Digital System Design

Logic Design with Behavioral Models

Lecturer: Wein-Tsung Shen (沈文中)
Date: 2005.03.11
Outline

- Operation
- Assignments
- Procedure
- Function & Task
Various Abstraction of Verilog HDL

- Switch level Verilog description of a inverter

```verilog
define module inv(out, in);
    // port declaration
    output  out;
    input    in;
    // declare power and ground
    supply1  pwr;
    supply0  gnd;
    // Switch level description
    pmos  S0(out, pwr, in);
    nmos  S1(out, gnd, in);
endmodule
```

Diagram: 
- `in` connected to `pwr`
- `out` connected to `gnd`
Various Abstraction of Verilog HDL

- Gate level Verilog description of a full-adder

```verilog
module fadder(cout, sum, a, b, cin);
// port declaration
    output cout, sum;
    input a, b, cin;
    wire net1, net2, net3;
// Netlist description
    xor U0(sum,a,b,cin);
    and U1(net1, a, b);
    and U2(net2, b, cin);
    and U3(net3, cin, a);
    or U4(cout, net1, net2, net3;
endmodule
```
Various Abstraction of Verilog HDL

- RTL level Verilog description of a full adder

```verilog
module fadder(cout, sum, a, b, cin);
// port declaration
    output cout, sum;
    input  a, b, cin;
    wire   cout, sum;
// RTL description
    assign sum = a^b^cin;
    assign cout = (a&b)|(b&cin)|(cin&a);
endmodule
```

Whenever \( a \) or \( b \) or \( c \) changes its logic state, evaluate \( sum \) and \( cout \) by using the equation:

\[
sum = a \oplus b \oplus c_i \\
cout = ab + bc + ca
\]
Various Abstraction of Verilog HDL

- Behavioral level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output cout, sum;
  input  a, b, cin;
  reg    cout, sum;
// behavior description
  always @(a or b or cin)
    begin
      {cout,sum} = a + b + cin;
    end
endmodule
```
• Arguments and results of Verilog operator classes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>2 operands</td>
<td>Binary word</td>
</tr>
<tr>
<td>Bitwise</td>
<td>2 operands</td>
<td>Binary word</td>
</tr>
<tr>
<td>Reduction</td>
<td>1 operand</td>
<td>Bit</td>
</tr>
<tr>
<td>Logical</td>
<td>2 operands</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Relational</td>
<td>2 operands</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Shift</td>
<td>1 operand</td>
<td>Binary word</td>
</tr>
<tr>
<td>Conditional</td>
<td>3 operands</td>
<td>Expression</td>
</tr>
</tbody>
</table>
 Operators

- Arithmetic (pair of operands, binary word)
  - [binary: +, -, *, /, %]; [unary: +, -]
- Bitwise (pair of operands, binary word)
  - [~, &, |, ^, ~, ^~]
- Reduction (single operand, bit)
  - [&, &&, |, ~|, ^, ^~]
- Logical (pair of operands, boolean value)
  - [!, &&, ||, ==, !=, ===, !==]
- Relational (pair of operands, boolean value)
  - [<, <=, >, >=]
- Shift (single operand, binary word)
  - [>>, <<]
- Conditional ? : (three operands, expression)
- Concatenation and Replications {,} {int{ }}
Operators

Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
</tbody>
</table>

examples:

A=4'b0011; B=4'b0100; // A and B are register vectors
D=6; E=4; // D and E are integers
A*B // 4'b1100
D/E // 1
A+B // 4'b0111
in1=4'b101x; in2=4'b1010;
sum=in1 + in2; // 4'bx
-10 % 3 = -1 // Take sign of the first operand
14 % -3 = 2
module test;
    reg [3:0] A,B;
    wire [4:0]   sum,diff1,diff2,neg;

    assign sum=A+B;
    assign diff1=A-B;
    assign diff2=B-A;
    assign neg=-A;

initial
begin
    #5 A=5;B=2;
    $display("t_sim A B A+B A-B B-A -A");
    $monitor($time,"%d%d%d%d%d",A,B,sum,
               diff1,diff2,neg);
    #10 $monitoroff;
    $monitor($time,"%b%b%b%b%b",A,B,sum,
             diff1,diff2,neg);
endmodule
Operator

- Bit-wise operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive nor</td>
</tr>
</tbody>
</table>

\[ \sim(101011) = 010100 \]
\[ (010101) \& (001100) = 000100 \]
\[ (010101) \mid (001100) = 011101 \]
\[ (010101) ^ (001100) = 011001 \]
Operator

Reduction operators (1-bit result)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Reduction and</td>
</tr>
<tr>
<td>~&amp;</td>
<td>Reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Reduction exclusive or</td>
</tr>
<tr>
<td><del>^, ^</del></td>
<td>Reduction exclusive nor</td>
</tr>
</tbody>
</table>

&(10101010)=1’b0
|(10101010)=1’b1
&(10x0x0x)=1’b0
|(10x01010)=1’b1
Operator

Logical operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>Logical equality</td>
</tr>
<tr>
<td>!=</td>
<td>Logical inequality</td>
</tr>
<tr>
<td>===</td>
<td>Case equality</td>
</tr>
<tr>
<td>!==</td>
<td>Case inequality</td>
</tr>
</tbody>
</table>

=== : determines whether two words match identically on a bit-by-bit basis, including bits that have values “x” and “z”
## Logical operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Possible logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a==b</code></td>
<td>a equal to b, result unknown if <code>x</code> or <code>z</code> in a or b</td>
<td>0, 1, <code>x</code></td>
</tr>
<tr>
<td><code>a!=b</code></td>
<td>a not equal to b, result unknown if <code>x</code> or <code>z</code> in a or b</td>
<td>0, 1, <code>x</code></td>
</tr>
<tr>
<td><code>a===b</code></td>
<td>a equal to b, including <code>x</code> and <code>z</code></td>
<td>0, 1</td>
</tr>
<tr>
<td><code>a!==b</code></td>
<td>a not equal to b, including <code>x</code> and <code>z</code></td>
<td>0, 1</td>
</tr>
</tbody>
</table>

// A=4, B=3, I=4'b1010, J=4'b1101, K=4'b1xxz, L=4'b1xxz, M=4'b1xxx
A==B // result in logical 0
I!=J // result in logical 1
I==k // result in `x`
K==L // result in logical 1 (all bits match, including `x` and `z`)
K===M // result in logical 0 (least significant bit does not match)
L!==M // result in logical 1
Operator

- Shift operator
  - `>>` logical shift right
  - `<<` logical shift left

```verilog
module shift_reg(out,in);
    output [5:0] out;
    input [5:0] in;
    parameter shift=3;
    assign out=in<<shift;
endmodule
```

```verbatim
reg_in=6'b011100
reg_in<<3   100000
reg_in>>3   000011
```
Operator

- Conditional Operator

Usage: `conditional_expression ? true_expression : false_expression;`

- The action of a conditional operator is similar to a multiplexer
**Operator**

- **Examples of Conditional Operator**
  
  //model functionality of a tristate buffer
  assign addr_bus = drive_enable ? addr_out : 36’bz;

  //model functionality of a 2-to-1 multiplexer
  assign out = control ? in1 : in0;

- Conditional operations can be nested. Each *true_expression* or *false_expression* can itself be a conditional operation
  assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );
Operator

Conditional Operator

module MUX4_1(out,i0,i1,i2,i3,sel);
  output [3:0] out;
  input [3:0] i0,i1,i2,i3;
  input [1:0] sel;

  assign out=(sel==2’b00)?i0:
               (sel==2’b01)?i1:
               (sel==2’b10)?i2:
               (sel==2’b11)?i3:
               4’bx;
endmodule
Operator

Concatenation and Replication Operator

Concatenation operator in LHS

```verilog
module add_32 (co, sum, a, b, ci);
    output co;
    output [31:0] sum;
    input  [31:0] a, b;
    input   ci;
    assign #100 {co, sum} = a + b + ci;
endmodule
```

Bit replication to produce 01010101

```verilog
assign byte = {4{2'`b01}};
```

Sign Extension

```verilog
assign word = {{8{byte[7]}}, byte};
```
Expression Bit Widths

- Depends on:
  - widths of operands and
  - types of operators
- Verilog fills in smaller-width operands by using zero extension.
- Final or intermediate result width may increase expression width
- Unsized constant number - same as integer (usually 32bit)
- Sized constant number - as specified
- $x \text{ op } y$ where op is $+, -, *, /, \%, \&, \|, ^, ^\sim$:
  - Arithmetic binary and bitwise
  - Bit width = max (width($x$), width($y$))
Expression Bit Widths (continued)

- **op x where op is +, -**
  - Arithmetic unary
  - Bit width = width(x)
- **op x where op is ~**
  - Bitwise negation
  - Bit width = width(x)
- **x op y where op is ==, !==, ===, !===, &&, ||, >, >=, <, <= or op y where op is !, &, |, ^, ~&, ~|, ~^**
  - Logical, relational and reduction
  - Bit width = 1
- **x op y where op is <<, >>**
  - Shift
  - Bit width = width(x)
Expression Bit Widths (continued)

- $x ? y : z$
  - Conditional
  - Bit width = $\text{max}(\text{width}(y), \text{width}(z))$

- $\{x, \ldots, y\}$
  - Concatenation
  - Bit width = $\text{width}(x) + \ldots + \text{width}(y)$

- $\{x\{y, \ldots, z\}\}$
  - Replication
  - Bit width = $x \ast (\text{width}(y) + \ldots + \text{width}(z))$
Expressions with Operands
Containing x or z

- **Arithmetic**
  - If any bit is x or z, result is all x’s.
  - Divide by 0 produces all x’s.

- **Relational**
  - If any bit is x or z, result is x.

- **Logical**
  - == and != If any bit is x or z, result is x.
  - === and !== All bits including x and z values must match for equality
Expressions with Operands Containing x or z (cont’d)

- **Bitwise**
  - Defined by tables for 0, 1, x, z operands.

- **Reduction**
  - Defined by tables as for bitwise operators.

- **Shifts**
  - z changed to x. Vacated positions zero filled.

- **Conditional**
  - If conditional expression is ambiguous (e.g., x or z), both expressions are evaluated and bitwise combined as follows: $f(1,1) = 1$, $f(0,0) = 0$, otherwise x.
Procedures and Assignments

- Verilog procedures
  - initial and always statements
  - UDP
  - tasks
  - functions

- Sequential block: a group of statements that appear between a `begin` and an `end`
  - executed sequentially
  - considered as a statement – can be nested

- Procedures execute concurrently with other procedures

- Assignment statements
  - continuous assignments: appear outside procedures
  - procedural assignments: appear inside procedures
Assignments

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to nets
  - procedural assignment, which assigns values to registers
- Basic form

\[
<\text{left hand side}> = <\text{right hand side}>
\]

<table>
<thead>
<tr>
<th>Assignments</th>
<th>Left Hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous Assignment</td>
<td>Net</td>
</tr>
<tr>
<td></td>
<td>\text{wire, tri}</td>
</tr>
<tr>
<td>Procedural Assignment</td>
<td>\text{Register}</td>
</tr>
<tr>
<td></td>
<td>\text{reg, integer, real}</td>
</tr>
</tbody>
</table>

Left Hand Side = \text{LHS}
Right Hand Side = \text{RHS}
Assignments

- **Continuous assignment**

```verilog
cmodule holiday_1(sat, sun, weekend);
  input sat, sun; output weekend;
  assign weekend = sat | sun;     // outside a procedure
endmodule
```

- **Procedural assignment**

```verilog
cmodule holiday_2(sat, sun, weekend);
  input sat, sun; output weekend; reg weekend;
  always #1 weekend = sat | sun;     // inside a procedure
endmodule
```

```verilog
cmodule assignments
  // continuous assignments go here
always begin
  // procedural assignments go here
end
endmodule
```
Continuous Assignments

- The LHS of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
- Continuous assignments are always active. Any changes in RHS of the continuous assignment are evaluated and the LHS is updated.
- The operands on the RHS can be registers or nets or function calls. Register or nets can be scalars or vectors.
- Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates.
Continuous Assignment

- Drive a value onto a wire, wand, wor, or tri
  - Use an explicit continuous assignment statement after declaration
  - Specify the continuous assignment statement in the same line as the declaration for a wire
- Used for datapath descriptions
- Used to model combinational circuits
Continuous Assignments

- Convenient for logical or datapath specifications

```verilog
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;
assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be \(a+b+\text{carryin}\)

Recomputed when \(a\), \(b\), or \(\text{carryin}\) changes
Continuous Assignments

module assignment_1();
wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire;

assign pwr_stable = Ok & (!Fire); 
assign pwr_on = 1;
assign pwr_good = pwr_on & pwr_stable;

initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end 
initial begin $monitor("TIME=%0d"," ON"," STABLE"," OK"," FIRE"," GOOD"," pwr_on"," pwr_stable"," Ok"," Fire"," pwr_good);
   #10 $finish; end 
endmodule

>>> 
TIME=0 ON=1 STABLE=0 OK=0 FIRE=0 GOOD=0
TIME=1 ON=1 STABLE=1 OK=1 FIRE=0 GOOD=1
TIME=6 ON=1 STABLE=0 OK=1 FIRE=1 GOOD=0
Continuous Assignments

- Continuous assignments provide a way to model combinational logic

### continuous assignment

```verilog
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    assign out =~ in;
endmodule
```

### gate-level modeling

```verilog
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    not U1(out[0],in[0]);
    not U2(out[1],in[1]);
    ..
    not U31(out[31],in[31]);
endmodule
```
Continuous Assignments

Examples of continuous Assignment

```verilog
assign out = i1 & i2;
    // i1 and i2 are nets
assign addr[15:0] = addr1[15:0] ^ addr2[15:0]
    // Continuous assign for vector nets addr is a 16-bit vector net
    // addr1 and addr2 are 16-bit vector registers
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
    // LHS is a concatenation of a scalar net and vector net
```

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.

```verilog
wire a;    --declare
assign a = b & c;  --assign
wire a = b & c;  --declare and assign
```
Assignment Delays

- Regular Assignment Delay
  ```
  wire out;
  assign #10 out = in1 & in2;  // delay in a continuous assign
  ```

- Implicit Continuous Assignment Delay
  ```
  wire #10 out = in1 & in2;
  ```

- Net Declaration Delay
  ```
  wire # 10 out; // net delays
  assign out = in1 & in2;
  ```
Continuous Assignment

- Avoid logic loop
  - HDL Compiler and Design Compiler will automatically open up asynchronous logic loops
  - Without disabling the combinational feedback loop, the static timing analyzer can’t resolve
- Example
Procedural Assignments

- 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)
    {carry, sum} = a + b + ci;
endmodule
```
Procedural Assignments

- Inside an initial or always block:
  ```
  initial
  begin
  {cout, sum} = a + b + cin;
  end
  ```
  ```
  always
  begin
  {cout, sum} = a + b + cin;
  end
  ```

- Just like in C: RHS evaluated and assigned to LHS before next statement executes.

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

- LHS must be a reg:
  - Primitives or cont. assignment may 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'b0010;  r2 = 4'b1001;
            sum2 = r1 + r2;
            $display("r1 r2 sum1 sum2");
            $monitorb(r1, r2, sum1, sum2);
            #10 r1 = 4'b0011;
        end
endmodule

Result

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>sum1</th>
<th>sum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>1001</td>
<td>01011</td>
<td>01011</td>
</tr>
<tr>
<td>0011</td>
<td>1001</td>
<td>01100</td>
<td>01011</td>
</tr>
</tbody>
</table>
Procedural Assignments

- Major difference between continuous assignment and procedural assignment
  - the LHS are updated in different time
    - in continuous assignment, LHS are updated whenever the RHS changes value
    - in procedural assignment, LHS are updated when this statement is encountered
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 @(posedge clk) blocks run in some undefined sequence.
Procedural Assignments

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

```verbatim
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;
    time
    statements executed
    0 x=0; y=1; z=1;
    count=0;
    rega=8'b0; regb=rega;
    15 rega[2]=1'b1;
    regb=rega=0;
    25 regb[7:5]={x, y, z};
    count=count+1;
end
```
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
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;
  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; count=count+1;</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>

```
A Flawed Shift Register

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 which priority
Non-blocking Assignments

- This version does work:

```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 Can Behave Oddly

- A sequence of nonblocking assignments don’t communicate

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

Blocking assignment:  
Nonblocking assignment:

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

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

```
\[ a = 1; \quad b = a; \quad c = b; \]
\[ a \leq 1; \quad b \leq a; \quad c \leq b; \]
```
Non-blocking statements to Eliminate race conditions

// Illustration 1: blocking
always @(posedge clk)
begin
  a = b;
  b = a;
end
// Illustration 2: nonblocking
always @(posedge clk)
begin
  a <= b;
  b <= a;
end

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

- Application of non-blocking assignments

```verilog
class 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
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>

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

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

- Run until they encounter a delay

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

- or a wait for an event

```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=" , Clk,"  Y=" , Y);
initial #70 $finish;
endmodule

>>>>
T  =  0  Clk=0  Y=0
T=10  Clk=1  Y=0
T=15  Clk=1  Y=1
T=20  Clk=0  Y=1
T=30  Clk=1  Y=1
T=35  Clk=1  Y=0
T=40  Clk=0  Y=0
T=50  Clk=1  Y=0
T=55  Clk=1  Y=1
T=60  Clk=0  Y=1
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

```vhdl
//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
```
Conditional Statements

- If and If-else statements

```plaintext
if (expression)
  statement
else
  statement

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

If and If-Else Statements (cont.)

- Examples

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

```vhdl
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

```plaintext
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  default: default_statement;
endcase
```

```plaintext
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 be represented by `?`. In that position.
  - `casex` treats all `x` and `z` values in the case item or the case expression as don’t cares.

```verbatim
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
```

```plaintext
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=%d",count);
    count=count+1;
  end
end
```

```
initial begin
  reg [7:0] tempreg;
  count = 0;
  tempreg = reg;
  while (tempreg) begin
    $display("count=%d",count);
    count=count+1;
    if (tempreg[0]) count = count + 1;
    tempreg = tempreg >> 1;
  end
end
```

<table>
<thead>
<tr>
<th>rega</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 contains 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

```
//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
```
The keyword `repeat` is used for this loop. The `repeat` construct executes the loop a **fixed** number of times.

```verilog
module multiplier(result, op_a, op_b);

...  // remainder of the function

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
```
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.

//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;
Modeling A Flip-Flop With Always

- Very basic: an edge-sensitive flip-flop

```verilog
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
Block Disable

Disabling named blocks (example: comparator)

```verilog
module comparator(a, b, a_gt_b, a_lt_b, a_eq_b);
  parameter size = 2;
  input [size:1] a, b;
  output a_gt_b, a_lt_b, a_eq_b;
  reg a_gt_b, a.lt_b, a_eq_b;
  integer k;

  always @(a or b) begin: compare_loop
    for (k = size; k > 0; k = k - 1) begin
      if (a[k] != b[k]) begin
        a_gt_b = a[k];
        a_lt_b = ~a[k];
        a_eq_b = 0;
        disable compare_loop;
      end
    end
    a_gt_b = 0;
    a_lt_b = 0;
    a_eq_b = 1;
  end
endmodule
```

Behavioral Models

- Behavioral models are abstract descriptions of functionality.
- Widely used for quick development of model
- Follow by synthesis
- We'll consider two types:
  - Continuous assignment (Boolean equations)
  - Cyclic behavior (more general, e.g. algorithms)
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;
y=#5 x+z;
y=#tempxz;
```
Timing Controls

- Event-Based Timing Control
  - Regular Event Control

```verilog
module 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=d;
        @(posedge clk) q=4;
        @(negedge clk) q=1;
        q=@(posedge clk) 3;
        #10 $finish;
        clk=0;
    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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q[2:0]</td>
<td></td>
<td></td>
<td></td>
<td></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;
  ```
Timing Controls

Level-Sensitive Timing Control

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

Function & Task
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

Tasks example2
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.
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

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
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;
endtask
endmodule
### Differences Between Tasks and Functions

#### Tasks and Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A function can enable another function but not another task</td>
<td>A task can enable other tasks and functions</td>
</tr>
<tr>
<td>Function always execute in 0 simulation time</td>
<td>Tasks may execute in non-zero simulation time</td>
</tr>
<tr>
<td>Functions must not contain any delay, event, or timing control statements</td>
<td>Tasks may contain delay, event, or timing control statements</td>
</tr>
<tr>
<td>Functions must have <strong>at least one input</strong> argument. They can have more than one input</td>
<td>Task may have <strong>zero or more</strong> arguments of type <strong>input, output or inout</strong></td>
</tr>
<tr>
<td>Functions always return a single value. They <strong>cannot</strong> have <strong>output or inout</strong> arguments</td>
<td>Tasks do not return with a value but can pass multiple values through <strong>output and inout</strong> arguments</td>
</tr>
</tbody>
</table>
Differences Between Tasks and Functions

- Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments.
- Functions are used when common provides exactly one output. Functions are typically used for conversions and commonly used calculations.
- Tasks can have `input`, `output`, and `inout` ports; functions can have `input` ports.
- Tasks and functions have local variables, registers, time variables, integers, real, or events.
Homework #2

- Try to build your design by using RTL code
- Functional simulation

Design examples:

- 32-bits Adder/Subtractor (sign, unsign)
- 8-bits multiplier (sign, unsign)
- 8-bits divider/square root
- Up/Down Counter (binary/Gray)
- …