Outline

- Behavior level modeling and Event-driven simulation
- Procedural Construct and Assignment
  - initial block
  - always block
  - procedural assignment
- Control Construct
  - if/else, case-endcase, for, repeat, while, forever, fork/join
- Modeling Examples
- Function and Task
Behavior Level

- Recall: Register-Transfer level
  - Describe sequential elements (register) and combinational elements (transfer) of a synchronous circuit structurally in different parallel constructs.

- Behavior Level
  - Describe the functionality (operations, timing, sensitivity list) of a circuit. (in or not in different parallel constructs)
Relationship

• Flexible
• Hardware modeling
• Waveform description
• Implicit structure description for modeling

Behavior level model

RT-level model

• Accurate
• Hardware modeling
• Explicit structure description for modeling
Levels of Abstraction for Synchronous Circuit

- Description of a RT level model
  - Structure: constructs are separated for combinational and sequential circuits
  - Signal: continuous evaluate-update, pin accurate
  - Timing: cycle accurate

- Description of a behavior level model
  - Structure: construct can be combinational, sequential, or hybrid circuits.
  - Signal: event-driven, timing, arithmetic (floating point or integer), ... to pin accurate
  - Timing: untimed (ordered), approximate-timed (with delay notification), cycle accurate
Event

- A data type has state and timing description.
  - state: level change (1→0, 0→1, 1→X, 0→Z, etc.), edge (defined logic-level transition: e.g. 0→1)
  - timing: virtual time in simulator

- example: signal s changed from 0 to 1 at 3ns
**Event-Driven Simulation**

- Execute statement (evaluate expression and update variable) when defined event occurs.
  - input transition cause an event on circuit.
  - simulation is on the OR-ed occurrence of sensitive events

- Accelerate simulation speed
  - only evaluate expression when variables on RHS change

- Allow high-level description (behavior) of a construct
Example

\`timescale 1ns/10ps

Event-driven statement execution number:
4+(2+1+1)+(1+1+1)+(1+1+1)+1 = 15

Continuous assignment execution number:
(40ns / 10ps)+1 = 4001
Event-Driven Construct

- Event-driven construct is a non-preemptive execution unit in simulator. (Important!!)
  - preemptive/non-preemptive

- Event-driven construct is triggered to simulate on programmer-defined events with event control identifier @

- Event-driven construct may generate new events to trigger other event-driven construct

- Simulator runs until there is no triggered event and then advance time.
Example
Example

- Event de-scheduling
  - statement execution is cancelled

![Diagram showing event de-scheduling with arrows and timing details.](image)
Rules of Event-Driven Construct

- Each construct is defined by keyword:
  - *initial*: not triggered by any signal, triggered on the beginning of simulation
  - *always*: triggered by programmer defined events

- Only event-driven variable (*reg*) can be LHS in event-driven construct.
  - RHS is not restricted.

- Event-driven variable is updated by *procedural assignment*. 
Procedural Assignments (1)

- Procedural assignments drive values or expressions onto registers (\textit{reg, integer, real, time})

```verilog
module adder32(sum, cout, a, b, ci);
    output [31:0] sum;
    output cout;
    input [31:0] a, b;
    input ci;
    reg [31:0] sum;
    reg cout;
    
    always @(a or b or ci)
    begin
        {carry, sum} = a + b + ci;
    end
endmodule
```

Event Sensitivity List:
carry and sum depend on a, b, ci

Statement
Procedural Assignments (2)

- Appear in initial block or always block:

  ```
  initial
  begin
  {cout, sum} = a + b + cin;
  end

  always@( a or b or cin)
  begin
  {cout, sum} = a + b + cin;
  end
  ```

- Like C: RHS evaluated and assigned to LHS before next statement executes

- RHS may contain wires and regs → Reference
  - Two possible sources for data

- LHS must be a reg → Event-driven Variable
  - Primitives or cont. assignment may only set wire values
module assignment_test;
  reg [3:0] r1, r2;
  reg [4:0] sum2;
  reg [4:0] sum1;

  always @(r1 or r2)
    sum1 = r1 + r2;

  initial
  begin
    r1 = 4’d0010; r2=4’d1001;
    sum2 = r1+r2;
    $display(" r1   r2   sum1   sum2");
    $monitorb(r1, r2, sum1, sum2);

    #10 r1 = 4’d0011;
  end

endmodule
Procedural Assignments

- There are two types of procedural assignment statements: blocking and nonblocking.

- Fundamental problem:
  - In a synchronous system, all flip-flops sample simultaneously.
  - In Verilog, always blocks triggered by the same event run in some undefined order, but *virtually* concurrently.
Shift Register (INCORRECT)

- This doesn’t work as you’d expect:

```verilog
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

- These run in some order, but you don’t know the priority in simulator.
- May lead to race condition.
Race Condition

Race condition occurs when two or more statements that are scheduled to execute in the same simulation time-step (at the same virtual time), would give different results when the order of statement execution is changed, as permitted by the IEEE Verilog Standard.
Non-blocking Assignments

- To model concurrent statements.

```verilog
reg d1, d2, d3, d4;
always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Nonblocking rule:
RHS evaluated when assignment runs
LHS updated only after all events for the current instant have run
Non-Blocking/Blocking Assignments

- A sequence of nonblocking assignments don’t communicate

\[
\begin{align*}
a & = 1; \\
b & = a; \\
c & = b; \\
\text{Blocking assignment:} \\
a & = b = c = 1
\end{align*}
\]

\[
\begin{align*}
a & = 1; \\
b & = a; \\
c & = b; \\
\text{Nonblocking assignment:} \\
a & = 1 \\
b & = \text{old value of } a \\
c & = \text{old value of } b
\end{align*}
\]
Non-blocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

\[ \begin{align*}
  a &= 1; \\
  b &= a; \\
  c &= b; \\
  a &= 1; \\
  b &= a; \\
  c &= b;
\end{align*} \]
Programming Tip

- Use Non-blocking statements to eliminate race conditions

// Illustration 1: blocking
always @(posedge clk)
    a = b;
always @(posedge clk)
    b = a;

// Illustration 2: nonblocking
always @(posedge clk)
    a <= b;
always @(posedge clk)
    b <= a;

- In illustration 1, there is a race condition. In illustration 2, using nonblocking to eliminate it.
Exercise

- Try to give a race condition example.

- Explain the possible result if simulation runs.
Procedural Assignments

- **Blocking assignment** statements are executed in the order they are specified in a sequential block

```verilog
reg x, y, z;
reg [7:0] rega, regb;
integer count;
initial
begin
    x=0; y=1; z=1;
    count=0;
    rega=8'b0; regb=rega;
    #15 rega[2]=1'b1;
    #10 regb[7:5]={x, y, z};
    count=count+1;
end
```

**Statements executed**
- At time 0:
  - `x=0; y=1; z=1;
  - `count=0; rega=0; regb=rega=0;`
- At time 15:
  - `rega[2]=1;`
- At time 25:
  - `rega[7:5]={x, y, z};
  - `count=count+1;`
Procedural Assignments

- **Nonblocking assignment** statements allow scheduling of assignment without blocking execution of the statements that follow in a sequential block. A `<=` operator is used to specify nonblocking assignments.

```verilog
initial begin
    x=0; y=1; z=1;
    count=0;
    rega = 8'b0;
    regb = rega;
end
initial begin
    rega[2] <= #15 1'b1;
    regb[7:5] <= #10 {x,y,z};
    count <= count + 1;
end
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Statements executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x=0; y=1; z=1; count=0; rega=0; regb=rega=0;</td>
</tr>
<tr>
<td>10</td>
<td>regb[7:5]={x,y,z};</td>
</tr>
<tr>
<td>15</td>
<td>rega[2]=1;</td>
</tr>
</tbody>
</table>

---

**Graduate Institute of Electronics Engineering, NTU**

**Digital System Design**

**Logic Design at Behavior Level - 2009.3.11**

**Chihhao Chao**

**pp. 25**
Procedural Assignments

- Application of non-blocking assignments

```verilog
always @(posedge clk)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clk) in2^in3;
    reg3 <= #1 reg1; //the old value of reg1
end
```
Register Assignment

- A register may be assigned value only within:
  - a procedural statement
  - a user-defined sequential primitive
  - a task, or
  - a function.

- A reg object may never be assigned value by:
  - a primitive gate output or
  - a continuous assignment

Event-driven simulation
Continuous simulation
(Evaluate every time advance)
Sequential Block

- Sequential block may appear in an always or initial statement

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

Runs when simulation starts
Terminates when control reaches the end
(one time sequential activity flow)
Good for providing stimulus (testbenches); not synthesizable

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

Runs when simulation starts
Restarts when control reaches the end
(cycle sequential activity flow)
Good for modeling/describing hardware
initial and always

- Run until they encounter a delay (which define an event to wait)

```verilog
initial begin
    #10 a = 1; b = 0;
    #10 a = 0; b = 1;
end
```

- or a wait for an event (convert to Boolean logic)

```verilog
always
    begin
        wait(i); a = 0;
        wait(~i); a = 1;
    end
```
module always_1; reg Y, Clk;
always // Statements in an always statement execute repeatedly:
begin: my_block // Start of sequential block.
   @(posedge Clk) #5 Y = 1; // At +ve edge set Y=1,
   @(posedge Clk) #5 Y = 0; // at the NEXT +ve edge set Y=0.
end // End of sequential block.
always #10 Clk = ~ Clk; // We need a clock.
initial Y = 0; // These initial statements execute
initial Clk = 0; // only once, but first.
initial $monitor("T=%2g","time," Clk"," Y",");
initial #70 $finish;
endmodule

>>>>

<table>
<thead>
<tr>
<th>T</th>
<th>Clk</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>55</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Sequential and Parallel Blocks

- There are two types of blocks: **sequential** blocks and **parallel** blocks.

//Illustration 1: sequential block without delay
reg x, y;
reg [1:0] z, w;
initial
begin
  x=1'b0;
  y=1'b1;
  z={x, y};
  w={y, x};
end

//Illustration 2: sequential blocks with delay
reg x, y;
reg [1:0] z, w;
initial
begin
  x=1'b0;
  #5 y=1'b1;
  #10 z={x, y};
  #20 w={y, x};
end
Sequential and Parallel Blocks

- Parallel blocks, specified by keywords fork and join, execute concurrently

```verilog
// Parallel blocks with delay
reg x, y;
reg [1:0] z, w;
initial
fork
  x = 1'b0;
  #5 y = 1'b1;
  #10 z = {x, y};
  #10 z = {x, y};
  #20 w = {y, x};
join
```
Sensitive to Multiple Events

- Use **or** in sensitivity list
- **example:**

```vhdl
always@( operand1 or operand2 )
    sum = operand1 + operand2;

always@( operand1 or operand2 or carry_in ) begin
    sum = operand1 + operand2 + carry_in;
end
```
Exercise

Draw the waveform for the following code (run to 20ns):

```verilog
reg a, b;
reg c, d, e;
initial begin
    a = 1'b0;
    #3 b = 1'b1;
    #4 a = 1'b1;
end
initial begin
    #5 c = a ^ b;
    #10 d = a | b;
end
initial
    e <= #8 a & b;
```

```verilog
reg a, b;
reg c, d, e;
initial begin
    a = 1'b0;
    #3 b = 1'b1;
    #3 a = 1'b1;
end
always@( a or b ) begin
    #5 c = a ^ b;
    #10 d = a | b;
end
always@( a or b )
    e <= #8 a & b;
```
Conditional Statements

- If and If-else statements

```python
if (expression)
    statement
else
    statement
```

```python
if (expression)
    statement
else if (expression)
    statement
else
    statement
```
Conditional Statements

If and If-Else Statements (cont.)

- Examples

```plaintext
if (rega >= regb)
    result = 1;
else
    result = 0;
```

```plaintext
if (index > 0)
    if (rega > regb)
        result = rega;
    else
        result = 0;
else
    $display("* Warning * index is equal or small than 0!");
```
Multiway Branching

- The nested **if-else-if** can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the **case** statement.

```verbatim
reg [1:0] alu_control;
... 
case (alu_control)
    2'b00: y=x+z;
    2'b01: y=x-z;
    2'b10: x*z;
... 
default: y='bx;
endcase
```
Multiway Branching

4-to-1 Multiplexer with case Statement

```verilog
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
//port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
  case ({s1, s0})
    2'd0: out = i0;
    2'd1: out = i1;
    2'd2: out = i2;
    2'd3: out = i3;
    default: out = 1'bx;
  endcase

endmodule
```
Multiway Branching

- There are 2 variations of the case statement. They are denoted by keywords `casex` and `casez`
  - `casez` treats all `z` values in the case alternatives or the case expression as don’t cases. All bit positions with `z` can also represented by `?` In that position
  - `casex` treats all `x` and `z` values in the case item or the case expression as don’t cares

```vhdl
reg [3:0] encoding;
integer state;

casex (encoding)
  //logic value x represents a don't care bit
  4'b1xxx: next_state=3;
  4'bx1xx: next_state=2;
  4'bxx1x: next_state=1;
  4'bxxx1: next_state=0;
  default: next_state=0;
endcase
```

```
encoding=4'b10xz     next_state=3
```
While Loop

- The **while** loop executes until the **while**-expression becomes false

```
initial    //Illustration 1:
begin
  count=0;
  while(count<128) begin
    $display("count=",count);
    count=count+1;
  end
end
```

```
initial
begin
  reg [7:0] tempreg;
  count = 0;
  tempreg = reg;
  while (tempreg)
    begin
      if (tempreg[0]) count = count + 1;
      tempreg = tempreg >> 1;
    end
rega = 101;
<table>
<thead>
<tr>
<th>tempreg</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>2</td>
</tr>
</tbody>
</table>
```
For Loop

- The keyword `for` is used to specify this loop. The `for` loop contain 3 parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

```vhdl
//Initialize array elements
`define MAX_STAGES 32
reg [0:31] array;
integer i;

initial
begin
  for(i=0;i<32;i=i+2)
    array[i]=0;
  for(i=1;i<32;i=i+2)
    array[i]=1;
end
```
Repeat Loop

The keyword `repeat` is used for this loop. The `repeat` construct executes the loop a fixed number of times. module multiplier(result, op_a, op_b);

...  
reg shift_opa, shift_opb;
parameter size = 8;
initial begin
  result = 0; shift_opa = op_a; shift_opb = op_b;
  repeat (size)
      begin
          #10 if (shift_opb[1])
              result = result + shift_opa;
          shift_opa = shift_opa << 1;
          shift_opb = shift_opb >> 1;
      end
  end
endmodule
Forever Loop

- The keyword **forever** is used to express the loop. The loop does not contain any expression and executes forever until the **$finish** task is encountered.

```verilog
//Clock generation
//Clock with period of 20 units)
reg clk;

initial
begin
  clk=1'b0;
  forever #10 clk=~clk;
end

//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

initial
  forever @(posedge clk) x=y;
```
Encoder

module encoder (code, data);
output [2:0] code;
input [5:0] data;
reg [2:0] code;

always@( data ) begin
  if ( data==6'b000001 ) code = 3'd0;
  else if ( data==6'b000010 ) code = 3'd1;
  else if ( data==6'b000100 ) code = 3'd2;
  else if ( data==6'b001000 ) code = 3'd3;
  else if ( data==6'b010000 ) code = 3'd4;
  else                                     code = 3'd5;
end
Multiplexer

module MUX_4_32( out, in3, in2, in1, in0, sel, en);
output out;
input [31:0] in3, in2, in1, in0;
input [1:0] sel;
input en;
reg [31:0] mux_r;
assign out = en ? mux_r : 32’bz;
always@( in3 or in2 or in1 or in0 or sel ) begin
  case( sel )
    2’d0: mux_r = in0;
    2’d1: mux_r = in1;
    2’d2: mux_r = in2;
    2’d3: mux_r = in3;
  endcase
end
endmodule
Exercise

- Try to write a one-hot to binary decoder using
  - if-else
  - case
**D Flip-Flop**

- Very basic: an edge-sensitive flip-flop

```vhdl
reg q;
always @(posedge clk)
  q <= d;
```

- q <= d assignment runs when clock rises: exactly the behavior you expect

**Keywords:**
- `posedge` for positive edge trigger
- `negedge` for negative edge trigger
D Flip-Flop with Asynchronous Reset

- sensitive to another edge event

```verilog
reg q;
always @(posedge clk or posedge rst) begin
  if( rst==1'b1 ) q <= 1'b0;
  else q <= d;
end
```
Shift Register

module shift_register_4x8( so, si, clk);
output [3:0] so;
input [3:0] si;
input clk;
reg [3:0] pipe [0:7];
integer p;

assign so = pipe[7];

always@( posedge clk ) begin
    for( p=7; p<=1; p=p-1 ) pipe[p]<=pipe[p-1];
    pipe[0] <= si;
end
Exercise

- Try to write a Finite Impulse Response (FIR) filter with coefficient {0.25, 0.5, 0.25}
Digital System Design

Timing Modeling at Behavior Level

Supplement
Timing Controls

- There are 3 methods of timing control: delay-based timing control, event-based timing control, and level-sensitive timing control.

- Delay-Based Timing Control

```plaintext
#2 y=1;
#4 x=0;
#(1:2:3) q=0;
#5 y=tempxz;

y<=#15 x+z;
```
Timing Controls

- Event-Based Timing Control
  - Regular Event Control

```verilog
test;
reg clk;
reg [2:0] q,d;
always #5 clk=~clk;
initial
begin
  clk=0;
  #2 q=0;d=0;
  #2 d=2;
  @(posedge clk) q=4;
  @(negedge clk) q=1;
  q=@(posedge clk) 3;
  #10 $finish;
end
endmodule
```

<table>
<thead>
<tr>
<th>Name</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d[2:0]</td>
<td>x</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>q[2:0]</td>
<td>x</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Timing Controls

- Event-Based Timing Control
  - Named Event Control
    ```
    event received_data;
    always @(posedge clk)
    begin
      if (last_data_packet)
        -> received_data;
    end
    always @(received_data)
    data_buf = {data_pkt[0],data_pkt[1]};
    ```

- Event OR Control
  ```
  always @(rst or posedge clk or set)
  begin
    if(rst) q=0;
    else if (set) q=1;
    else q=d;
  end
  assign _q=~q;
  ```
Named Block

- named blocks (example: comparator)

```verilog
text
```
Timing Controls

- Level-Sensitive Timing Control

```verbatim
always
  wait (count_enable) #20 count = count +1 ;
  // 1. If count_enable is logical 1, count=count+1
  //    after 20 time unit.
  // 2. If count_enable stays at 1, count will be
  //    incremented every 20 time units
```
Digital System Design

Function & Task
What are Function and Task for?

- Similar to the role of the function in high-level programming language

- Improve modeling/describing efficiency
- Reduce code size
- Improve readability

- Only invoked in event-driven construct
  - always block and initial block
Functions

- Functions are declared with the keywords **function** and **endfunction**. Functions are used if all of the following conditions are true for the procedure:
  - There are no delay, timing, or event control constructs in the procedure
  - The procedure returns a single value
  - There is at least one input argument

- There are some peculiarities of functions. When a function is declared, a register with name (name of function) is declared implicitly inside.

- Functions cannot invoke other tasks. They can only invoke other functions.
Functions example

module check(number);
  input [9:0] number;
  reg correct;

  always @(number)
  begin
    correct = berger(number);
    $display("The correct of the number: %b is %b ",number,correct);
  end

function berger;
  input [9:0] number;
  reg [2:0] temp3;
  integer i;
  begin
    temp3 = 3'b000;
    for(i=3;i<10;i=i+1)
      temp3 = temp3 + number[i];
    temp3 = temp3 ^ 3'b111;
    if( temp3 === number[2:0])
      berger = 1'b1;
    else
      berger = 1'b0;
  end
endfunction
endmodule

module test;
  reg [9:0] num;
  check chk1(num);
  initial
    begin
      num = 10'b0111010011;
      #10 num = 10'b0111111011;
      #10 $finish;
    end
endmodule

simulation result
The correct of the number: 0111010011 is 1
The correct of the number: 0111111011 is 0
module mark1Task;
    reg [15:0] m[0:8191];
    reg [12:0] pc;
    reg [12:0] acc;
    reg [15:0] ir; // instruction register
    reg clk;
    always begin: executeInstructions
        @(posedge clk)
            ir=m[pc];
        @(posedge clk)
            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;
            3'b111: acc=multiply(acc,m[ir[12:0]]);
        endcase
        pc=pc+1;
    end
endmodule

function [12:0] multiply;
    input [12:0] a;
    input [15:0] b;
    reg [5:0] mcnd, mpy;
    begin: serialMult
        mpy=b[5:0];
        mcnd=a[5:0];
        multiply=0;
        repeat(6)
            begin
                if (mpy[0])
                    multiply=multiply+{mcnd, 6'b0};
                multiply=multiply>>1;
                mpy = mpy >> 1;
            end
        end
        a=prod;
    end
endtask
endmodule
Tasks

- Task are declared with the keyword `task` and `endtask`. It must be used if any one of the following conditions is true for the procedure:
  - There are delay, timing, or event control constructs in the procedure.
  - The procedure has zero or more than one output arguments.
  - The procedure has no input argument.
Tasks example1

module operation;
...

parameter delay=10;

reg [15:0] A,B;
reg [15:0] AB_AND,AB_OR,AB_XOR;

always @(A or B)
begin
    bitwise_oper(AB_AND,AB_OR,AB_XOR,A,B);
end
...

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
endmodule
module mark1Task;
  reg [15:0] m[0:8191];
  reg [12:0] pc;
  reg [12:0] acc;
  reg [15:0] ir; //instruction register
  reg clk;
  always begin:
    executeInstructions
      @(posedge clk)
        ir=m[pc];
      @(posedge clk)
        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;
          3'b111: multiply(acc,m[ir[12:0]]);
        endcase
    pc=pc+1;
  end

  task multiply;
  inout [12:0] a;
  input [15:0] b;
  reg [5:0] mcnd, mpy;
  reg [12:0] prod;
  begin: serialMult
    mpy=b[5:0];
    mcnd=a[5:0];
    prod=0;
    repeat(6)
      begin
        if(mpy[0])
          prod=prod+{mcnd,6'b0};
        prod=prod>>1;
        mpy = mpy >>1;
      end
    a=prod;
  end
endtask
endmodule
Differences Between Tasks and Functions

- Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments.
- Function are used when common provides exactly one output. Function are typically used for conversions and commonly used calculations.
- Tasks can have input, output, and inout port; functions can have input ports.
- Tasks and functions have local variables, registers, time variables, integers, real, or events.
# Summary: Tasks and Functions

## Functions

A function can enable another function but not another task

Function always execute in 0 simulation time

Functions must not contain any delay, event, or timing control statements

Functions must have **at least one input** argument. They can have more than one input

Functions always return a single value. They **cannot** have output or inout arguments

## Tasks

A task can enable other tasks and functions

Tasks may execute in non-zero simulation time

Tasks may contain delay, event, or timing control statements

Task may have **zero or more** arguments of type input, output or inout

Tasks do not return with a value but can pass multiple values through output and inout arguments