ELEC2221
Digital Systems and Signal Processing

Loops, time, functions, tasks
and system functions
Loop statements

- for
- while
- do – while
- repeat
- foreach
- forever
for loop

```c
for (initialisation_expression; condition_expression; step_expression)
    statement;
```

- `initialisation_expression` executes once at the start;
- `condition_expression` executes at the start of each iteration; if its boolean value is not true (e.g. false, x, or z), the loop exits; otherwise statement executes;
- `step_expression` executes at the end of each iteration.

**for loop execution order:**

1. **first iteration:**
   - `initialisation_expression`
   - `condition_expression`
   - `statement`
   - `step_expression`
2. **2nd iteration:**
   - `initialisation_expression`
   - `condition_expression`
   - `statement`
   - `step_expression`
3. **etc...**
for loop control variable

module example;

initial
begin
loop1:
    for (int i = 0; i < 10; i++)
        ...
end

initial
begin
loop2:
    for (int i = 10; i > 0; i--)
        ...
end
endmodule

In this example both loops execute simultaneously in two separate initial blocks.

each loop has its own control variable i; these are two different variables.
while loop

`while (expression) statement;`

`expression` is evaluated at the start of each iteration. If it returns true, `statement` executes, otherwise the loop exits.

```plaintext
int i=0;
while(i<10) begin
    #5 $display("i = %d\n",i);
    i++;
end
```
do-while loop

do statement while (condition);

At each iteration statement executes first, then condition executes. The loop exits when condition is false. In this loop, statement executes at least once.

```c
int i=0;
do
    a[i] = i*i;
i++; // Note: blocking assignment
while(i<10);
```
repeat loop

repeat (expression) statement;

The repeat loop is similar to for loop but has no loop control variable; statement is evaluated expression times.

```cpp
int i=0;
repeat (16)
   $display("i = %d", i++);
```

is equivalent to:

```cpp
for(int i=0;i<16;i++)
   $display("i = %d",i);
```
foreach loop

```
foreach ( array_identifier [ loop_variables])
    statement;
```

**foreach** loop iterates over the elements of an array. Its argument is an identifier that designates an array followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

```plaintext
string words [1:2] = { "hello", "world" }; // array of strings
foreach( words [ j ] )
    $display( j , words[j] ); // print each index and value
```

```plaintext
logic mem [0:63] [7:0];
foreach( mem[ address, word ] )
    mem[address][word] = 1'b0; // initialize
```
forever loop

forever statement;

forever loop continuously repeats its statement. Only use in procedural blocks and with timing controls, otherwise it will hang the simulation.

```
initial
begin
  clk=0;
  forever #10 clk = ~clk;
end
```
time literals, timescale directive

time literals can have integer or fixed-point format:
0.1ns
40ps

literals in simulation time delays may specify time without a time unit:
#5 ...
#2 ...

The time unit is defined in a timescale directive:
`timescale time_resolution / precision

`timescale 1ns / 10ps
...
#5; // wait 5ns

#0.01; // wait for 0.01*time_resolution (10ps)
#0.001; // this is below time resolution hence it will produce zero delay
SystemVerilog timeunit, timeprecision

- Instead of a directive, can use declarations in SystemVerilog:

  timeunit 1ns;

  timeprecision 100ps;

- Better because the scope is defined. Timescale directive applies across modules.

- Precision must be less than or equal to time unit. Ratio of 1:10 or 1:100 is common

- Some simulators (e.g. ModelSim) insist that, if used in one, timeunit, timeprecision must be in every module. (Watch out for this!)
Procedural blocks (processes)

- initial // executes only once at the beginning of simulation
- final // executes once at the end of simulation – very rare
- always, always_comb, always_latch, always_ff
  // these blocks execute continuously when triggered by events
- task // executes when called, similar to function with no return value
- function // executes when called and returns a value
**Simple Function**

```plaintext
function int sqr(int a);
    sqr = a*a;
endfunction

In SystemVerilog, we can also write

```plaintext
function int sqr(int a);
    return a*a;
endfunction
```

We can also specify the input

```plaintext
function int sqr(input int a);
    sqr = a*a;
endfunction
```

This is an example of a "pure" function: no modification of variables outside the function; no output or inout parameters.

(System)Verilog allows "impure" functions, but this is often considered bad practice – difficult to understand and keep track of.
Constant Function

Called once at elaboration (link) time

module decoderlogN
  #(parameter N = 8)
  (output logic [N-1:0] y,
   input logic [clog2(N)-1:0] a);

  //function clog2 declared here

  always_comb
    y = 1'b1 << a;

endmodule

function int clog2(input int n);
begin
  // begin .. end aren't strictly necessary
  clog2 = 0;
  n--;
  while (n > 0)
    begin
      clog2++;
      n >>= 1;
    end
  end
endfunction
Reentrant (automatic) functions

- In C, C++ etc. functions are "reentrant"
  - New data structure is created each time the function is called.

- In Verilog, by default, functions are not reentrant
  - Data structure is shared between all active instances of a function

- For example, this fails
  
  function int factorial (input int op);
  if (op >= 2)
    factorial = factorial (op - 1) * op;
  else
    factorial = 1;
  endfunction

- We can force a function to be reentrant

  function automatic int factorial (input int op);
  if (op >= 2)
    factorial = factorial (op - 1) * op;
  else
    factorial = 1;
  endfunction

  This will work correctly – new data structure for each instance
Dining Philosophers

• Example of Resource sharing
  – N (e.g. 5) philosophers sat round a table. N chopsticks (very unhygenic!)
  – Each has a bowl of noodles
  – Can eat if the two chopsticks each side are not being used
  – Pick up chopsticks, eat for a random time, put chopsticks down
  – Wait for a random time and try again

• Non-synthesisable code to model system
module dp;
  localparam shortint unsigned n = 5;
  localparam time timeout = 5;
  time delay[1:n] = {7, 19, 7, 7, 7};
  // note that philosopher 2 is slow
  int seed;
  logic active [1:n] = {0, 0, 0, 0, 0};
  logic chopsticks [0:n-1] = {1, 1, 1, 1, 1};

function automatic getsticks (int i);
  int rh = i;
  reg ok;
  if (i == n)
    rh = 0;
  ok = chopsticks[i-1] && chopsticks[rh];
  if (ok) begin
    chopsticks[i-1] = 0;
    chopsticks[rh] = 0;
  end
  getsticks = ok; // or return ok;
endfunction

Note that getsticks is a reentrant (automatic) function. It is impure because vector chopsticks is modified. It might be better to use a task instead.
task automatic putsticks(int i);
    int rh = i;
    if (i == n)
        rh = 0;
    chopsticks[i-1] = 1;
    chopsticks[rh] = 1;
endtask

• This is a task, not a function, because nothing is returned.
• It is automatic because each call needs its own data space.
System Verilog provides a large number of built-in system commands and tasks

$display - displays a message
$time - returns current simulation time
$finish - terminates simulation
$monitor - displays a message if there are changes in monitored variables
$error - displays an error message in an assertion
$past() - return the value of an expression in the previous clock cycle.
$warning - displays a warning message in an assertion
$fatal - specifies a run-time fatal error
$info - displays an information message in an assertion
$random - returns a random number
etc.
initial
begin
  // specify variables to be monitored
  $monitor ("a=%h, b=%h, c=%h\n", a,b,c);
  reset = 0;
  a = 0;
  b = 0;  // $monitor executes now
  #5 reset = 1;
  #15 reset = 0;
  #10 a = 1;  // $monitor executes, as variable a changed
  #10 {a,b} = 2'b11;  // $monitor executes again, b has now changed
  #10 $finish;  // simulation stops
end // simulation stops
$error and $time example

always_ff @(posedge clk)
begin
    state <= next;

    if (state == idle)
        assert (out1 == 1’bz) // sequential assertion
    else // else clause executes if assertion fails:
        $error("out1 is not high-z in idle state at time %0t",$time);
end

Assertion statements are ignored in hardware synthesis, they are only used in simulations. More in a later lecture.