Tasks

properties

- "called" from an initial or always statement
- can have ~~to~~ any number ports (includes zero)
- may have timing and/or $\underbrace{\text{event controls}}_{@(\text{posedge}\cdots)}$
  (# delays)

syntax:

```
  ✓          ✓
task [automatic] task_name ;
      port declarations;
      procedural statements ;
  ✓
endtask
```

✓ denotes keywords

Some notes

1) default type for all ports is <u>reg</u>

2)
$$\begin{tikzcd}
\text{ports in modules} \arrow[r, phantom, "\neq"] \arrow[d, Rightarrow] & \text{ports in tasks} \arrow[d, Rightarrow] \\
\text{I/F signals} & \text{values passed to/from the task}
\end{tikzcd}$$

3) tasks can operates on any variables declared in the module

Example 8-4

Direct Operation on reg Variables

```
//Define a module that contains the task asymmetric_sequence
module sequence;
...
reg clock;  ← declaration
...
initial
        init_sequence; //Invoke the task init_sequence
...                                              task called
always
begin
        asymmetric_sequence; //Invoke the task asymmetric_sequence
end
...
...
//Initialization sequence
task init_sequence;  ← task
begin                               template
        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;             task
        #3 clock = 1'b0;             template
        #10 clock = 1'b1;
end
endtask
...
...
endmodule
```

example 8-2

Input and Output Arguments in Tasks

```
//Define a module called operation that contains the task bitwise_oper
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
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
...
endmodule
```

ordering matters...

4) Keyword "automatic", if included,
makes the task re-entrant

example 8-5

Re-entrant (Automatic) Tasks

```
// Module that contains an automatic (re-entrant) task
// Only a small portion of the module that contains the task definition
// is shown in this example. There are two clocks.
// clk2 runs at twice the frequency of clk and is synchronous
// with clk.
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
    [redacted]
    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
```

for functions,

- 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 no output or inout arguments.
- There are no nonblocking assignments.

(Pg 178)

functions

Syntax : ✓
```
function [range] function_name;
    input declarations;
    other local declarations;
  ✓ procedural statements;
endfunction [illegible]
```

✓ Keywords

if no range is specified, function
returns a single bit

e.g.

```
module foo;
 parameter max = 8;
   :
   :
  function [max-1:0] reverse; // func template
     input [max-1:0] Din;
     integer K;
    begin
      for (K=0; K<max; K=K+1)
         reverse[max-K-1] = Din[K];
    end
  endfunction
```

---

comment: "function" ⟹ some sort of register to store the function result

How do we call this function?

```
module example;
     ⋮
  reg [max-1:0] new_val, old_val;
       ⋮
     function [max-1:0] reverse; // template
         ⋮
     endfunction
         ⋮
    always @(        )
         new_val = reverse(old_val);   ← func call

endmodule
```

Example 8-7 Parity Calculation

```
//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
...
...
//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
```

example 8-9

Left/Right Shifter

```
//Define a module that contains the function shift
module shifter;
...
//Left/right shifter
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

//Compute the right- and left-shifted values whenever
//a new address value appears
always @(addr)
begin
       //call the function defined below to do left and right shift.
        left_addr = shift(addr, `LEFT_SHIFT);
        right_addr = shift(addr, `RIGHT_SHIFT);
end
...
...
//define shift function. The output is a 32-bit value.
function [31:0] shift;
input [31:0] address;
input control;
begin
        //set the output value appropriately based on a control signal.
        shift = (control == `LEFT_SHIFT) ?(address << 1) : (address >> 1);

end
endfunction
...
...
endmodule
```