Overview

- Simulation and Synthesis
- Modules and Primitives
- Styles
- Structural Descriptions
- Language Conventions
- Data Types
- Delay
- Behavioral Constructs
- Compiler Directives
- Simulation and Testbenches
Simulation and Synthesis

- Simulation tools typically accept full set of Verilog language constructs
- Some language constructs and their use in a Verilog description make simulation efficient and are ignored by synthesis tools
- Synthesis tools typically accept only a subset of the full Verilog language constructs
  - In this presentation, Verilog language constructs not supported in Synopsys FPGA Express are in *red italics*
  - There are other restrictions not detailed here, see [2].

Modules

- The Module Concept
  - Basic design unit
  - Modules are:
    - Declared
    - Instantiated
  - Modules declarations cannot be nested
Module Declaration (FIO*)

- Syntax

module_declaration ::= module_keyword module_identifier [list of ports]; 
{module_item} endmodule

module_keyword ::= module | macromodule
list_of_ports ::= (port {, port})

* For Information Only – not to be covered in presentation

Module Declaration (FIO)

- Syntax (continued)

module_item ::= module_item_declaration | parameter_override | continuous_assign 
| gate_instantiation | udp_instantiation | module_instantiation | 
specify_block | initial_construct | always_construct

module_item_declaration ::= parameter_declaration | input_declaration | output_declaration | 
inout_declaration | net_declaration | reg_declaration | 
integer_declaration | real_declaration | time_declaration | 
realtime_declaration | event_declaration | task_declaration | 
function_declaration
parameter_override ::= defparam list_of_parameter_assignments
udp_declaration

1/28/2001 5

1/28/2001 6
Module Declaration

- Annotated Example

```verbatim
/* module_keyword module_identifier (list of ports) */
module C_2_4_decoder_with_enable (A[1:0], E_n, D);
input [1:0] A;
input E_n;
output [3:0] D;
assign D = {4{~E_n}} & 
(A == 2'b00) ? 4'b0001 :
(A == 2'b01) ? 4'b0010 :
(A == 2'b10) ? 4'b0100 :
(A == 2'b11) ? 4'b1000 :
4'bxxxx) ; // continuous_assign
endmodule
```

Module Declaration

- Identifiers - must not be keywords!
- Ports
  - First example of signals
  - Scalar: e. g., E_n
  - Vector: e. g., A[1:0], A[0:1], D[3:0], and D[0:3]
    - Range is MSB to LSB
    - Can refer to partial ranges - D[2:1]
  - Type: defined by keywords
    - input
    - output
    - inout (bi-directional)
Module Instantiation

- Example

```verilog
module C_4_16_decoder_with_enable (A, E_n, D);

input [3:0] A;
input E_n;
output [15:0] D;

wire [3:0] S;
wire [3:0] S_n;
C_2_4_decoder_with_enable DE (A[3:2], E_n, S);
not N0 (S_n, S);
C_2_4_decoder_with_enable D0 (A[1:0], S_n[0], D[3:0]);
C_2_4_decoder_with_enable D1 (A[1:0], S_n[1], D[7:4]);
C_2_4_decoder_with_enable D2 (A[1:0], S_n[2], D[11:8]);
C_2_4_decoder_with_enable D3 (A[1:0], S_n[3], D[15:12]);
endmodule
```

Module Instantiation (FIO)

- Syntax

```verilog
module_instantiation ::= 
module_identifier [parameter_value_assignment] module_instance {,
    module_instance};
parameter_value_assignment ::= # (expression {, expression})
module_instance ::= name_of_instance ([list_of_module_connections])
name_of_instance ::= module_instance_identifier [range]
list_of_module_connections ::= 
    ordered_port_connection {, ordered_port_connection} 
    | named_port_connection {, named_port_connection}
ordered_port_connection ::= [expression]
named_port_connection ::= . port_identifier ([expression])
```
Module Instantiation

- More Examples
  - Single module instantiation for five module instances
    
    \[
    \begin{align*}
    &C_{\text{2.4 decoder with enable}} \ DE (A[3:2], \ E_n, \ S), \\
    &D0 (A[1:0], \ S_n[0], \ D[3:0]), \\
    &D1 (A[1:0], \ S_n[1], \ D[7:4]), \\
    &D2 (A[1:0], \ S_n[2], \ D[11:8]), \\
    &D3 (A[1:0], \ S_n[3], \ D[15:12]);
    \end{align*}
    \]

  - Named_port connection
    
    \[
    C_{\text{2.4 decoder with enable}} \ DE (.E_n (E_n), .A (A[3:2]) .D (S)); \\
    // Note order in list no longer important (E_n and A interchanged).
    \]

Primitives

- Gate Level
  - and, nand
  - or, nor
  - xor, xnor
  - buf, not
  - bufif0, bufif1, notif0, notif1 (three-state)

- Switch Level
  - *mos where * is n, p, c, rn, rp, rc; pullup, pulldown;
  - *tran where * is (null), r and * (null), if0, if1 with both * and * not (null)
**Primitives**

- No declaration; can only be instantiated
- All output ports appear in list before any input ports
- Optional drive strength, delay, name of instance
- Example: `and N25 (Z, A, B, C); // instance name`
- Example: `and #10 (Z, A, B, X); // delay (X, C, D, E); // delay`
  /*Usually better to provide instance name for debugging.*/

- Example: `or N30 (SET, Q1, AB, N5),`
- `N41 (N25, ABC, R1);`

---

**Styles**

- Structural - instantiation of primitives and modules
- RTL/Dataflow - continuous assignments
- Behavioral - procedural assignments
**Style Example - Structural**

```verilog
module full_add (A, B, CI, S, CO);
input A, B, CI;
output S, CO;
wire N1, N2, N3;
half_add HA1 (A, B, N1, N2), HA2 (N1, CI, S, N3);
or P1 (CO, N3, N2);
endmodule
```

```verilog
module half_add (X, Y, S, C);
input X, Y;
output S, C;
xor (S, X, Y);
and (C, X, Y);
endmodule
```

**Style Example - RTL/Dataflow**

```verilog
module fa_rtl (A, B, CI, S, CO);
input A, B, CI;
output S, CO;
assign S = A ^ B ^ CI; //continuous assignment
assign CO = A & B | A & CI | B & CI; //continuous assignment
endmodule
```

1/28/2001
Style Example - Behavioral

module fa_bhv (A, B, CI, S, CO);

input A, B, CI;
output S, CO;

reg S, CO; // required to “hold” values between events.

always@(A or B or CI);
begin
  S <= A ^ B ^ CI; // procedural assignment
  CO <= A & B | A & CI | B & CI; // procedural assignment
end
endmodule

Structural Descriptions

- Textual description of schematic
- Form of netlist
- Connections
- Hierarchy
- Arrays of instances
- Hierarchy established by instantiation of modules and primitives within modules
Connections

- **By position association**
  - module C_2_4_decoder_with_enable (A, E_n, D);
  - C_4_16_decoder_with_enable DX (X[3:2], W_n, word);
  - A = X[3:2], E_n = W_n, D = word

- **By name association**
  - module C_2_4_decoder_with_enable (A, E_n, D);
  - C_2_4_decoder_with_enable DX (.E_n(W_n), .A(X[3:2]), .D(word));
  - A = X[3:2], E_n = W_n, D = word

Connections

- **Empty Port Connections**
  - module C_2_4_decoder_with_enable (A, E_n, D);
  - C_2_4_decoder_with_enable DX (X[3:2], , word);
  - E_n is at high-impedance state (z)
  - C_2_4_decoder_with_enable DX (X[3:2], W_n ,);
Arrays of Instances

- `{ , }` is concatenate
- Example

```verilog
module add_array (A, B, CIN, S, COUT);
  input [7:0] A, B;
  input CIN;
  output [7:0] S;
  output COUT;
  wire [7:1] carry;
  full_add FA[7:0] (A,B,{carry, CIN},S,{COUT, carry});
  // full_add is a module
endmodule
```

Language Conventions

- Case-sensitivity
  - Verilog is case-sensitive.
  - Some simulators are case-insensitive
  - Advice: Don't use case-sensitive feature!
  - Keywords are lower case
- Different names must be used for different items within the same scope
- Identifier alphabet:
  - Upper and lower case alphabeticals
  - decimal digits
  - underscore
Language Conventions

- Maximum of 1024 characters in identifier
- First character not a digit
- Statement terminated by ;
- Free format within statement except for within quotes
- Comments:
  - All characters after // in a line are treated as a comment
  - Multi-line comments begin with /* and end with */
- Compiler directives begin with // synopsys
- Built-in system tasks or functions begin with $
- Strings enclosed in double quotes and must be on a single line

Logic Values

- Verilog signal values
  - 0 - Logical 0 or FALSE
  - 1 - Logical 1 or TRUE
  - x, X - Unknown logic value
  - z, Z - High impedance condition
- Also may have associated strength for switch level modeling of MOS devices
  - 7 signal strengths plus 3 charge strengths
Number Representation

- Format: `<size><base_format><number>`
  - `<size>` - decimal specification of number of bits
    - default is unsized and machine-dependent but at least 32 bits
  - `<base format>` - `'` followed by arithmetic base of number
    - `<d>` `<D>` - decimal - default base if no `<base_format> given
    - `<h>` `<H>` - hexadecimal
    - `<o>` `<O>` - octal
    - `<b>` `<B>` - binary
  - `<number>` - value given in base of `<base_format>`
    - `_` can be used for reading clarity
    - If first character of sized, binary number is 0, 1, the value is 0-filled up to size. If x or z, value is extended using x or z, respectively.

Examples:
- `6'b010_111` gives 010111
- `8'b0110` gives 00000110
- `8'b1110` gives 00001110
- `4'bx01` gives xx01
- `16'H3AB` gives 0000001110101011
- `24` gives 0...0011000
- `5'O36` gives 11100
- `16'Hx` gives xxxxxxxxxxxxxxxxx
- `8'hz` gives zzzzzzzz
**Variables**

- **Nets**
  - Used for structural connectivity

- **Registers**
  - Abstraction of storage (May or may not be real physical storage)

- **Properties of Both**
  - Informally called signals
  - May be either scalar or vector

**Data Types - Nets - Semantics**

- **wire** - connectivity only; no logical
- **tri** - same as wire, but will be 3-stated in hardware
- **wand** - multiple drivers - wired and
- **wor** - multiple drivers - wired or
- **triand** - same as wand, but 3-state
- **trior** - same as wor but 3-state
- **supply 0** - Global net GND
- **supply 1** - Global Net VCC (VDD)
- **tri0, tri 1, trireg**
Data Types - Nets - Syntax

- Net declaration ::= 
  net type [vectored | scalared] [range] [delay3] list_of_net_identifiers;
  | trireg [vectored | scalared] [charge strength] [range] [delay3]
  list_of_net_identifiers;
  | net type [vectored | scalared] [drive strength] [range] [delay 3]
  list_of_net_decl_assignments;

- **V**ectored - multiple-bit net treated as a single object - cannot reference individual bits or part-select
- **S**calared - bits can be referenced individually or be part selected
  ⊗ Value implicitly assigned by connection to primitive or module output

Net Examples - Single Driver

- wire x;
- wire x, y;
- wire [15:0] data, address;
- wire vectored [0:7] control;
- data[15] (instantiation)
- address[15:8] (instantiation)
- wire address = offset + index;
Net Examples - Multiple Drivers and Constant Nets

- `wor interrupt_1, interrupt_2;`
- `tri [31:0] data_bus, operand_bus;`

Initial Value & Undeclared Nets

- Initial value of a net
  - At t_{sim} = 0, initial value is x.
- Undeclared Nets - Default type
  - Not explicitly declared default to wire
  - default_nettype compiler directive can specify others except for supply0 and supply1
Data Types - Register Semantics

- **reg** - stores a logic value
- **integer** – stores values which are not to be stored in hardware
  - Defaults to simulation computer register length or 32 bits whichever is larger
  - No ranges or arrays supported
  - May yield excess hardware if value needs to be stored in hardware; in such a case, use sized reg.
- **time** - stores time 64-bit unsigned
- **real** - stores values as real num
- **realtime** - stores time values as real numbers

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 by assigned value by:
  - a primitive gate output or
  - a continuous assignment
Register Examples

- reg a, b, c;
- reg [15:0] counter, shift_reg;
- integer sum, difference;

Strings

- No explicit data type
- Must be stored in reg (or array)
- reg [255:0] buffer; //stores 32 characters
Constants

- Declaration of parameters
  - parameter A = 2'b00, B = 2'b01, C = 2'b10;
  - parameter regsize = 8;
    - reg [regsize - 1:0]; /* illustrates use of parameter regsize */

Operators

- Arithmetic (binary: +, -, *, /, %, %); (unary: +, -)
- Bitwise (~, &, |, ^, ~^, ~~)
- Reduction (&, ~& |, ~|, ^, ~^, ~~)
- Logical (!, &&, ||, ==, !=, ===, !==)
- Relational (<, <=, >, >=)
- Shift (>>, <<)
- Conditional ??
- Concatenation and Replications {,} {int{ }}
  * unsupported for variables
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

Expression Bit Widths

- Unsized constant number- same as integer (usually 32)
- Sized constant number - as specified
- x op y where op is +, -, *, /, %, &, |, ^, ^~:
  - 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)
  - Carry can be captured if final result width > width(x)

- op x where op is ~
  - Bitwise negation
  - Bit width = width(x)

Expression Bit Widths (continued)

- 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 = max(width(y), width(z))
- $\{x, ..., y\}$
  - Concatenation
  - Bit width = width(x) + ... + width(y)
- $\{x\{y, ..., z\}\}$
  - Replication
  - Bit width = x * (width(y) + ... + 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
  - $== \text{ and } !=$ If any bit is x or z, result is x.
  - $=== \text{ and } !==$ All bits including x and z values must match for equality
Expressions with Operands Containing x or z

- 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.

Synthesis from Verilog

- Note use of reg in behavioral descriptions; does not always imply actual storage such as latches or registers in synthesis results.
- Procedural statements are executed sequentially.
Delay Uses and Types

- Ignored by FPGA Express; may be useful for simulation
- Uses
  - Behavioral (Pre-synthesis) Timing Simulation
  - Testbenches
  - Gate Level (Post-synthesis and Pre-Layout) Timing Simulation
  - Post-Layout Timing Simulation
- Types
  - Gate Delay (Inertial Delay)
  - Net Delay (Transport Delay)
  - Module Path Delay

Transport and Inertial Delay

- Transport delay - pure time delay
- Inertial delay
  - Multiple events cannot occur on the output in a time less than the delay.
- Example AND with delay = 2
  \[ A \longrightarrow 1 \text{ ns} \]
  \[ B \]
  \[ C \]
  \[ C \]
  Transport Delay
  Inertial Delay
Gate Propagation Delay (Inertial) - Syntax (FIO)

- delay3 ::= # delay_value | # (delay_value [, delay_value [,delay_value]])
- delay2 ::= # delay_value | # (delay_value [,delay_value])
- delay_value ::= unsigned number | parameter_identifier | constant_mintypmax_expression
- constant_mintypmax_expression ::= constant_expression | constant_expression : constant_expression : constant_expression : constant_expression

Gate Propagation Delay (Inertial) - Examples

- No delay value - default - delays are all 0.
  - nor (z, x1 x2)
- Delay_value - unsigned_number = 1 - unit delay
  - nor #1 (z, x1, x2);
- Delay_value - unsigned_number ≠ 1 - prop delay
  - nor #5 (z, x1, x2);
- Delay_value - parameter_identifier - allows easy change of delay value
  - nor #nor_delay (z, x1,x2);
Gate Propagation Delay (Inertial) - Examples (FIO)

- Delay_value2 - unsigned_number - *rising delay, falling delay*
  - nor #(1,2) (z, x1, x2);
- Delay_value3 - unsigned_number - *rising delay, falling delay, turnoff delay*
  - nor #(3,2,4) (z, x1, x2);
- Delay_value3 - constant_mintypmax_expression - *rising delay* - min:typ:max, *falling delay* - min:typ:max, *turnoff delay* - min:typ:max
  - nor #(2:3:4, 1:2:5, 2:4:6)

Simulation Time Scales

- Compiler Directive `timescale <time_unit> / <time_precision>
- time_unit - the time multiplier for time values
- time_precision - minimum step size during simulation - determines rounding of numerical values
- Allowed unit/precision values:
  \{1| 10 | 100, s | ms | us | ns | ps\}
Simulation Time Scales (continued)

- Example:
  ```
  `timescale 10ps / 1ps
  nor #3.57 (z, x1, x2);
  nor delay used = 3.57 x 10 ps = 35.7 ps => 36 ps
  ```
- Different timescales can be used for different sequences of modules
- The smallest time precision determines the precision of the simulation.
- Will ignore time issues for system tasks/functions

Net Delay (Transport)

- Delay assigned to net such as wire
- Type of delay (inertial or transport) defined by object assigned to.
- Example - Structural:
  ```
  `timescale 10ps /1ps
  wire #4 N25;
  nor #(20,30) GA (N25, x1, x2), GB (z, N25, X3);
  For rising output from x1 to z, 300 + 40 + 200 = 540 ps
  ```
Net Delay (Transport)

- Example - Continuous Assignment
  `timescale 10ps /1ps
  wire #4 N25; transport delay
  assign #(20,30) N25 = ~ (x1 | x2); inertial delay
  For rising output from x1 to N25, 200 + 40 = 240 ps

- Example - Implicit Continuous Assignment
  `timescale 10ps /1ps
  wire #(24,34) N25 = ~ (x1 | x2); inertial delay only
  For rising output from x1 to N25, 240 ps

Module Delay - Example

- Add to module:
  
  specify
  
  (x1, x2 *-> z) = (18:25:33, 24, 31, 40);

  endspecify

- Specifies minimum, typical, and maximum delays on paths from x1 to z and x2 to z.
Behavioral Constructs

- Concurrent communicating behaviors => processes same as behaviors

- Two constructs
  - **initial** - one-time sequential activity flow - not synthesizable but good for testbenches
  - **Always** - cyclic (repetitive) sequential activity flow

- Use procedural statements that assign only register variables (with one exception)

Behavioral Constructs (continued)

- Continuous assignments and primitives assign outputs whenever there are events on the 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.
Behavioral Constructs (continued)

- Body may consist of a single statement or a block statement
- A block statement begins with `begin` and ends with `end`
- Statements within a block statement execute sequentially
- Behaviors are an elaborate form of continuous assignments or primitives but operate on registers (with one exception) rather than nets

Behavioral Constructs - Example

- Initial:     Always:
  
  ```
  initial always
  begin begin
  one = 1; F1 = 0, F2 = 0;
  two = one + 1; # 2 F1 = 1;
  three = two + 1; # 4 F2 = 0;
  four = three + 1; # 2 F1 = 1;
  five = four + 1; # 4;
  end end
  ```

- What are results of each of above?
Procedural Assignments

- Types
  - = blocking assignment
  - assign = continuous assignment
  - <= non-blocking assignment

- Assignments (with one exception) to:
  - reg
  - integer
  - real
  - realtime
  - time

Procedural Assignments - Some Rules

- Register variable can be referenced anywhere in module
- Register variable can be assigned only with procedural statement, task or function
- Register variable cannot be input or inout
- Net variable can be referenced anywhere in module
- Net variable may not be assigned within behavior, task or function. Exception: force ... release
- Net variable within a module must be driven by primitive, continuous assignment, force ... release or module port
Procedural Continuous Assignment (FIO)

- Two types
  - assign ... deassign
    - to register variable
    - dynamic binding to target register
  - force ... release
    - to register or net variable
    - dynamic binding to target register or net variable

Procedural Continuous Assignment - Examples

- Example 1:
  // Q is a reg. What does this describe?
  
  ```vhdl
  always @ (clk)
  begin
    if clk = 1 then assign Q = D;
    else assign Q = Q;
  end
  ```
Procedural Continuous Assignment - More (FIO)

- A Procedural Continuous Assignment **overrides** all regular procedural assignments to variables
- Assignment Modes - See [5] Figure 7-8 p. 172

Procedural Timing, Controls & Synchronization

- Mechanisms
  - Delay Control Operator (#)
  - Event Control Operator (@)*
  - Event or
  - Named Events
  - **wait** construct

*Ignored by FPGA express unless a synchronous trigger that infers a register
Procedural Timing, Controls & Synchronization

- Delay Control Operator (#)
  - Precedes assignment statement - postpones execution of statement
  - For blocking assignment (=), delays all statements that follow it
  - Blocking assignment statement must execute before subsequent statements can execute.
  - Example: always @(posedge clk),
    #10 Q = D;

Procedural Timing, Controls & Synchronization

- Event Control Operator (@)*
  - Synchronizes the activity flow of a behavior to an event (change) in a register or net variable or expression
  - Example 1: @ (start) RegA = Data;
  - Example 2: @(toggle) begin
    ...
    @(posedge clk) Q = D;
    ...
  end
  - “toggle” above will be ignored unless in block
*Ignored by FPGA express unless a synchronous trigger that infers a register
Procedural Timing, Controls & Synchronization

- Event or - allows formation of event expression
- Example:
  ```
  always @ (X1 or X2 or X3)
  assign Y = X1 & X2 | ~ X3;
  ```
- All RHS variables in sensitivity list and no unspecified conditional results => combinational logic

Procedural Timing, Controls & Synchronization

- Meaning of `posedge`: 0 -> 1, 0 -> x, x -> 1
- Special Example:
  ```
  always @ (set or reset or posedge clk)
  begin
    if (reset == 1) Q = 0;
    else if (set == 1) Q = 1;
    else if (clk == 1) Q = data;
  end
  // Does this work correctly? Why or why not?
  ```
Procedural Timing, Controls & Synchronization (FIO)

- Named Events
  ```
  module cpu (...);
  always @ (peripheral.interrupt)
  begin
    ...
  end
  module peripheral (...);
  event interrupt;
  ...
  -> interrupt;
  ```

Procedural Timing, Controls & Synchronization (FIO)

- `wait` Construct
  - Suspends activity in behavior until expression following `wait` is TRUE

- Example:
  ```
  always
  begin
    a = b;
    c = d;
    wait (advance);
  end
  ```
Blocking Assignments

- Identified by =
- Sequence of blocking assignments executes sequentially
- Example:

\[
\text{always } @(\text{posedge } clk) \\
\text{begin} \\
b = 0; c = 0; \\
b = a + a; \\
c = b + a; \\
d = c + a; \\
\text{end}
\]

Non-Blocking Assignments

- Identified by <=
- Sequence of non-blocking assignments executes concurrently
- Example 1:

\[
\text{always } @(\text{posedge } clk) \\
\text{begin} \\
b <= 0; c <= 0; \\
b <= a + a; \\
c <= b + a; \\
d <= c + a; \\
\text{end}
\]

/*Calculates b = 2a, c = b + a, d <= c + a. All values used on RHS are those at posedge clock. Note that there are two assignments to b and c. Only the last one is effective. */
Blocking Assignments - Inter-Assignment Delay

- Delays evaluation of RHS and assignment to LHS
- Example:

```markdown
always @(posedge clk)
begin
    b = 0; c = 0;
    b = a + a;  // uses a at posedge clock
    #5 c = b + a; // uses a at posedge clock + 5
    d = c + a;  // uses a at posedge clock + 5
end /*c = 2 a(at posedge clock)+ a(at posedge clock + 5)
       d = 2 a(at posedge clock) + 2 a(at posedge clock + 5)*/
```

Blocking Assignment - Intra-Assignment Delay

- Delays assignment to LHS, not evaluation of RHS
- Example:

```markdown
always @(posedge clk)
begin
    b = 0; c = 0;
    b = a + a;  // uses a at posedge clock
    c = #5 b + a; // uses a at posedge clock
    d = c + a;  // uses a at posedge clock + 5
end /* c = 3 a(at posedge clock)
      d = 3a (at posedge clock)+ a (at posedge clock + 5)*/
```
Non-Blocking Assignment - Inter-Assignment Delay

- Delays evaluation of RHS and assignment to LHS
- Delays subsequent statements
- Example:

```plaintext
always @(posedge clk)
begin
  b <= 0; c <= 0;
  b <= a + a; // uses a at posedge clock
  #5 c <= b + a; // uses b and a at posedge clock + 5
  d <= c + a; // uses a at posedge clock + 5
end
/*c = b(at posedge clock + 5) + a(at posedge clock + 5)
  d = c(at posedge clock + 5) + a (at posedge clock +5)*/
```

Non-Blocking Assignment - Intra-Assignment Delay

- Delays only assignment to LHS
- Example:

```plaintext
always @(posedge clk)
begin
  b <= 0; c <= 0;
  b <= a + a; // uses a at posedge clock
  c <= #5 b + a; // uses a and b at posedge clock
  d <= c + a; // uses a and c at posedge clock
end
/* Calculates *c(posedge clock + 5) = b(at posedge clock)
   + a(at posedge clock); d(posedge clock) =
   c(at posedge clock) + a (at posedge clock) */
```
Mixed Blocking/Non-Blocking Assignments

Example 1:

```vhdl
always @(posedge clk)
begin
    b = 0; c = 0; d = 0;
    b = a + a;
    c <= b + a + d;
    d = c + a;
end
/*Calculates b = 2a, c = 2a, d = a since 1) RHS of c evaluates
when statement reached, but LHS assigned to c last after all
blocking assignments including that for d and 2) assignment of c
does not delay execution of evaluation of d */
```

Mixed Blocking/Nonblocking Assignments

- Example: `always @(posedge clk)
begin
d <= 0;
b = a + a;
c = b + a;
d <= c + a;
c = d + a;
end
/* Since the d <= c + a is non-blocking, c = d + a proceeds to
execute before the assignment of d <= c + a. The resulting
values calculated are b = 2a, d = 4a, and c = a + d (value
of d is that at posedge clk, not that due to non-blocking
assignment statement. */`
Mixed Blocking/Nonblocking Assignment

- For synthesis in ECE 554:
  - A given register can have either blocking or non-blocking assignments, not both.
  - Delays cannot be used in always statements with mixed assignments
  - It is advisable to avoid the confusion of the prior example to write code with all non-blocking assignments last among the code statements

Activity Control

Overview

- Constructs for Activity Control
  - Conditional operator
  - case statement
  - if ... else statement
  - Loops: repeat, for, while, forever
  - disable statement
  - fork ... join statement
- Tasks and Functions
Conditional Operator

- ? ... :
- Same as for use in continuous assignment statement for net types except applied to register types
- Example:
  ```
  always@(posedge clock)
  Q <= S ? A : B //combined DFF and 2-to-1 MUX
  ```

Case Statement (FIO)

- case Syntax:
  ```
  case_statement ::= case (expression)
  case_item {case_item} endcase
  | casex (expression)
  case_item {case_item} endcase
  | casez (expression)
  case_item {case_item} endcase
  |
  case_item ::= expression {,expression} :
  statement_or_null |default [:] statement_or_null
  statement_or_null ::= statement | 
  ```
case Statement

- Requires complete bitwise match over all four values so expression and case item expression must have same bit length
- Example: `always @(state, x) begin
  reg[1:0] state;
  case (state)
    2'b00: next_state <= s1;
    2'b01: next_state <= s2;
    2'b10: if x next_state <= s0;
      else next_state <= s1;
    end
  default next_state = 1'bxx;
  endcase
end`

casex Statement

- Requires bitwise match over all but positions containing x or z; executes first match encountered if multiple matches.
- Example: `always @(code) begin
  casex (code)
    2'b0x: control <= 8'b00100110; //same for 2'b0z
    2'b10: control <= 8'b11000010;
    2'b11: control <= 8'b00111101;
    default control <= 8'b'xxxxxxxx;
  endcase
end`
casez Statement

- Requires bitwise match over all but positions containing z or ? (? is explicit don’t care); executes first match encountered if multiple matches.

- Example:
  ```vhdl
  reg [1:0] code;
  always @(code) begin
    casez (code)
      2'b0z: control <= 8'b00100110;
      2'b1?: control <= 8'b11000010;
      default control <= 8'b'xxxxxxx;
    endcase
  end
  ```

Conditional (if ... else) Statement Example

```vhdl
always @(a or b or c) begin
  if (a == b) begin
    q <= data;
    stop <= 1'b1;
  end
  else if (a > b) q <= a;
  else q <= b;
end
end
```
Conditional (if ... else) Statement (continued)

- Must be careful to define outcome for all possible conditions – failure do do so can cause unintentional inference of latches!
- else is paired with nearest if when ambiguous - use begin and end in nesting to clarify.
- Nested if ... else will generate a “serial” or priority like circuit in synthesis which may have a very long delay - better to use case statements to get “parallel” circuit.

for Loop Example

- Example:
  initial
    integer r, I;
  begin
    r = 0;
    for (i = 1; i <= 7; i = i + 2)
      begin
        r[i] = 1;
      end
  end
- If the loop above were in time rather than space, should use reg instead of integer!
while Loop Example

- Not synthesizable since forms combinational loop!

```plaintext
initial
begin
  r = 0;
  i = 0;
  while (i <= 7)
  begin
    r[2*i + 1] = 1;
    i = i + 1;
  end
end
```

forever Loop Example

```plaintext
initial
begin
  clk = 0;
  forever
  begin
    #50 clk = 1;
    #50 clk = 0;
  end
end
```

- Usually used in testbenches rather than for synthesized logic.
Tasks (FIO)

- Declared within a module
- Referenced only by a behavior within the module
- Parameters passed to task as inputs and inouts and from task as outputs or inouts
- Local variables can be declared
- Recursion not supported although nesting permitted (nested copies of variables use same storage)
- See Fig. 7.43 p. 226 of [5] for rules

1/28/2001 93

 Tasks (FIO)

- Syntax
  task_declaration ::= task task_identifier
  {task_item_declarations}
  statement or null
  endtask

1/28/2001 94
Task Example

task leading_1;
  input [7:0] data_word;
  output [2:0] position;
  reg [7:0] temp;
  reg [2:0] position;
  begin
    temp = data_word;
    position = 3'b111;
    while (!temp[7]) //*
      @(posedge clock) //*
        begin
          temp = temp << 1;
          position = position - 1;
        end
  end
endtask //* This may not work – unclear contradictory statements in FPGA Express documentation.

Functions (FIO)

- Implement combinational behavior
- No timing controls or tasks which implies no while
- May call other functions with no recursion
- Reference in an expression, e.g. RHS
- No output or inout allowed
- Implicit register having name and range of function
Functions (FIO)

- Syntax:
  
  functionDeclaration ::= function [range or type] function_identifier;

  functionCall ::= function_identifier (expression {, expression})

- Example:
  
  position = leading_1(data_val);

Function Example

```
function [2:0] leading_1;
  input [7:0] data_word;
  reg [7:0] temp;
  begin
    temp = data_word;
    leading_1 = 3'b111;
    while (!temp[7])
      begin
        temp = temp << 1;
        leading_1 = leading_1 - 1;
      end
  end
endfunction
```

- Is the above code synthesizable? No
Finite State Machines - Explicit and Implicit Models

- Explicit - declares a state register that stores the FSM state
- Implicit - describes state implicitly by using multiple event controls
- Mealy versus Moore types

Types of Explicit Models

- State register - Combinational next state and output logic
- State register - Combinational next state logic - Combinational output logic
- State register - Combinational next state logic - Registered output logic
State register - Combinational next state and output logic

Inputs → Next State and Output Logic → Outputs

State Register

 FF

State register - Combinational next state logic - Combinational output logic

Inputs → Next State Logic → Outputs

State Register

 FF

Output Logic

Mealy
State register - Combinational next state and output logic - Output register

State register - Combinational next state logic - Registered output logic
FSM Example: Washer

Verilog - state register - next state and output logic

```
module control_es1 (reset, clk, start, full, empty, timeout, drain, spin, timeset, water);
//state register - combined next state and output logic
input reset, clk, start, full, empty, timeout;
output drain, spin, timeset, water;
reg drain, spin, timeset, water;
reg [2:0] state, next_state;
parameter start_s = 3'b000, fill_s = 3'b001, wash_s = 3'b010, drain_s = 3'b011, wring_s = 3'b100;
always@(posedge clk or posedge reset)
begin
  if (reset) state <= start_s;
  else if (clk) state <= next_state;
end
always@(state or start or full or empty or timeout)
begin
  drain <= 1'b0; spin <= 1'b0;
  timeset <= 1'b0; water <= 1'b0;
  // above sets outputs to default value - in the following,
  // only output changes to 1 are specified
  case (state)
    start_s:
      if (start) next_state <= fill_s;
      else next_state <= start_s;
    fill_s:
      begin
        water <= 1'b1;
        if (full) begin
          next_state <= wash_s;
          timeset <= 1'b1;
        end
        else
          next_state <= fill_s;
      end
    end
  endcase
```

1/28/2001 105
Verilog - state register - next state and output logic (continued)

```verilog
wash_s: begin
    spin <= 1'b1;
    if (timeout)
        next_state <= drain_s;
    else
        next_state <= wash_s;
end

drain_s: begin
    drain <= 1'b1;
    if (empty) begin
        next_state <= wring_s;
        timeset <= 1'b1;
    end
    else
        next_state <= drain_s;
end

wring_s: begin
    spin <= 1'b1;
    drain <= 1'b1;
    if (timeout)
        next_state <= start_s;
    else
        next_state <= wring_s;
end

default
    next_state <= start_s;
endcase
end
endmodule
```

Verilog - state register - next state logic and output logic (FIO)

```verilog
module control_el1 (reset, clk, start, full, empty, timeout, drain, spin, timeset, water);
//state register - next state logic and output logic
input reset, clk, start, full, empty, timeout;
output drain, spin, timeset, water;
reg drain, spin, timeset, water;
reg [2:0] state, next_state;
parameter start_s = 3'b000, fill_s = 3'b001,
    wash_s = 3'b010, drain_s = 3'b011, wring_s = 3'b100;
always@(posedge clk or posedge reset)
    begin
        if (reset) state <= start_s;
        else if (clk) state <= next_state;
    end
```

```verilog
always@ (state or start or full or empty or timeout)
    begin
        start_s: if (start) next_state <= fill_s;
        else next_state <= start_s;
        fill_s: begin
            if (full) begin
                next_state <= wash_s;
                timeset <= 1'b1;
            end
            else
                next_state <= fill_s;
        end
        default next_state <= start_s;
    endcase
endmodule
```

```verilog
wash_s: begin
    if (timeout)
        next_state <= drain_s;
    else
        next_state <= wash_s;
end
```

1/28/2001 107

1/28/2001 108
Verilog - state register - next state logic and output logic (continued) (FIO)

```verilog
drain_s: begin
  if (empty) next_state <= wring_s;
  else next_state <= drain_s;
end
wring_s: begin
  if (timeout) next_state <= start_s;
  else next_state <= wring_s;
end
default next_state <= start_s;
endcase
end
always@(state or full or empty)
  drain <= 1'b0; spin <= 1'b0;
  timeset <= 1'b0; water <= 1'b0;
  // sets outputs to default value - in the following,
  // only output changes to 1 are specified
endcase
endmodule
```

Verilog - State register - Combinational next state and output logic - Output register (FIO)

- Same as state and output register - state and output logic
- Same as combined state and output logic and registers
- Both state and outputs are from flip-flops and synchronized with the clock.
Verilog - State register - Combinational next state and output logic - Output register (FIO)

- If delay of the output for one clock cycle acceptable, then same output logic can feed output flip-flop inputs as originally feed combinational outputs
- Suppose outputs are to obey specifications on a clock cycle specific basis, i.e., are not delayed
- Then the output flip-flop D-input functions must be defined one cycle earlier than the normal combinational output.

Verilog - State register - Combinational next state and output logic - Output register (FIO)

- How is this done?
- Example:

  \[ M(t + 1) = A X + B Y + C Z \] (Moore)
  \[ N(t + 1): \text{Impossible!} \] (Mealy)
Verilog - State register - Combinational next state and output logic - Output register (FIO)

module control_er1 (reset, clk, start, full, empty, timeout, drain, spin, timeset, water);
//state register - combined next state and output logic - output register

input reset, clk, start, full, empty, timeout;
output drain, spin, timeset, water;
reg drain, spin, timeset, water;
reg [2:0] state, next_state;

parameter start_s = 3'b000, fill_s = 3'b001,
wash_s = 3'b010, drain_s = 3'b011, wring_s = 3'b100;
always @(posedge clk or posedge reset)
begin
  drain <= 1'b0; spin <= 1'b0;
timeset <= 1'b0; water <= 1'b0;
  // sets outputs to default value - in the following, // only output changes to 1 are specified
  if (reset) state <= start_s;
else if (clk)
  case (state)
  start_s:
    if (start) begin
      state <= fill_s;
      water <= 1'b1;
    end
    else state <= start_s;
  fill_s:
    if (full) begin
      state <= wash_s;
      spin <= 1'b1;
    end
    else begin
      state <= fill_s;
      water <= 1'b1;
timeset <= 1'b1;
    end
  wash_s:
    if (timeout) begin
      state <= drain_s;
      drain <= 1'b1;
    end
    else spin <= 1'b1;
  drain_s:
    begin
      drain <= 1'b1;
      if (empty) begin
        state <= wring_s;
        spin <= 1'b1;
      end
      else begin
        state <= drain_s;
        drain <= 1'b1;
timeset <= 1'b1;
      end
    end
endcase
end

Verilog - State register - Combinational next state and output logic - Output register (continued)(FIO)

wring_s: if (timeout) begin
  state <= start_s;
else begin
  state <= wring_s;
  spin <= 1'b1;
  drain <= 1'b1;
end
endcase
end
endmodule
Verilog - State register - Combinational next state and output logic - Output register (continued)(FIO)

- How is (Mealy) timeset handled?
  - Timeset is not “used” while in states fill_s and drain_s.
  - Time value is fixed during last cycle before conditions to leave these states, full = 1 and empty = 1, respectively, occur.
  - Can “hammer” timeset every clock cycle until condition to leave these states satisfied.
  - End result in terms of loading the time value is the same as for original design

- Works only for specific conditions!

Implicit Model

- More abstract representation
- Restricted to structures in which a given state can be entered from only one other state!
- Yields simpler code
- Description of reset behavior more complex
- Ciletti examples not good illustrations [5]
- For novice, good route to disaster!
Compiler Directives

- Useful for controlling what is synthesized and the resulting logic
- Warning: Not recognized by other compilers – therefore reduce code portability
- Examples:
  - // synopsys translate_off
    Code here describes something that is not to be synthesized such as a simulation testbench - can contain non-synthesizable constructs such as delays
  - // synopsys translate_on

Compiler Directives (Continued)

- Examples:
  - // synopsys parallel_case
    Forces generation of multiplexer-like structure instead of priority structure when included after case declaration
  - // synopsys full_case
    Indicates that all cases have been considered when included in case declaration; when used, no default statement needed and latches will not be inferred
  - case (state) // synopsys parallel_case full_case
Compiler Directives (Continued)

- Other Directives
  - For FSMs:
    - // synopsys state_vector
    - // synopsys enum
  - For instantiating modules in behavioral (always) code
    - // synopsys map_to_module modulename
    - // synopsys return_port_name portname
- See Chapter 8 of [2]

Simulation and Testbenches

- Generic Simulation Structure
Testbench Approach

- Use Verilog module to produce testing environment including stimulus generation and/or response monitoring

Stimulus Generation Example

```verilog
`timescale 1ns /1ns
module com_test_bench_v;
    reg[8:0] stim;
    wire[3:0] S;
    wire C4;
    adder_4_b_v a1(stim[8:5], stim[4:1], stim[0], S, C4);

    //Continued on next slide
endmodule
```
Stimulus Generation Example (Continued)

```vhdl
//Generate stimulus
initial
begin
    stim = 9'b000000000;
    #10 stim = 9'b111100001;
    #10 stim = 9'b000011111;
    #10 stim = 9'b111100010;
    #10 stim = 9'b000111110;
    #10 $stop;
end
```

Other Testbench Stimuli Generators

- Counters (Good for up to 8 or 9 input Variables)
- Linear Feedback Shift Registers
- Loadable Shift Register with Initialization Memory
- Memory Containing Test Vectors
- FSM
Testbench Response Analyzers

- Comparison to Memory Containing Response Vectors
- Linear Feedback Shift Register
- Comparison to Behavioral Verilog Model Response
- FSM

References