Sequential and Parallel Blocks

• Up to this point, all block constructs used have been nominally sequential, that is, they are “begin … end” blocks.

• Verilog also supports parallel blocks that use a “fork…join” construct.

• Once again, the growth of the language has blurred the distinction.
Block Statements

• Block statements are used to group two or more statements together so that they act as one statement, syntactically. They are of two types.

• Sequential block statements are enclosed between the keywords `begin` and `end`. The statements in this block are executed in a sequential manner.

• Parallel block statements are enclosed between the keywords `fork` and `join`. The statements in this block are executed concurrently.
Synthesis Support

- Only “begin…end” blocks are supported for synthesis.
- “fork…join” is not. Therefore “fork…join” must never be used in circuit descriptions but only in test fixtures.
Sequential Blocks

In sequential blocks, statements tend to be evaluated sequentially:

```verilog
initial begin
    x = 1'b0;
    y = 1'b1;
    z = x & y;
    w = x | y;
end
```
Delays in Sequential Blocks

When delays are used in sequential blocks, the delay times are accumulated sequentially:

```
initial begin
    #5 x = 1’b1;  // assign value at t = 5 units
    #5 x = 1’b0;  // assign value at t = 10 units
    #5 x = 1’b1;  // assign value at t = 15 units
end
```
Parallel Blocks

In a parallel block, assignments are concurrent:

    initial fork
    x = 1'b0;
    y = 1'b1;
    z = x & y;
    w = x | y;

    join

This is a RACE CONDITION. It is impossible to predict the values of z and w.
Delays in Parallel Blocks

In parallel blocks, delays are relative to the time of entering the block:

initial fork

#5 x = 1’b1;//assign value at t = 5 units
#5 x = 1’b0;//assign value at t = 5 units
#5 x = 1’b1;//assign value at t = 5 units

join

Again, this is a RACE CONDITION. It is impossible to predict the value of x. All assignments to x are concurrent.
“fork...join” is unsupported for synthesis and can cause race conditions with uncertain evaluation in test fixtures.

Nevertheless, it can make stimulus more readable:

```
initial fork

bus = 16'h0000; //do at t = 0

#10 bus = 16'hC5A5; //do at t = 10

#20 bus = 16'hFFAA; //do at t = 20

join
```
More Fork Join Uses

• Fork spawns new processes on the host system.
• All processes in a fork...join block run in parallel.
• They do not need to be simple statements. Can be complex routines, subroutines.
• Main thread resumes only after all parallel tasks have finished.
Non-blocking Assignment Operator

- The addition of the non-blocking assignment operator (<=) thoroughly confuses the distinction between sequential and parallel blocks.
- With that operator, assignments in a nominally sequential “begin…end” block are parallel.
- Circuit descriptions always use “begin…end” blocks. “fork…join” constructs are never to be used in synthesizable code.
Block Statements

initial Condition
begin
-------
-------
end

always Condition
begin
-------
-------
end

initial Condition
fork
-------
-------
join

always Condition
fork
-------
-------
join
Fork...Join Example

```vhdl
module forkjoin(clk, a, b);
input clk;
output a;
output b;
reg a, b;
initial begin
    a = 0;
b = 0;
end
always @(posedge clk)
fork
    #2 a = 1;
    #1 b = 1;
join
endmodule
```
Despite blocking assignment operator, execution of assignment statements is simultaneous. Assignment to b has a shorter delay so it happens before assignment to a.
Slight Modification

module forkjoin(clk, a, b);
input clk;
output a;
output b;
reg a, b;
initial begin
  a = 0;
  b = 0;
end

always @(posedge clk)
fork
  #2 a = 1;
  #1 b = a; //Change is here
join
endmodule
Output

Assignment to b happens while a is still zero.
module forkjoin(clk, a, b);
input clk;
output a;
output b;
reg a, b;
initial begin
  a = 0;
b = 0;
end

always @(posedge clk)
begin
  #2 a = 1;
  #1 b = a;
end
endmodule
Now blocking assignment operator forces sequential evaluation of statements. $a$ already is set to 1 before it is evaluated in the assignment to $b$ statement.
YOU MIGHT BE AN ENGINEER IF...
The salespeople at Best Buy can't answer any of your questions. You've ever repaired a $5 radio. Everyone else on the Alaskan cruise is on deck peering at the scenery, and you are still on a personal tour of the engine room. You are at an air show and know how fast the skydivers are falling, and how much horsepower each plane has. You go on the rides at Disneyland and sit backwards in the chairs to see how they do the special effects. You have "Dilbert" comics displayed anywhere in your work area. You have ever saved the power cord from a broken appliance. You know what http:// stands for. You spent more on your calculator than on your wedding ring. You still own a slide rule and you know how to work it. You window shop at Radio Shack. Your laptop computer costs more than your car.
Multiple Forever Loops

module forkjoin;

reg CLK0, CLK1;
initial begin
    CLK0 = 1'b1;
    CLK1 = 1'b1;
    fork
        forever #1 CLK0 = ~CLK0;
        forever #2 CLK1 = ~CLK1;
    join
end
endmodule
Running forever loops in parallel gets around the limitation of anything in a begin...end block after a forever loop never running.
Block Statements

• *fork-join* and *begin-end* blocks could be nested within each other.

• Nesting *begin-end* within *begin-end* is not useful because all statements will be executed sequentially anyway.
Block Statements

• Nesting *begin-end* within *begin-end* is justified only if it is used to control flow.

```plaintext
begin
if (condition)
begin
    ....
end
end
```

• Nesting *fork-join* block within *fork-join* block is not meaningful unless there is a delay preceding the inner *fork-join* block.
Nesting Example

module nesting1(clk, a, b, c, d, e, f);
  input clk;
  output a, b, c, d, e, f;
  reg a, b, c, d, e, f;
  initial begin
    a = 0;
    b = 0;
    c = 0;
    d = 0;
    e = 0;
    f = 0;
  end

always @(posedge clk)
fork
  #2 a = 1;
  #2 b = 1;
  begin
    #2 c = 1;
    #2 d = 1;
    #2 e = 1;
  end
  #2 f = 1;
join
endmodule
Nesting Example Results

a, b, c and f all go high two ns after the clock edge. d has an additional two ns delay and e two more.
Nesting, Opposite Case

module nesting2(clk, a, b, c, d, e, f);
  input clk;
  output a, b, c, d, e, f;
  reg a, b, c, d, e, f;

  initial
  begin
    a = 0;
    b = 0;
    c = 0;
    d = 0;
    e = 0;
    f = 0;
  end

always @(posedge clk)
  begin
    #2 a = 1;
    #2 b = 1;
    fork
      #2 c = 1;
      #2 d = 1;
      #2 e = 1;
    join
      #2 f = 1;
  end
endmodule
Nesting Example 2 Results

Now a comes 2 ns after clock, b 2 ns after a, c, d and e all together after another 2 ns delay and f 2 ns after them.
One More Fork Join Example

fork
   #20 c = 1;
   #30 d = 1;
   #10 e = 1;
join

Main thread will resume 30 time units after entering the fork...join block. The slowest process is the limiting factor.
Event-Based Timing Control

• “events” can be used to trigger execution of statements or blocks.

• Four types of events:
  – Regular
  – Named
  – OR
  – Level-sensitive
Regular Events

Regular Events are signaled by the use of the `@` symbol:

```verbatim
//flipflops are inferred on events:
always @(posedge …)
always @(negedge …)
/*Sensitivity to any change (edge) is also a form of event control*/
always @(SIGNAL) …
```
Named Event

• A named event is a data type that you trigger in a procedural block to enable actions.

• A named event must be declared before you can reference it.

• A named event has no duration and carries no value.

• You can only make an event occur from within a procedure.

• The -> operator is the trigger for the named event.
-> is a Blocking Operator

- Like the = assignment operator, -> is a blocking operator.
- The event that gets triggered is a static software construct (has to do with memory allocation, static vs. dynamic).
- -> is meaningless for hardware: not synthesizable.
- ->>> is a SystemVerilog non-blocking equivalent, not available in Verilog.
Named Event

/*This is an example of a data buffer storing data after the last packet of data has arrived.*/

event received_data;  //declare an event called received_data

always @(posedge clock) //check at each positive clock edge
begin
    if(last_data_packet) //If this is the last data packet
        -> received_data; //trigger the event received_data
    last_data_packet = 0; // last_data_packet is set in another block
end

always @(received_data)  //Await triggering of event received_data
    Data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]}; //When event is triggered, store all four packets in data buffer.
OR Events

OR event control is merely a formal way of specifying sensitivity to multiple signals:

```verilog
always @(posedge CLK or negedge RST) begin
```

Level-sensitive Events

Level-sensitive events use the keyword *wait*.

```vhdl
always wait (ENABLE) #10 count = count + 1;
```

What kind of a circuit might this imply?
Named Blocks

• You can name a block by adding: `<name Of block>` after the keywords `begin` or `fork`.

• You can declare local variables in the named block.

• You can disable a named block.

• Named blocks define a new scope in Verilog.
module top;
always
    begin: block1 //sequential block named block1
        integer i; //integer i is static and local to block1
        //can be accessed by hierarchical naming, top.block1.i
    end

initial
    fork: block2 //parallel block named block2
        reg i; // register i is local to block2
        //can be accessed by hierarchical naming, top.block2.i
    join
Scope of Local Variables

- A block will use its local variable if it exists. It will use a module variable if there is no local variable of that name.
- There is no conflict if a block local variable and a module variable have the same name.
module localvariable;

    integer i, A, B;
    reg CLK;
    initial begin
        CLK = 1'b0;
        forever #1 CLK = ~CLK;
    end

    initial begin
        A = 4; B = 3;
        #2 B = 0;
        #2 A = 5;
        #2 B = 8;
    end

    always @(posedge CLK) begin: block1
        reg [3:0] i;
        i = A - B;
    end

    always @(posedge CLK) i = A + B;
endmodule

Block variable i will form difference. Module variable i will form sum. There is no conflict or ambiguity. There is also no problem in assigning the result of operations on integers to a four-bit reg. This is Verilog, not VHDL.
Disabling named blocks

• The keyword *disable* provides a way to terminate the execution of any named block in the design.

• Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.

• Syntax:

  \[ \textit{disable \ <name-of-block>} \]

• When a named block is disabled, all the \texttt{events} scheduled by it are removed from the event queues.
Disabling named blocks

Example:

module true_bit (flag);
input [15:0] flag;
integer i;
wire [15:0] flag;
initial
begin
  i = 0;
  begin: block1
    while ( i < 16 )
      begin
        if ( flag[i] )
          begin
            $display ("TRUE bit at bit number %d",i);
            disable block1;
          end
        i = i + 1;
      end
  end
endmodule
Disable Support

- Disable is generally supported by commercial tools for simulation.
- The language specification for removing scheduled events is vague and interpreted differently by different developers.
- Use of disable is not recommended.
Disable Can be Tricky

“disable” can just go on to the next statement or it can get out of the block. The first disable in this example will just go on to the next statement. The second will exit the loop.

begin :LOOP forever begin :BODY
@ (posedge clock);
if (COND1) disable BODY; // equivalent to VHDL ‘NEXT’
if (COND2) disable LOOP; // equivalent to VHDL ‘EXIT’
end end
Will request_out be set to 0?

request_out <= 1'b1;
@ (posedge clock);
begin :LOOP: forever begin
    if (acknowledge_in == 1'b1) begin
        request_out <= 1'b0;
        disable LOOP;
    end
    @(posedge clock);
end end