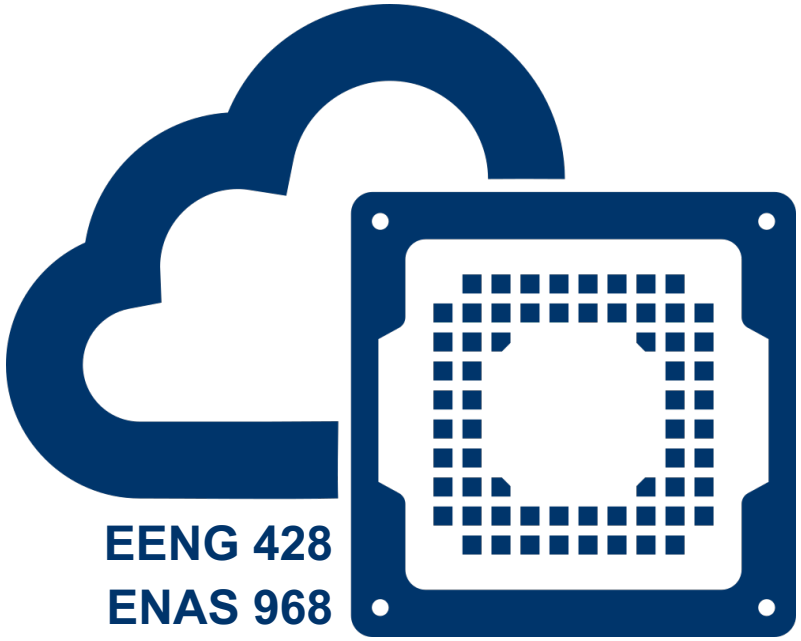
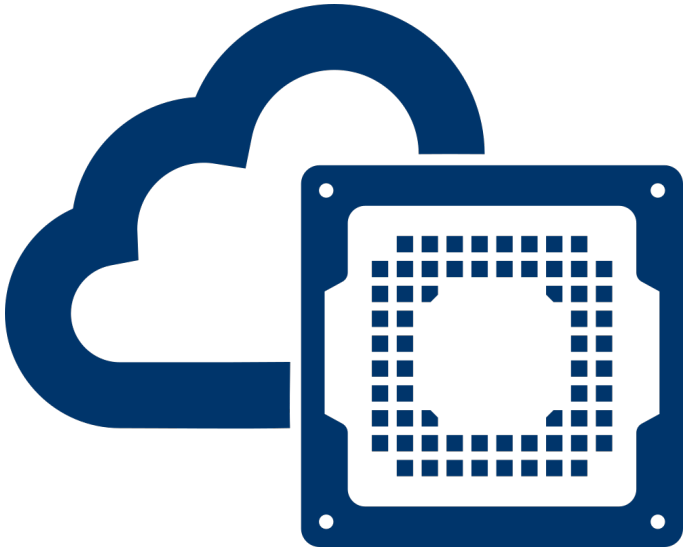
Cloud FPGA



bit.ly/cloudfpga



Lecture: Behavioral Modeling

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA

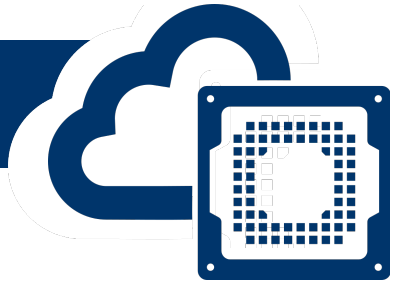
Logic Synthesis

This lecture is mostly based on contents of Chapter 3, from "The Verilog Hardware Description Language" book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

Topics covered:

- Blocking and non-blocking assignments
- Behavioral modeling with processes
- If-then-else, if-else-if, case statements
- Functions and tasks
- Structural view
- Rules of scope and hierarchical names

Blocking and Non-Blocking Assignments



Foreshadowing of chapter 8:

- The **<=** operator is allowed anywhere the **=** is allowed in procedural assignment statements
- The non-blocking assignment operator cannot be used in a continuous assignment statement
  - Don't use in **`assign`** statements
- Don't confuse with less-than-equal **<=**
  - Going left-to-right in an expression, first **<=** is assignment, others are comparisons

- Non-blocking behavior, the **<=** does not block the process:

```
a <= b
c <= a // previous a <= b did not block process,
       // so c <= a uses value of a before a became b
```

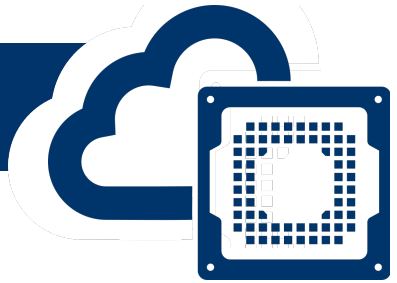
Style-guide:

- Use non-blocking in sequential logic, **`always @ (posedge clock)`**
- All others (combinatorial logic, functions, tasks) use blocking

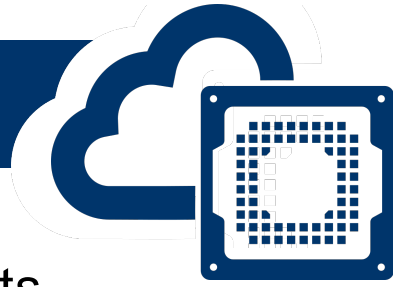
Using = here can possibly lead to extra storage elements being synthesized, affecting timing of circuit

Can use <= here, just be careful of the resulting logic

Behavioral Modeling and Processes



Processes

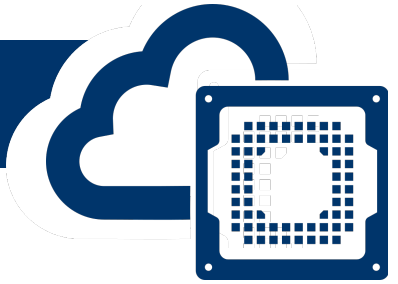


- A process is described in Verilog using **`always`** statements and **`initial`** statements
  - The **`always`** process continuously repeats itself
  - The **`initial`** statement only runs once (at start of simulation, for example)
  - There can be many such statements in a module, which logically execute concurrently
    - A module can also have none, in which case it only described structure of logic (e.g. connections between modules)

- The **`initial`** statements cannot be synthesized into hardware
  - They are used for simulation to initialize values
  - In synthesized hardware typically a 'reset' signal is used to initialize values
    - For FPGAs, the tools usually let you set initial value for a register
    - For ASIC need to explicitly reset all registers when system starts

```
// Initialize value with initial statement
reg myVal;

initial
begin
    myVal = 1;
end

// Initialize value when defining a register
reg myVal = 1;

// Initialize value using reset signal
always @(posedge clock, posedge reset)
begin
    if (reset)
        myVal <= 1;
    else
        myVal <= ...;
end
```

Execution Model of `always` and `initial`



- Within each process, the statements are evaluated serially, similar to a set of C instructions
  - When using blocking = values are assigned immediately and can be used in next statement
  - When using non-blocking <= the values are assigned in parallel

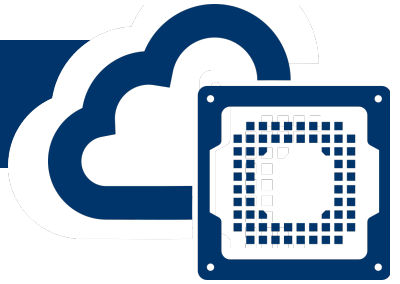
- Event statements `@`, delay statements `#`, and `wait` statements cause the evaluation of the process to be suspended until, respectively:
  - Event occurs
  - Number of time units has passed
  - Condition becomes true
- Event statements continue when condition is met

- The events, time delay, or conditions becoming true are triggers for statement evaluation to continue

Regardless, all `always` and `initial` work in parallel, there is no order of value updates

The `always` will continue to run from beginning, while the `initial` stops when it reaches the end

If-Then-Else and Other Features of Verilog



- If-then-else statements are used in processes to control the control flow
  - Like C programs, based on the conditions, do different things

```
`define DvLen 16
`define DdLen 32
`define QLen 16
`define HiDdMin 16

module divide
( input [`DdLen-1:0] ddInput, dvInput,
  output reg signed [`QLen-1:0] quotient,
  input go,
  output reg done
);

reg signed [`DdLen-1:0] dividend;
reg signed [`DvLen-1:0] divisor;
reg negDivisor, negDividend;

always begin
  done = 0;
  wait (go);
  divisor = dvInput;
  dividend = ddInput;
  quotient = 0;
  if (divisor) begin
    negDivisor = divisor[`DvLen-1];
    if (negDivisor)
      divisor = - divisor;
    negDividend = dividend[`DdLen-1];
    if (negDividend)
      dividend = - dividend;
    repeat (`DvLen) begin
      quotient = quotient << 1;
      dividend = dividend << 1;
      dividend[`DdLen-1:`HiDdMin] =
        dividend[`DdLen-1:`HiDdMin] - divisor;
      if (! dividend [`DdLen-1]
        quotient = quotient + 1;
      else
        dividend[`DdLen-1:`HiDdMin] =
          dividend[`DdLen-1:`HiDdMin] + divisor;
    end
    if (negDivisor != negDividend)
      quotient = - quotient;
  end
  done = 1;
  wait (~go);
  end

endmodule
```

Macros can be used to define useful string substitutions

All values are unsigned by default, but can define signed (2's complement)

The `wait` statement causes process evaluation to stop until condition becomes TRUE

The `go` is example of handshake signal, another module can trigger `go` to begin divide operation

Bit-select statements
- `[high : low]`
- `[value]`
- `[start+ : range] = [start+range-1 : start]`
- `[start- : range] = [start : start-range+1]`

The `repeat` is like a `for` loop that executes exactly the number of times specified by the repeat value

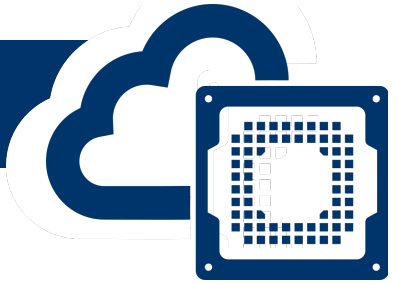
Arithmetic operators +, -, etc., will do correct computation with respect to signed / unsigned values

Same as:

```
quotient = (negDivisor != negDividend)
         ? -quotient : quotient;
```


Share: bit.ly/cloudfpga

If-Then-Else Condition Tests



- The if-then-else can test values of wires are registers using the typical operators
  - Greater then `>`, less than `<`, equal `==`, not equal `!=`
  - The unknown `x` or high-impedance `z` is considered FALSE
  - Triple equal `===` or triple not-equal `!==` consider unknown `x` or high-impedance `z`
    - `if (4'b110z === 4'b110z)` evaluates to TRUE
    - `if (4'b110z ==  4'b110z)` evaluates to FALSE
  - Triple equals cannot be synthesized to hardware, as in hardware all values are either `0` or `1` (no `x` or `z`)

- The `else` statement is associated with the closest `if` statement

```
if (expressionA)
  if (expressionB)
    a = a + b;
  else
    q = r + s;

if (expressionA)
begin
  if (expressionB)
    a = a + b;
end
else
  q = r + s;
```

Please see textbook Appendix G for formal specification of Verilog.

```
if (expressionA)
  d = e;
  f = g;
```

is same as:

```
if (expressionA)
begin
  d = e;
end
  f = g;
```

but:

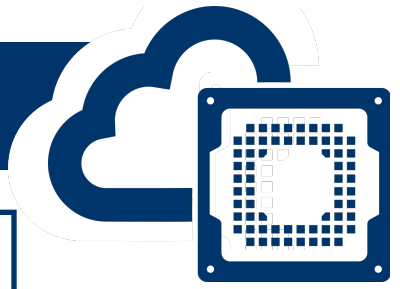
```
if (expressionA)
  if (expressionB)
    d = e;
  else
    f = g;
```

is same as:

```
if (expressionA)
begin
  if (expressionB)
    d = e;
  else
    f = g;
end
```

- Conditional tests can test conditions using typical operators
  - Operators such as AND, OR, XOR, etc. (not just Boolean algebra operators)
  - Bitwise operators compare each bit of the first operand to the corresponding bit of the second operand, logical operators treat each operand as a single value and compare the values

Loops



Four different loop statements are available in Verilog:

- **`repeat ( numTimes )`**
- **`for ( initLoopCond; testExpression; updateLoopCond )`**
- **`while ( someCondition )`**
- **`forever`**

Expressions or test conditions for the loop statements can use parameters, local parameters, or macros

Condition needs to be updated in loop body, can't use external conditions, e.g. module inputs

Use `integer` values or registers with extra bits to avoid wrap-around when updating test condition

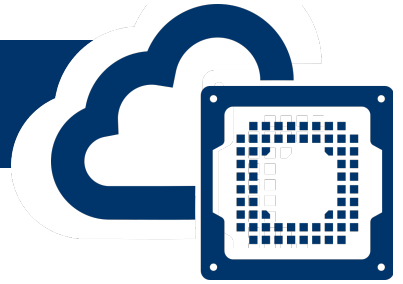
The loops will not be actually created in hardware, they are just used to describe the behavior of the system or module

- Will generate simple logic if each loop condition is independent and can execute in parallel
- May generate very complex logic if there are interdependencies between each loop iteration

Loops can be exited early using **`disable`** statement

- Similar to **`break`** statement in C

Multi-Way Branching



- **If-else-if** statements and case selection statements allow for multi-way branching
- Example of code to emulate simplified Mark-1 processor

```
module mark1;
  reg [15:0] signed m [0:8191]; // signed 8192 x 16 bit memory
  reg [12:0] signed pc; // signed 13 bit program counter
  reg [12:0] signed acc; // signed 13 bit accumulator
  reg [15:0] ir; // 16 bit instruction register
  reg ck; // a clock signal

  always
  begin

    @(posedge ck)
      ir <= m [pc]; // fetch an instruction

    @(posedge ck)
      if (ir[15:13] == 3'b000) // begin decoding
        pc <= m [ir [12:0]]; // and executing
      else if (ir[15:13] == 3'b001)
        pc <= pc + m [ir [12:0]];
      else if (ir[15:13] == 3'b010)
        acc <= -m [ir [12:0]];
      else if (ir[15:13] == 3'b011)
        m [ir [12:0]] <= acc;
      else if ((ir[15:13] == 3'b101) || (ir[15:13] == 3'b100))
        acc <= acc - m [ir [12:0]];
      else if (ir[15:13] == 3'b110)
        if (acc < 0)
          pc <= pc + 1;

    pc <= pc + 1; //increment program counter

  end

endmodule
```

Multiple event statements

Using posedge will create sequential logic

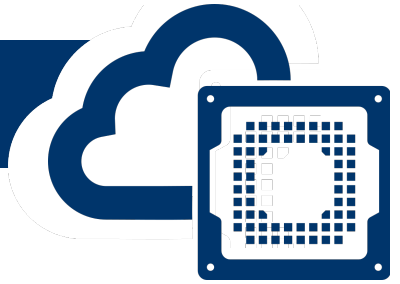
Combinatorial logic

Multiple control flows based on different if-else-if tests

The ASCC (Mark-1) was built from switches, relays, rotating shafts, and clutches. It used 765,000 electromechanical components and hundreds of miles of wire in 1944.

The first transistor was invented in 1947.

Multi-Way Branching



- If-else-if statements and **case selection** statements allow for multi-way branching
- Example of code to emulate simplified Mark-1 processor

```
module mark1Case;
  reg [15:0] signed m [0:8191]; // signed 8192 x 16 bit memory
  reg [12:0] signed pc; // signed 13 bit program counter
  reg [12:0] signed acc; // signed 13 bit accumulator
  reg [15:0] ir; // 16 bit instruction register
  reg ck; // a clock signal

  always
  begin

    @(posedge ck)
      ir <= m [pc];

    @(posedge ck)
      case (ir [15:13])
        3'b000 : pc <= m [ir [12:0]];
        3'b001 : pc <= pc + m [ir [12:0]];
        3'b010 : acc <= -m [ir [12:0]];
        3'b011 : m [ir [12:0]] <= acc;
        3'b100,
        3'b101 : acc <= acc - m [ir [12:0]];
        3'b110 : if (acc < 0) pc <= pc + 1;
      endcase

    pc <= pc + 1;

end

endmodule
```

Conditions are checked in order, similar to if-else-if

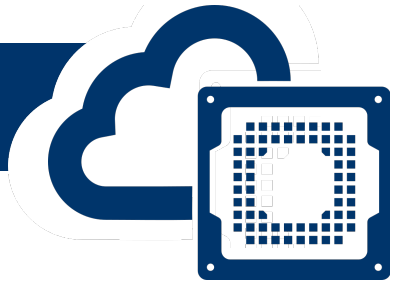
Multiple cases can have save operation

Already generation sequential logic (because of `posedge`), but it is good style to still list default case

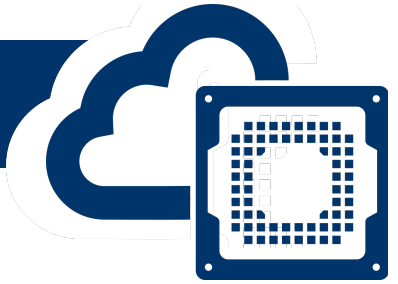
Different case statements exist:
- `CASE` specify don't care as `?` on left-hand side
- `CASEX` allows for both `z` and `x` values to be treated as don't cares when doing comparison
- `CASEZ` allows for `z` values to be treated as don't cares when doing comparisons

For logic to synthesize to hardware, there can be no don't cares `x` on right-hand side of assignments

Functions and Tasks



Functions and Tasks



- Verilog provides functions and tasks as primitives similar to software functions
- They allow for the behavioral description of a module to be broken down into even more-manageable parts
  1. First, break design into module – hierarchical design
  2. Second, use functions, tasks, macros, etc. in module – further break down the complexity

- Functions and tasks can be written for often-used behavioral sequences, write the description once and then re-use many times

- Functions are simpler (less options) and can be used for synthesizing hardware
- Task are more complex, and mainly used for simulation
  - Can't synthesize to hardware, e.g., when delay is used in a task

Comparison of Functions and Tasks

- Comparison table from the textbook [1] shows the different features of functions and tasks

Category	Tasks	Functions
Enabling (calling)	A task call is a separate procedural statement. It cannot be called from a continuous assignment statement.	A function call is an operand in an expression. It is called from within the expression and returns a value used in the expression. Functions may be called from within procedural and continuous assignment statements.
Inputs and outputs	A task can have zero or more arguments of any type.	A function has at least one input. It does not have inouts or outputs. However, a value is returned.
Timing and event controls (#, @, and wait)	A task can contain timing and event control statements. Thus it can be concurrently active if called from concurrent always/initial blocks.	Functions may not contain these statements. They are not re-entrant.
Enabling (calling) other tasks and functions	A task may enable other tasks and functions	A function can enable other functions but not other tasks.
Storage	Storage of the inputs, outputs, and internally declared variables is static — concurrent calls share the storage. However, if the task is declared automatic, then the storage is dynamic and each call gets its own copy.	Storage of the inputs and internally declared variables is static. If the function is declared automatic, then the storage is dynamic and recursive calls get their own copies.
Values returned	A task does not return a value to an expression. However, values written by the task into its inout or output ports are copied back at the end of the task execution.	A function returns a single value to the expression that called it. The value to be returned is assigned to the function identifier within the function.

Not re-entrant, recursive calls to functions and tasks use same storage

Need to define as `automatic` to allow for recurrence

Non re-entrant have static storage, re-entrant have dynamic storage, may not synthesize to hardware

Tasks

- Tasks are very similar to functions, but can set multiple outputs and use timing (not shown in example)

```
// code omitted

    @(posedge ck)
      case (ir [15:13])
        // other case expressions as before
        3'b111 : multiply (acc, m [ir [12:0]]);
      endcase

// code omitted

task multiply
( inout [12:0] a,
  input [15:0] b
);

begin: serialMult
  reg [5:0] mcnd, mpy; //multiplicand and multiplier
  reg [12:0] prod; //product

  mpy = b[5:0];
  mcnd = a[5:0];
  prod = 0;

  repeat (6)
  begin
    if (mpy[0])
      prod = prod + {mcnd, 6'b000000};
    prod = prod >> 1;
    mpy = mpy >> 1;
  end

  a = prod;

end
endtask
```

Task inputs and outputs

Behavioral description of the task's operation

`inout` is both input and output, copy as input at beginning and send as out at end

Same as `acc <= acc * m [ir [12:0]];`

Functions

- Functions are simpler tasks, useful for synthesizable code

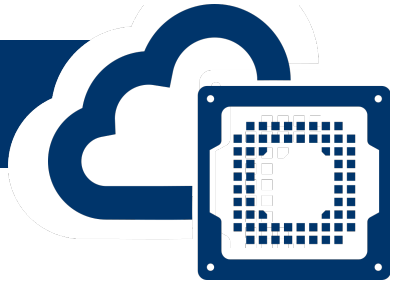
```
// code omitted

    @(posedge ck)
      case (ir [15:13])
        //case expressions, as before
        3'b111: acc <= multiply(acc, m [ir [12:0]]);
      endcase

// code omitted

function signed [12:0] multiply
( input signed [12:0] a,
  input signed [15:0] b
);

begin: serialMult
  reg [5:0] mcnd, mpy;

  mpy = b[5:0];
  mcnd = a[5:0];
  multiply = 0;

  repeat (6)
  begin
    if (mpy[0])
      multiply = multiply + {mcnd, 6'b000000};
    multiply = multiply >> 1;
    mpy = mpy >> 1;
  end

end
endfunction
```

Function inputs and outputs

Output is same as function name

Behavioral description of the function's operation

Constant Functions

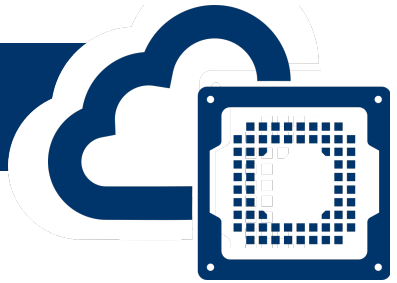


- Constant functions are just functions, but inputs come from parameters or local parameters and are not values of wires or registers

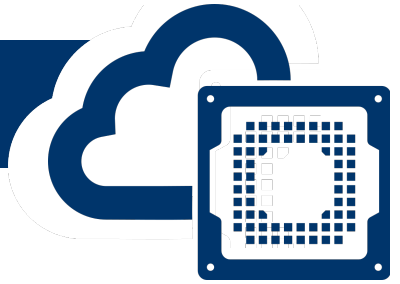
```
module RAM
# (parameter Width = 16,
             NumWords = 8192
)
( inout [Width-1:0] data,
  input [clog2b(NumWords):0] address,
  input rw, ck
);

  reg [Width-1:0] m [0:NumWords-1];

  function integer clog2b // constant function
  ( input integer size    // assumes non-zero size
  );
  begin
    for (clogb2 = -1; size > 0; clogb2 = clogb2 + 1)
      size = size >> 1;
  end
  endfunction

always ... // internal behavior of the RAM

endmodule
```

Use function to compute size of wire

Structural View, Rules of Scope, and Hierarchical Names



Structural View

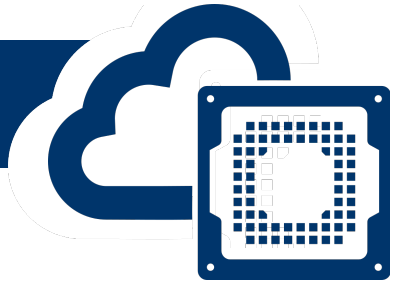


- Tasks and functions help to organize the behavioral models
- Modules help to build hierarchical designs

- All three help to design structure of the system
  - Progressively can implement more detailed design
  - Begin by using * for multiple
  - Finish by writing gate-level description of the multiplier unit

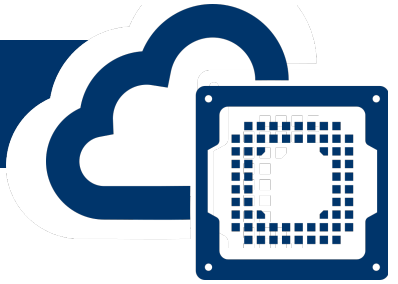
- Behavioral modeling helps to get the design started more quickly

Rules of Scope and Hierarchical Names



- Module names are known globally across the whole design in Verilog
  - Each module instance requires a unique instance name
- Identifiers for modules, tasks, functions, and named begin-end blocks are allowed to be forward referencing and thus may be used before they have been defined
- Forward referencing is not allowed with register and net accesses
  - If you forget to declare a variable, or declare variable after it is used, synthesis tools may automatically declare it as a wires – leading to errors about double declaration or conflicts
- Each entity in the design can be accessed through hierarchy of names
  - Top entity is usually top
  - Use dot . to specify hierarchy, e.g., **`top.abc.xyz`** may mean module **`xyz`** inside **`abc`** inside top

Example of Rules of Scope and Hierarchical Names



Textbook example of hierarchical names:

```
module top;
  reg r;          //hierarchical name is top.r
  wire w;         //hierarchical name is top.w

  b instance1();

  always
  begin: y
    reg q;        //hierarchical name is top.y.q
  end

  task t;
  begin: c        //hierarchical name is top.t.c
    reg q;        //hierarchical name is top.t.c.q
    disable y;    //OK
  end
  endtask
endmodule

module b;
  reg s;          //hierarchical name is top.instance1.s

  always
  begin
    t;            //OK
    disable y;    //OK
    disable c;    //Nope, c is not known
    disable t.c; //OK
    s = 1;        //OK
    r = 1;        //Nope, r is not known
    top.r = 1;    //OK
    t.c.q = 1;    //OK
    y.q = 1;      //OK, a different q than t.c.q
  end
endmodule
```

The `disable` keyword is used to terminate tasks, similar to

References

1. Donald E. Thomas and Philip R. Moorby. " The Verilog Hardware Description Language, Fifth Edition." Springer. 2002