Behavioral Descriptions
Behaviors (Processes)

- Two constructs
  - **initial**: one-shot activity flow (not synthesizable but good for testbenches)
  - **always**: cyclic (repetitive) activity flow
- Use procedural statements that assign values to register variables (exception: `force...release`)
Behaviors

- Continuous assignments and primitives assign outputs whenever there are events on inputs.
- Behaviors assign values when an assignment statement in the activity flow executes. (Input events on the RHS do not initiate activity – control must be passed to the statement.)
Behaviors

- Body may consist of a single statement or a block statement
- Block statement begins with `begin` and ends with `end`
- Behaviors are an elaborate form of continuous assignments or primitives but operate on registers rather than nets (exception: `force...release`)
initial

- Run once

```plaintext
... 
initial
begin
    sig_a = 0;
    sig_b = 1;
    sig_c = 0;
end
...
```

// An “initial” behavior
// Procedural assignments
// execute sequentially.
always

- Infinite loop until simulation stops
  ...
  initial
  clock = 0;

always
  #10 clock = ~clock;

initial
  #100 $finish;
...

...
Structural vs. Behavioral Descriptions

module my_module(...);

...
assign ...; // continuous assignment
and (...); // instantiation of a primitive
adder_16 M(...); // instantiation of a module

always @ (...) 
begin ... end

initial
begin ... end

endmodule
Procedural Assignment

- Assign value to registers
- Blocking procedural assignment
  - Use “=”
  - An assignment is completed before the next assignment starts
    
    \[
    a = 0; \quad a = 1; \quad c = a; \quad // \quad c = 1
    \]

- Non-blocking procedural assignment
  - Use “\(<=\)”
  - Assignments are executed in parallel
    
    \[
    a = 0; \quad a <= 1; \quad c = a; \quad // \quad c = 0
    \]
    
    \[
    d <= 0; \quad d <= 1; \quad // \quad d = 0 \text{ or } 1?
    \]
Examples

\[
\begin{align*}
a &= 1; \\
b &= 0; \\
\ldots \\
a &\leq b; & \text{// Use } b = 0 \\
b &\leq a; & \text{// Use } a = 1 \\
\end{align*}
\]
Examples

\begin{align*}
\text{a} &= 1; \\
\text{b} &= 0; \\
\ldots \\
\text{a} &= \text{b}; & \text{a} &= 1; \\
\text{b} &= \text{a}; & \text{b} &= 0; \\
\text{b} &= \text{a}; & \ldots \\
\text{a} &= \text{b}; & \text{b} &= \text{a}; \\
\end{align*}

// Use b=0
// Use a=0
// Use a=1
// Use b=1
Procedural Assignments – Some Rules

- A register (net) variable can be referenced anywhere in a module
- A register variable can be assigned only within a procedural assignment, task or function
- A register variable cannot be input or inout
- A net variable may not be assigned within a behavior, task or function (exception: force…release)
- A net variable within a module must be driven by a primitive, continuous assignment, force…release, or module port
Procedural Continuous Assignment (PCA)

- **assign ... deassign**
  - Assign value to register
  - Dynamic binding to target register
  - Override all procedural assignments to target register
  - Binding can not be removed until **deassign** or a new PCA is established
  - **deassign** is optional
  - Synthesis tools might not support it
  - Can be used to model level-sensitive behavior of combinational logic, transparent latches, and asynchronous control of sequential logic
Example

module mux4_PCA (a, b, c, d, select, y_out);
  input a, b, c, d;
  input [1:0] select;
  output y_out;
  reg y_out;

  always @ (select)
    if (select == 0) assign y_out = a; else
    if (select == 1) assign y_out = b; else
    if (select == 2) assign y_out = c; else
    if (select == 3) assign y_out = d; else
      assign y_out = 1’bx;
endmodule
Alternative

module mux4_PCA (a, b, c, d, select, y_out);
  input a, b, c, d;
  input [1:0] select;
  output y_out;
  reg y_out;

always @ (select or a or b or c or d)
  if (select == 0) y_out = a; else
  if (select == 1) y_out = b; else
  if (select == 2) y_out = c; else
  if (select == 3) y_out = d; else
    y_out = 1’bx;
endmodule
Example

module Flop_PCA(preset, clear, q, qbar, clock, data);
input preset, clear, clock, data;
output q, qbar;
reg q;

assign qbar = ~q;

always @ (negedge clock)
  q = data;

always @ (clear or preset)
  begin
    if (!clear) assign q = 0;
    else if (!preset) assign q = 1;
    else deassign q;
  end

endmodule
Procedural Continuous Assignment (PCA)

- **force … release**
  - Assign value to net or register
  - Dynamic binding to target net or register
  - Overwrite primitive and continuous assignment to a net, and override procedural assignment and `assign … deassign` to a register
  - Binding can not be removed until `release`
  - Synthesis tools might not support it
  - Can be used with hierarchical de-referencing in testbenches
Example

...  

```plaintext
for sig_a = 1;
for sig_b = 1;
for sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
// Insert code to construct tests
release sig_a;
release sig_b;
release sig_c;
```
## Assignment Modes

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Output of primitive</th>
<th>Continuous assignment</th>
<th>Procedural assignment</th>
<th>assign ... deassign</th>
<th>force ... release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Register</td>
<td>Comb - No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Seq - Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Procedural Timing Controls

- **Mechanisms**
  - delay control operator (#)
  - event control operator (@)
  - event *or*
  - named events
  - *wait* construct
Delay Control Operator (#)

- Suspend the activity flow at the location of the operator
- Examples:

```plaintext
... always begin
    #0 clock = 0;
    #50 clock = 1;
    #50;
end
...
... always begin
    #clock_period/2;
    clock = ~clock;
end
...
```
Event Control Operator (@)

- Synchronize execution to an event

Example 1:

```plaintext
... @ signal_1 a=b;
...
```

Example 2:

```plaintext
... @ (event_a) begin
  ...
  @ (event_b) begin
    ...
    end
  end
end
...
```
posedge and negedge

- **posedge**: $0 \rightarrow 1$, $0 \rightarrow x$, $x \rightarrow 1$
- **negedge**: $1 \rightarrow 0$, $1 \rightarrow x$, $x \rightarrow 0$

**Example:**

- `always @ (posedge clock) #10 b = a;`
The above behavior does not correctly model a positive edge-triggered D flip-flop with asynchronous (active-low) set and reset.
Fix

... 

always @(negedge set or negedge reset or posedge clk)
begin
  if (reset == 0) q = 0;
  else if (set == 0) q = 1;
  else if (clk == 1) q = data;
end

...
Named Events

- Synchronization within and between modules
- A named event can be declared only in a module, with keyword `event`; it can then be referenced within that module directly, or in other modules by hierarchically de-referencing the name of the event.
- The occurrence of an event is explicitly determined by a procedural statement using the event-trigger operator, “->”.

Example: …

```
    event up_edge;

    always @(posedge clk)
        -> up_edge;

    always @(up_edge)
        ...

    ...
```

```
Example

module top (clk, data, q);
input clk, data;
output q;

talker M1 (clk);
receiver M2 (q, data);
endmodule

module talker (clk);
input clk;
event do_it;
always @ (posedge clk)
  -> do_it;
endmodule

module receiver (q, data);
input data;
output q;
reg q;
always @ (top.M1.do_it)
  q = data;
endmodule
wait

- Suspend activity flow until expression following `wait` is true
- Can model level-sensitive behavior
- Example:
  ```
  wait (enable) a = b;
  ```
Intra-Assignment Delay

- Postpone assignment, not evaluation
- Examples (blocking assignments)

\[
\begin{align*}
\text{...} & \hspace{2cm} \text{...} \\
\text{a = } & \#5 \text{ b;} \hspace{1cm} \text{e = @} (\text{bus}) \text{ f;} \\
\text{c = d;} & \hspace{1cm} \text{g = h;} \\
\text{...} & \hspace{1cm} \text{...}
\end{align*}
\]
Intra-Assignment Delay

- Examples (non-blocking assignments)

\[
\begin{align*}
\text{initial begin} & \quad \text{always begin} \\
@ (\text{posedge clk}) & \quad @ (\text{posedge clock}) \\
g & \leftarrow @(\text{bus}) k; & \quad g & \leftarrow @(\text{bus}) k; \\
p & \leftarrow q; & \quad \text{end} \\
\text{end} & \quad \text{end} \\
\text{...} & \quad \text{...}
\end{align*}
\]
Blocking vs. Non-Blocking Assignments

// blocking
a = #10 1;
b = #2 0;
c = #3 1;

// non-blocking
a <= #10 1;
b <= #2 0;
c <= #3 1;
Examples

assign y = a | b;

always @(a or b)
y = a | b;

assign #5 y = a | b;

always @(a or b)
#5 y = a | b;

always @(a or b)
y = #5 a | b;

always @(a or b)
y <= #5 a | b;
Repeated Intra-Assignment Delay

\[
\text{reg}_a = \text{repeat} \ (5) \ \@ \ (\text{nedge} \ \text{clock}) \ \text{reg}_b;
\]

\[
\begin{align*}
\text{begin} & \\
\text{temp} & = \text{reg}_b; \\
\@ & \ (\text{nedge} \ \text{clock}); \\
\@ & \ (\text{nedge} \ \text{clock}); \\
\@ & \ (\text{nedge} \ \text{clock}); \\
\@ & \ (\text{nedge} \ \text{clock}); \\
\@ & \ (\text{nedge} \ \text{clock}); \\
\text{reg}_a & = \text{temp}; \\
\text{end}
\end{align*}
\]
Example

module repeater;
  reg clock;
  reg reg_a, reg_b;

initial
  clock = 0;

initial begin
  #5 reg_a = 1;
  #10 reg_a = 0;
  #5 reg_a = 1;
  #20 reg_a = 0;
end

always
  #5 clock = ~ clock;

initial
  #100 $finish;
initial
  begin
    #10 reg_b = repeat (5) @ (posedge clock) reg_a;
  end
endmodule

Figure 7.20 Waveforms for repeated intra-assignment delay.
Simulation of Procedural Assignments

- At a given time step,
  - evaluate expressions on RHS
  - execute blocking assignments
  - execute non-blocking assignments that do not have intra-assignment timing control
  - execute past assignments which have been scheduled to execute in the current time step
  - execute $\text{monitor}$ (note: $\text{display}$ is executed when it is encountered, but is executed before non-blocking assignments)
  - Advance simulation time
$display$ vs. $monitor$

\[
\text{initial begin} \\
\quad a = 1; \\
\quad b = 0; \\
\quad a \leq b; \\
\quad b \leq a; \\
\quad \text{$display$ (“a=\%b b=\%b”, a, b);} \\
\text{end}
\]

\[
\text{initial begin} \\
\quad a = 1; \\
\quad b = 0; \\
\quad a \leq b; \\
\quad b \leq a; \\
\quad \text{$monitor$ (“a=\%b b=\%b”, a, b);} \\
\text{end}
\]

display: a=1 b=0

monitor: a=0 b=1
Indeterminate Assignments

module ambiguity();
    reg a, b, c;

    ...
    always @ (a)
        c = a;

    always @(a)
        c = b;

    ...
endmodule
No Ambiguity

module example;
    reg wave;
    reg [2:0] i;

    initial
        begin
            for (i=0; i<=5; i=i+1)
                wave <= #(i*10) i[0];
        end
endmodule