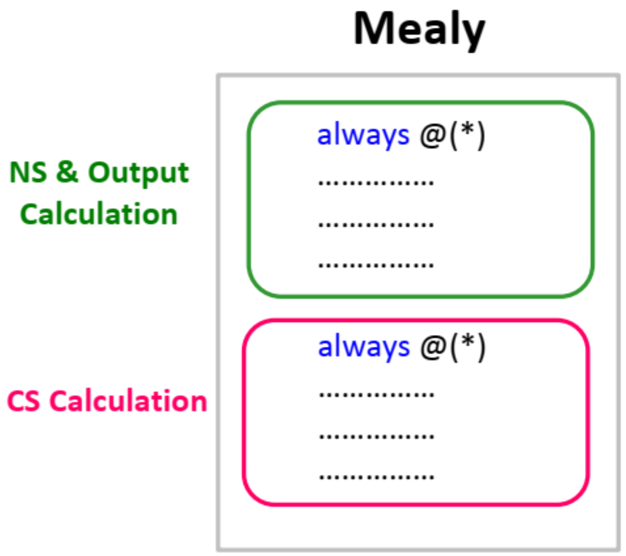
hfalahati@ipm.ir
hfalahati@ce.sharif.edu

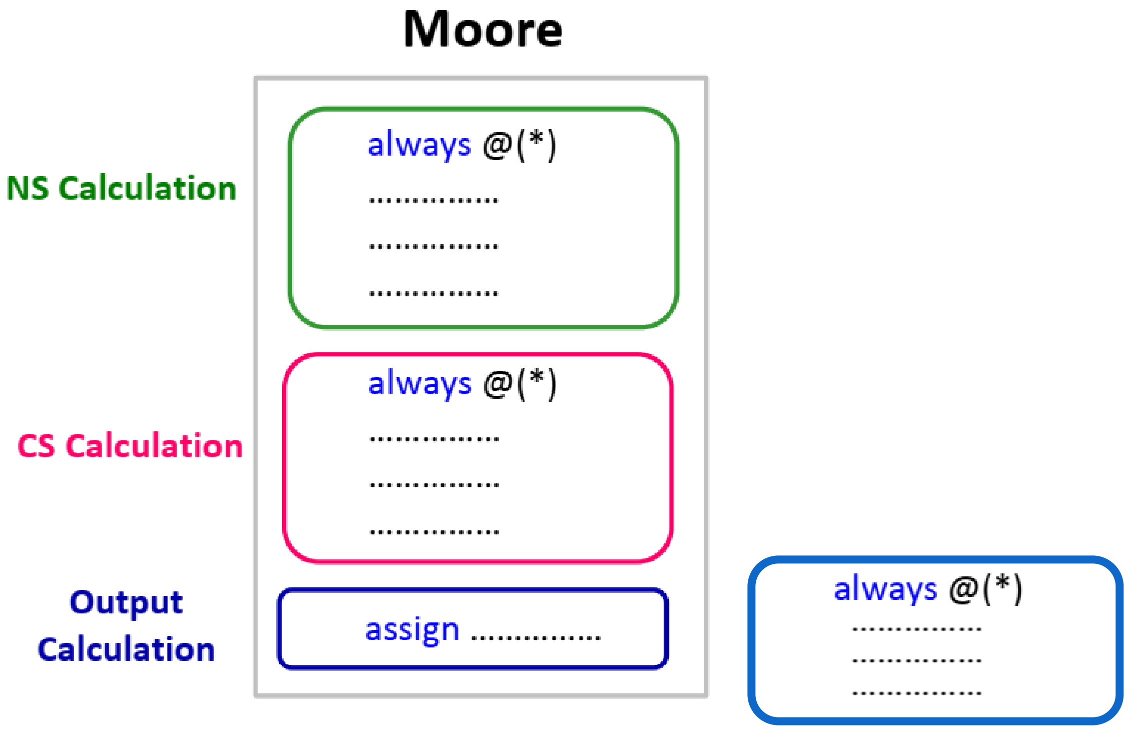
Combinational Logic & Verilog

- Combinational Logic
  - Use **always block** + **"blocking" assignments**
    - Normally for high-complexity Comb. Logic
    - When output depends on several conditions, which requires if-else
- Sequential Logic
  - Can **only** be realized using an **always block**
  - When using the **always block** for the sequential Logic, **"Non-blocking" assignments** are used

FSM Code Structure

- Mealy
  - Output depends on input
  - Output declared as reg
- Moore
  - Output does not depends on input
  - Output declared as wire





Outline

- Function
- Task

Function

Function

- Design hierarchy
  - Hierarchical name
- Modular coding
  - May be used to have a modular code without defining separate modules

```
function my4-to-1MUX;
    input [0:3] W;
    input [1:0] s;
    if (s==0) my4-to-1MUX = W[0];
    else if (s==1) my4-to-1MUX = W[1];
    else if (s==2) my4-to-1MUX = W[2];
    else if (s==3) my4-to-1MUX = W[3];
endfunction
```

More About Function

- Do we must use functions?
  - No crucial
  - Might **facilitate** modular coding
- Where to define?
  - Defined inside a module
  - Local to the module
- Which assignments?
  - Can be called both in continuous and procedural assignments

Function: Input & output

- Input argument
  - At least one input argument

- Output argument
  - Does **not** have **any output**
  - Function name **serves as the output**
  - Returns **exactly one single value**
  - Returned value is placed wherever the function was invoked

How to Declare a Function?

```
function <range_or_type> <function_name>;
    input <input argument (s)>;   // at least one input

   <variable_declaration (s) >;  // optional

     begin    // if more than one statement needed
        <statements>
     end

endfuction
```

```
range_or_type ::= range | integer | real | realtime | time
```

Inside a Function

- Keyword
  - function
  - endfunction
- Have local variables
  - Registers
  - Time variables
  - Integers
  - Real
  - Events
- No wires
- Contain behavioral statements only
- No always or initial statements

```
function <range_or_type> <function_name>;
     input <input argument (s)>;   // at least one input

    <variable_declaration (s) >;  // optional

      begin    // if more than one statement needed
          <statements>
      end

endfuction
```

```
range_or_type ::= range | integer | real | realtime | time
```

When Do We Use Function

- Verilog code is purely combinational
- Executes in zero simulation time
  - I.e., no timing control constructs
  - I.e., no delay
- Provides exactly one output
- No event

```
<function_name>  ( <arguments (s) > ) ;
```

Sample 1 : Not

```
function  my_not;
     input in;

      my_not = ~in;

endfuction
```

```
reg ni;

initial
     ni = my_not(i);
```

Sample 2

- Please implement a 16-to-1 multiplexer
  - Using 4-1 MUX
  - Using function

Sample 2: 16-to-1 MUX

```
module my16-to-1MUX (W, S, Out);
   input  [0:15] W;
   input [3:0] S;
   output  reg Out;
   reg  [0:3] M;

   function my4-to-1MUX;
       input [0:3] W;
       input [1:0] s;
       if (s==0) my4-to-1MUX = W[0];
       else if (s==1) my4-to-1MUX = W[1];
       else if (s==2) my4-to-1MUX = W[2];
       else if (s==3) my4-to-1MUX = W[3];
   endfunction

   always@ (W, S)
      begin
         M[0] = my4-to-1MUX(W[0:3],S[1:0]);
         M[1] = my4-to-1MUX(W[4:7],S[1:0]);
         M[2] = my4-to-1MUX(W[8:11],S[1:0]);
         M[3] = my4-to-1MUX(W[12:15],S[1:0]);

         Out   = my4-to-1MUX(M[0:3], S[3:2]);

      end
endmodule
```

```
if      (S[3:2]==0) Out= M[0];
else if (S[3:2]==1) Out= M[1];
else if (S[3:2]==2) Out= M[2];
else if (S[3:2]==3) Out= M[3];
```

Sample 3: Multi-bit Output

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output reg [2:0] Out;

function [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

always @(*)
     Out = myfcn(a,b,c);

endmodule
```

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output [2:0] Out;

function [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

assign   Out = myfcn(a,b,c);

endmodule
```

Sample 4: Signed Output

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output reg [2:0] Out;

function signed [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

always @(*)
     Out = myfcn(a,b,c);

endmodule
```

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output [2:0] Out;

function  signed [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

assign   Out = myfcn(a,b,c);

endmodule
```

Function using ANSI C Style

```
function  [1:0] compute_next_state (input [1:0] state, input i);
                  if (state == 0)     compute_next_state = i ? 1 : 0;
                 else if (state == 1) compute_next_state = i ? 2 : 0;
                 else if (state == 2) compute_next_state = i ? 2 : 0;
                 else compute_next_state = 0;

                endfuction
```

Sample 5

- Left/right shifter
  - Shifts a 32-bit value to the left or right by one bit, based on a control signal

Sample 5: Shifter

```
module  shifter (addr, left_addr, right_addr);
        `define LEFT_SHIFT       1'b0
        `define RIGHT_SHIFT    1'b1

         input [31:0] addr
         output reg  [31:0] left_addr, right_addr;

         function  [31:0] shift (input [31:0] address, input control);
                   shift = control ? address << 1 : address >>1;
         endfuction

         always @ (addr)
         begin
                 left_addr = shift (addr, `LEFT_SHIFT);
                 right_addr = shift (addr, `RIGHT_SHIFT);
         end
endmodule
```

Check Yourself 1

- Parity Generator
  - Calculates the parity of a 32-bit address and returns the value.

Question 1 (cont'd)

```
//define the parity calculation function using ANSI C Style arguments
function calc_parity (input [31:0] address);
begin
        //set the output value appropriately. Use the implicit
        //internal register calc_parity.
        calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
```

Question 1 (cont'd)

```
//Define a module that contains the function calc_parity
module parity;
...
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @(addr)
begin
        parity = calc_parity(addr); //First invocation of calc_parity
        $display("Parity calculated = %b", calc_parity(addr) );
                                    //Second invocation of calc_parity
end
...
...
endmodule
```

Question 1 (cont'd)

```
module parity;
...
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @(addr)
begin
        parity = calc_parity(addr); //First invocation of calc_parity
        $display("Parity calculated = %b", calc_parity(addr) );
                                    //Second invocation of calc_parity
end
...
...
//define the parity calculation function
function calc_parity;
input [31:0] address;
begin
        //set the output value appropriately. Use the implicit
        //internal register calc_parity.
        calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
...
...
endmodule
```

How about Recursive Function

- Functions are normally used non-recursively
- If a function is called concurrently from two locations
  - Results are non-deterministic
  - Both calls operate on the same variable space

Automatic (Recursive) Function (cont'd)

- Declare a recursive (automatic) function
- All function declarations allocated dynamically for each recursive calls
  - Each call to an automatic function operates in an independent variable space
- Keyword
  - **automatic**

Sample 6

- Define a factorial function

Sample 6: Factorial

```
function automatic integer factorial;
input [31:0] oper;
integer i;
begin
if (operand >= 2)
   factorial = factorial (oper -1) * oper; //recursive call
else
   factorial = 1 ;
end
endfunction
```

```
// Call the function
integer result;
initial
begin
     result = factorial(4);
     $display("Factorial of 4 is %0d", result); //Displays 24
end
...
...
endmodule
```

Tasks

Task

- A task can **only** be called from inside **always (or initial) block**
- A task can have **multiple inputs and outputs**
- Tasks are used when Verilog code contains
  - Delay
  - Timing
  - Event
  - Zero or multiple input/output arguments

Inside a Task

- Keyword
  - task
  - endtask
- Have local variables
  - Registers
  - Time variables
  - Integers
  - Real
  - Events
- No wires
- Contain behavioral statements only
- No always or initial statements

```
task <range_or_type> <task_name>;

   <I/O decleration(s)>;   // optional

    <variable and event declaration(s) >;

      begin    // if more than one statement needed
         <statement(s)>
      end

endtask
```

```
<task_name>;
<task_name> (<arguments>);
```

Sample 7: Task Invocation

- Use of input and output arguments

```
module operation;
...
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) //whenever A or B changes in value
begin
        //invoke the task bitwise_oper. provide 2 input arguments A, B
        //Expect 3 output arguments AB_AND, AB_OR, AB_XOR
       //The arguments must be specified in the same order as they
       //appear in the task declaration.
        bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
...
...
//define task bitwise_oper
...
endmodule
```

Sample 7: Task Declaration

```
//define task bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //outputs from the task
input [15:0] a, b; //inputs to the task
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
```

```
//define task bitwise_oper
task bitwise_oper (output [15:0] ab_and, ab_or, ab_xor,
                    input [15:0] a, b);
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
```

Sample 8: Invocation Two Tasks

```
module sequence;
...
reg clock;
...
initial
        init_sequence; //Invoke the task init_sequence
...
always
begin
        asymmetric_sequence; //Invoke the task asymmetric_sequence
end
...
...
//Initialization sequence
task init_sequence;

 //define task to generate asymmetric sequence
 //operate directly on the clock defined in the module.
 task asymmetric_sequence;
 ...
 ...
 endmodule
```

Sample 8: Declaration Two Tasks

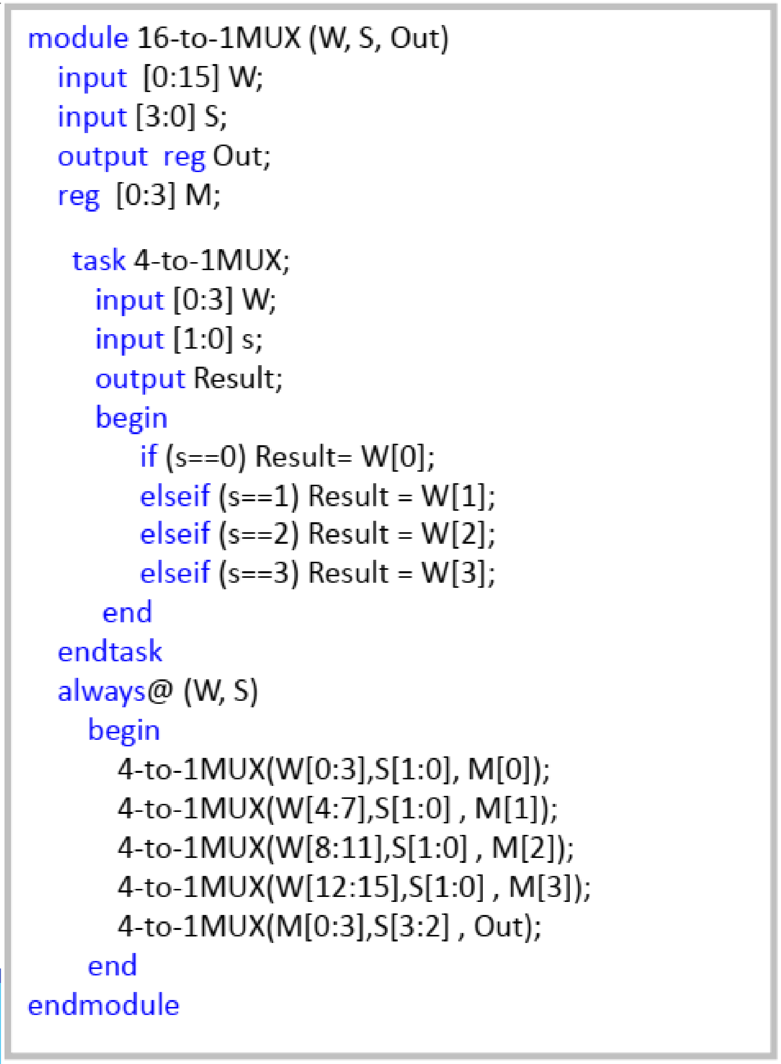
```
//Initialization sequence
task init_sequence;
begin
        clock = 1'b0;
end
endtask
```

```
//define task to generate asymmetric sequence
//operate directly on the clock defined in the module.
task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5 clock = 1'b1;
        #3 clock = 1'b0;
        #10 clock = 1'b1;
end
endtask
```

Sample 9

- Implement a 16-1 MUX

Sample 9: 16-1 MUX



```
module 16-to-1MUX (W, S, Out)
   input  [0:15] W;
   input [3:0] S;
   output  reg Out;
   reg  [0:3] M;

    task 4-to-1MUX;
       input [0:3] W;
       input [1:0] s;
       output Result;
       begin
           if (s==0) Result= W[0];
           elseif (s==1) Result = W[1];
           elseif (s==2) Result = W[2];
           elseif (s==3) Result = W[3];
        end
   endtask
   always@ (W, S)
      begin
         4-to-1MUX(W[0:3],S[1:0], M[0]);
         4-to-1MUX(W[4:7],S[1:0] , M[1]);
         4-to-1MUX(W[8:11],S[1:0] , M[2]);
         4-to-1MUX(W[12:15],S[1:0] , M[3]);
         4-to-1MUX(M[0:3],S[3:2] , Out);
      end
endmodule
```

Automatic Tasks ?

- Tasks are normally static in nature
- All declared items are statically allocated
- They are shared across all uses of the task executing concurrently
- If a task is called concurrently from two locations
  - These task calls will operate on the same task variables
  - Results will be incorrect

Automatic (Recursive) Function (cont'd)

- Declare an automatic task
- All declarations allocated dynamically for each invocation
- Each task call operates in an independent space
- Keyword
  - **automatic**

Sample 9

```
module top;
reg [15:0] cd_xor, ef_xor; //variables in module top
reg [15:0] c, d, e, f; //variables in module top
-
task automatic bitwise_xor;
output [15:0] ab_xor; //output from the task
input [15:0] a, b; //inputs to the task
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
...
-
// These two always blocks will call the bitwise_xor task
// concurrently at each positive edge of clk. However, since
// the task is re-entrant, these concurrent calls will work correctly.
always @(posedge clk)
    bitwise_xor(ef_xor, e, f);
-
always @(posedge clk2) // twice the frequency as the previous block
    bitwise_xor(cd_xor, c, d);
-
-
endmodule
```

Function vs. Task

Functions	**Tasks**
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
unctions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

Check Yourself ☺

- Define a task to compute the factorial of a 4-bit number.
- The output is a 32-bit value.

Thank You