計算機輔助設計

Computer-Aided Design

Chu Yu
Contents

1. VHDL/Verilog-HDL
2. HSPICE
3. FPGA (Altera/Xilinx development tools)
4. Logic Synthesis
5. Floorplan
6. Placement
7. Routing
8. Testing
Grading

- Midterm Exam 30%
- Term Project #1 (FPGA) 20%
- Term Project #2 (Full Custom) 30%
- Homework 20%
CAD

FPGA Design Flow

Full Custom Design Flow

Cell Based Design Flow

MMIC Design Flow
FPGA 設計流程

Workstation

HDL設計模擬

Gate Level設計模擬

佈局階層設計

佈局後驗證模擬

雛型製作

VCS, speedwave

FPGA Compiler II /Synplify

Alter, Xilinx

Gatesim, VCS, speedwave

VCS, speedwave

Verilog-XL, Polar, VSS

Verilog-XL, Polar, VSS

FPGA Express, Altera

FPGA/CPLD Download

FPGA/CPLD Download
Full-Custom 設計流程

電路階層設計模擬
Composer/S-Edit
Star-Hpice, SBTSPICE, Spectre, TSPICE
Virtuoso/Diva, Dracula/L-edit
RC Extraction (Star-RC, Dracula)
GDSII
Tapeout
Cell Based 設計流程

系统層級設計模擬

RT Level 設計模擬

Gate Level 設計模擬

電路層級模擬

佈局層級設計驗證

佈局後驗證模擬

光罩製作

CAD

SPW

Verilog-XL,Polaris,VSS

Design-Compiler,View Synthesis

Verilog-XL

Timemill,star-time star-sim

Silicon Ensemble/Dracula, Apollo//Hercules

GDSII

Tapeout

RC Extraction(Star-RC, Dracula)

CELL LIBRARY Compass(1.6um/0.35um)

Smart model

BCNeS

Visual Architect

HDL Debugger
MMIC 設計流程

系統階層設計模擬 → Symphony

電路階層設計模擬 → Harmonica

佈局階層設計驗證 → Virtuoso/Diva

佈局後驗證模擬 → GDSII

光罩製作 → Tapeout
Verilog HDL

- HDL – Hardware Description Language
  - A programming language that can describe the functionality and timing of the hardware.

- Why use an HDL?
  - It is becoming very difficult to design directly on hardware.
  - It is easier and cheaper to different design options.
  - Reduce time and cost.
Verilog HDL

- Verilog-XL is an **event-driven** simulator that can emulate the hardware described by Verilog HDL.
- Verilog-HDL allows you to describe the design at various levels of abstractions.
  - Behavior level
  - RTL level
  - Gate level
  - Switch level
Time Wheel in Event-Driven Simulation

Event queues at each time stamp

An event $E_t$ at time $t$
Schedules another event at time $t+2$

Time advances only when every event scheduled at that time is executed.
Different Levels of Abstraction

- **Architecture / Algorithmic**
  - A model that implements a design algorithm in high-level language construct.
  - A behavioral representation describes how a particular design should respond to a given set of inputs.

- **Register Transfer Logic (RTL)**
  - A model that describes the flow of data between registers and how a design process these data.

- **Gate Level**
  - A model that describes the logic gates and the interconnections between them.

- **Switch Level**
  - A model that describes the transistors and the interconnections between them.
Verilog-HDL Behavior Language

- Structural and procedural like the C language.
- Used to describe algorithmic-level and RTL-level Verilog models.
- Key features:
  - Procedural constructs for conditional, if-else, case, and looping operations.
  - Arithmetic, logical, bit-wise, and reduction operations for expressions.
  - Timing control.
Verilog-HDL Structural Language

- Used to describe gate-level and RTL-level Verilog models.
- Key features:
  - A complete set of combinational primitives.
  - Support the specification of primitive gate delay.
  - Support the specification of the output strength on a primitive gate.
CAD

16

Behavioral Domain
- Applications
- Operating Systems
- Programs
- Subroutines

Structural Domain
- PC
- RISC Processor
- Adders, Gates, Registers

Physical Domain
- Modules, Chips, Boards, Boxes
- Cells
- Transistors

Y-Chart
- Logic Abstraction Level
- Circuit Abstraction Level
- Architectural Abstraction Level
Top Down ASIC Design Flow

1. Idea and Specification
2. Behavioral Modeling
   - Behavior Model
     (Verilog HDL or C language)
   - Verification (Verilog-XL)
3. Partitioning and Re-modeling
4. Logic Blocks with Function Definition
5. RTL Modeling
   - RTL Model
     (Verilog HDL)
   - Verification (Verilog-XL)

Code Snippet:

```
... sum = a + b;
...

... always @( a or b or c)
  {carry, sum} = a+b+c;
...```

Edited by Chu Yu
Top Down ASIC Design Flow (Con’t)

RTL Model  
(*Verilog HDL*)

Logic Synthesis  
(*Synopsys*)

Gate Level Netlist  
(*Verilog HDL*)

Physical Design  
(*CELL3 Ensemble*)

Verification  
(*Verilog-XL*)

ASIC Libraries  
(*Compass cell library*)

GDS II

... 
{xo03d1 u0(sum,a,b,c); an02d1 u1(g2,a,b); ...}
Overview of Verilog Module

- A Verilog module includes the following parts:

  ```verilog
  module module_name (port_name);
  port declaration
  data type declaration
  Task & function declaration
  module functionality or declaration
  timing specification
  endmodule
  ```
Verilog-HDL Structural Language

- **Verilog Module**
  - Modules are basic building blocks in hierarchy.
  - Every module description starts with module name(output_ports,input_ports), and ends with endmodule.

- **Module Ports**
  - Module ports are equivalent to the pins in hardware.
  - Declare ports to be *input*, *output*, or *inout* (bidirectional) in the module description.
Example

- A Full Adder

```verbatim
module adder (carry, sum, a, b, cin);
  output carry, sum;
  input a, b, cin;
  wire w0, w1, w2;
  always @(a or b or cin);
    assign {carry, sum} = a + b + cin;
endmodule

module adder (carry, sum, a, b, cin);
  output carry, sum;
  input a, b, cin;
  wire w0, w1, w2;
  xor u0(sum, a, b, cin);
  and u1(w0, a, b);
  and u2(w1, a, b);
  and u3(w2, a, b);
  or  u4(carry, w0, w1, w2)
endmodule
```
Three Levels of Verilog-HDL

- **Behavioral level**
  
  assign {Co, Sum} = A + B + Ci

- **Gate level**
  
  xor  u0(.z(hs), .a1(A), .a2(B)),  
  u1(.z(Sum), .a1(Ci), .a2(hs));  
  and u2(.z(hc0), .a1(A), .a2(B)),  
  u3(.z(hc1), .a1(Ci), .a2(hs));  
  or u4(.z(Co), .a1(hc0), .a2(hc1));

- **Switch level**

  // AND gate of u2
  
  pmos p0(VDD, nand, A),  
  p1(VDD, nand, B);  
  nmos n0(nand, wire1, A),  
  n1(wire1, GND, B);  
  pmos p2(VDD, hc0, nand);  
  nmos n2(hc0, GND, nand);  
  ...

---

**CAD**

Edited by Chu Yu
Testing and Verification of Full Adder

- **Test the full adder’s Verilog model by applying test patterns and observing its output responses.**
  - Stimulus and control: Changes on device inputs, simulation finish time, ... etc.
  - Device under test: Behavior, gate, or switch level modules.
  - Response generation and verification: Which signals to save/display, verification of generated response.
Circuit Description

module add4(sum, carry, A, B, Cin);
  output [3:0] sum;
  ....
endmodule

Testfixture

module testfixture;
  reg [3:0] A, B;
  ....
endmodule

Verilog Simulation

Verilog Parser

Simulation Engine

User Interface

0.00 ns  in = 0  out = x
16.00 ns  in = 0  out = 1
100.00 ns in = 1  out = 1
....
Example with a Test fixture

- A Full Adder

```verilog
module testfixture;
  reg   a, b, cin;
  wire  sum, carry;
  adder u0 (carry, sum, a, b, cin);
initial begin
  $monitor($time, "a=%b b=%b cin=%b sum=%b carry=%b", a, b, cin, sum, carry);
  a=0; b=0; ci=0;
  #10 a=0; b=0; ci=1;
  #10 a=0; b=1; ci=0;
  #10 a=0; b=1; ci=1;
  #10 a=1; b=0; ci=0;
  #10 a=1; b=0; ci=1;
  #10 a=1; b=1; ci=0;
  #10 a=1; b=1; ci=1;
  #10 $finish;
end
endmodule
```

```verilog
module adder (carry, sum, a, b, cin);
  output carry, sum;
  input  a, b, cin;
  wire   w0, w1, w2;
  xor    u0(sum, a, b, cin);
  and    u1(w0, a, b);
  and    u2(w1, a, b);
  and    u3(w2, a, b);
  or      u4(carry, w0, w1, w2);
endmodule
```

This will generate some text outputs as

```
0 a=0 b=0 c=0 sum=0 carry=0
10 a=0 b=0 c=1 sum=1 carry=0
... ...
```
Monitor Function

- Any expression parameter that has no corresponding format specification is displayed using the default decimal format.

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%h</code> or <code>%H</code></td>
<td>display in hexadecimal format</td>
</tr>
<tr>
<td><code>%d</code> or <code>%D</code></td>
<td>display in decimal format</td>
</tr>
<tr>
<td><code>%o</code> or <code>%O</code></td>
<td>display in octal format</td>
</tr>
<tr>
<td><code>%b</code> or <code>%B</code></td>
<td>display in binary format</td>
</tr>
<tr>
<td><code>%c</code> or <code>%C</code></td>
<td>display in ASCII character format</td>
</tr>
<tr>
<td><code>%v</code> or <code>%V</code></td>
<td>display net signal strength</td>
</tr>
<tr>
<td><code>%n</code> or <code>%N</code></td>
<td>display net normalized voltage in Switch-RC</td>
</tr>
<tr>
<td><code>%m</code> or <code>%M</code></td>
<td>display hierarchical name</td>
</tr>
<tr>
<td><code>%s</code> or <code>%S</code></td>
<td>display as a string</td>
</tr>
<tr>
<td><code>%t</code> or <code>%T</code></td>
<td>display in current time format</td>
</tr>
</tbody>
</table>
System Functions

**Always and Initial**
- always
- initial
  - $\textit{stop}$: Stopping the simulation.
  - $\textit{finish}$: Finishing the simulation.

**Monitoring Commands**
- $\textit{monitor}($\$\textit{time},"a=%d, b=%b,...\n",a,b);$
- $\textit{gr\_waves}("<signal\_label>",<signal>,...);$  
- simWave: $\textit{shm\_open}("<file\_name>"),  $\textit{shm\_probe}())$
Procedural Timing Controls

- **Three types of timing controls**
  - `<delay>` : Simple delay.
  - `@(<signal>)` : Edge-triggered control.
  - `wait(<expr>)` : Level-sensitive control.

- **Examples**

```verilog
always @(posedge clk)
begin
  #5 q=d;
  #1 qb=~d;
end

initial
begin
  read=0;
  wait(en1|en2) read=1;
  #5 read=0;
end

always wait(set)
begin
  @(posedge clk);
  #3 q=1;
  #1 q=0;
  wait(!set);
end
```
Block Statement

- Block statement are used to group two or more statements together.
- Two types of blocks
  - Sequential Block
    - Enclosed by keyword `begin` and `end`.
  - Parallel Block
    - Enclosed by keyword `fork` and `join`.

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>join</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</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>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</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>join</td>
<td></td>
</tr>
</tbody>
</table>
SimWave

• Using system tasks to save the circuit state into waveform database.
• You can use SimWave to view the waveforms after Verilog-XL simulation.

• Example

module testfixture;

    .......
    initial begin
        $shm_open("adder,shm");
        $shm_probe("A");
        
    .......
    #10 $finish;
endmodule
Hierarchical Modules

- **4-bit carry-propagate adder**

```verilog
module CPA(Cout, Sum, a, b, Cin);
output Cout;
output [3:0] Sum;
input [3:0] a, b;
input Cin;
wire [2:0] c;
FA fa0(c[0], Sum[0], a[0], b[0], Cin),
fa1(c[1], Sum[1], a[1], b[1], c[0]),
fa2(c[2], Sum[2], a[2], b[2], c[1]),
fa3(Cout, Sum[3], a[3], b[3], c[2]);
endmodule
```

CPA.v
Syntax of Verilog

- **C-like structural language**
  - statement: `begin ... end`, `if ... then ... else`, and so on.
  - free writing format: statement ended with `;`.
  - remark: between `/* */` or `//` until the line feed.
  - hierarchical modules.

- **Two types of variables**
  - Nets
  - Registers
Nets and Registers

- **Nets**: nets are continuously driven by the devices that drive them.
  - wire, wor, wand, ...
    - example: wire [7:0] w1, w2;
    - if wire is not vector type, then it doesn’t need to declaration.

- **Registers**: registers are used extensively in behavioral modeling and in applying stimulus.
  - reg
    - example: reg [3:0] variable;
### Types of Nets

- Various net types are available for modeling design-specific and technology-specific functionality.

<table>
<thead>
<tr>
<th>Net Types</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire, tri</td>
<td>For multiple drivers that are Wired-OR</td>
</tr>
<tr>
<td>wand, triand</td>
<td>For multiple drivers that are Wired-AND</td>
</tr>
<tr>
<td>trireg</td>
<td>For nets with capacitive storage</td>
</tr>
<tr>
<td>tri1</td>
<td>For nets with weak pull up device</td>
</tr>
<tr>
<td>tri0</td>
<td>For nets with weak pull down device</td>
</tr>
<tr>
<td>supply1</td>
<td>Power net</td>
</tr>
<tr>
<td>supply0</td>
<td>Ground net</td>
</tr>
</tbody>
</table>
Types of Nets

- **Example I**

```
wire z;
```

- **tri0 z:**
  - if \( en = 1 \), \( z = i \);
  - if \( en = 0 \), \( z = 0 \);

```
trireg z;
```

- **if \( en = 1 \), \( z = i \);**
- **if \( en = 0 \), \( z \) stores its last value;**

```
tril z;
```

- **if \( en = 1 \), \( z = i \);**
- **if \( en = 0 \), \( z = 1 \);**
Types of Nets

- **Example II**
  - **TTL**

Any floating input in a TTL logic gate acts like a logical 1 applied at that input.

An open collector TTL output operates wired AND function when two or more gates are wired together.

\[ tri 1 \text{ i ; } \]
\[ i \]
\[ zn \]

\[ \text{wand zn ; } \]
\[ i_1 \]
\[ i_2 \]
\[ zn \]
Operators Used in Verilog

- **Concatenation**: \{x,y\}
  - example: adder4 a1(sum,carry,{a[2],a[2:0]},b[3:0]);

- **Arithmetic operation**: +,-,*
  - example: a=b+c;
    \[ x = y \times z; \]

- **Condition**: = =, !=, >, <, >=, <=, ...
  - example: if (a = = 0) b = c;
Operators Used in Verilog (Cont.)

- **Verilog Language Operators**

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Operators</td>
<td>+, -, *, /, %</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>==, !=, ===, !==</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>!, &amp;&amp;,</td>
</tr>
<tr>
<td>Bit-Wise Operators</td>
<td>~, &amp;,</td>
</tr>
<tr>
<td>Unary Reduction</td>
<td>&amp;, ~&amp;,</td>
</tr>
<tr>
<td>Shift Operators</td>
<td>&gt;&gt;, &lt;&lt;</td>
</tr>
<tr>
<td>Conditional Operators</td>
<td>:</td>
</tr>
<tr>
<td>Concatenations</td>
<td>{}</td>
</tr>
</tbody>
</table>
Operators Used in Verilog (Cont.)

- **Bitwise operators:**
  - and: &
  - or: |
  - not: ~
  - xor: ^

- **Index:**
  - example: a[11:6], b[2], ...
Literal Numbers

- Literal integers are interpreted as decimal numbers in the machine word size (32 bits) by default.
- Size and base may be explicitly specified

\(<\text{size}>'\text{<base>}\text{<value}>\)

- \(<\text{size}>\): size in bits as a decimal number.
- \(<\text{base}>\): b(binary), o(octal), h(hex), d(decimal).
- \(<\text{value}>\): 0-9, a-f, x, z, ? (must be legal number in \(<\text{base}>\))

- Four types of logic value
  - 0 (logical 0), 1 (logical 1), x (unknown), z (high impedance)
# Summary of Shorthand Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic 0</td>
<td>input field</td>
</tr>
<tr>
<td>1</td>
<td>Logic 1</td>
<td>input field</td>
</tr>
<tr>
<td>x</td>
<td>Unknown</td>
<td>output field</td>
</tr>
<tr>
<td>?</td>
<td>Iteration of 0, 1, and x</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Iteration of 0 and 1</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>(vw)</td>
<td>Change of value from v to w</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Same as (??)</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>Same as (01)</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>Same as (10)</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>Iteration of (01), (0x), and (x1)</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>Iteration of (10), (1x), and (x0)</td>
<td></td>
</tr>
</tbody>
</table>

- **Comments:**
  - Any value change on input
  - Rising edge on input
  - Falling edge on input
  - Positive edge including x
  - Negative edge including x
Literal Numbers (cont.)

• **Examples**

  12  32-bit decimal  
  8'd45  8-bit decimal  
  10'hF1  10-bit hex (left-extended with zero)  
  1'B1  1-bit binary  
  32'bz  32-bit Z  
  6'b001_010  6-bit binary with underscore for readability.

Underscores are ignored. 
X and Z values are automatically extended. 
A question mark ? in <value> is interpreted as a Z.
High-level Programming Language Constructs

● **Looping Controls**

  ♦ **Forever loop**

  
  **example**

  forever #100 clk=~clk;

  ♦ **Repeat loop**

  
  **example**

  repeat(mem_depth)
  begin
  mem[address]=0;
  address=address+1;
  end

  ♦ **While loop**

  
  **example**

  while(val[index]==1'b0)
  index=index-1;

  ♦ **For loop**

  
  **example**

  for(index=0;index<size;
  index=index+1)
  if(val[index]==1'bx)
  $display("found an x");
High-Level Programming Language Constructs (cont.)

- **Decision-making controls**

  **If statement**

  *example*

  ```
  if(set == 1) out = 1;
  if(clear == 0) q = 0;
  else q = d;
  ```

  **Case statement**

  *example*

  ```
  case(instruction)
  2'b00: out = a + b;
  2'b01: out = a - b;
  endcase
  ```
# Logic Level Modeling

- **Built-in primitive functions**

<table>
<thead>
<tr>
<th>Gates</th>
<th>MOS Switches and Bidirectional Transistors</th>
<th>Nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
<td>wire</td>
</tr>
<tr>
<td>nand</td>
<td>bufif0</td>
<td>supply0</td>
</tr>
<tr>
<td>nor</td>
<td>bufif1</td>
<td>wand</td>
</tr>
<tr>
<td>or</td>
<td>notif0</td>
<td>wor</td>
</tr>
<tr>
<td>xor</td>
<td>notif1</td>
<td>tri</td>
</tr>
<tr>
<td>xnor</td>
<td>pullup</td>
<td>triand</td>
</tr>
<tr>
<td>not</td>
<td>pulldown</td>
<td>trior</td>
</tr>
<tr>
<td></td>
<td>nmos</td>
<td>trireg</td>
</tr>
<tr>
<td></td>
<td>tran</td>
<td>tri0</td>
</tr>
<tr>
<td></td>
<td>pmos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tranif0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cmos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tranif1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rnmos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rtran</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rpmos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rtranif0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rcmos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rtranif1</td>
<td></td>
</tr>
</tbody>
</table>

- `bufif1 (z, i, en)`;
- `bufif0 (z, i, en)`;
- `notif1 (zn, i, en)`;
- `notif0 (zn, i, en)`;

![Logic Level Modeling Diagram]
Switch Level Modeling

nMOS (unidirectional)

pmOS (unidirectional)

cMOS (unidirectional)

nmos(out, in, ctl);

pmos(out, in, ctl);

cmos(out, in, nctl, pctl);
Continuous Assignment

Continuous assignment provide a means to abstractly model combinational hardware driving values onto nets. An alternate version of the 1-bit full adder is shown below:

```verilog
module FA(Cout, Sum, a,b,Cin);
output Cout, Sum;
input a,b,Cin;

assign Sum = a ^ b ^ Cin,
            Cout = (a & b) | (b & Cin) | (a & Cin);
endmodule
```
Procedural Assignments

- Assignments made within procedural blocks are known as procedural assignments.
- The left-hand side of procedural assignment must be a data type in the register class.

**Example**

```
initial
begin
    out=0;
    #10 en1=~net23;
    #5 set=(r1|en1)&net4;
end
```
Intra-Assignment Timing Control

- Previously described timing control.
  
  ```
  #100 clk = ~ clk;
  @(posedge clock) q = d;
  ```

- Intra-assignment timing control.
  
  ```
  clk = #100 ~ clk;
  q = @(posedge clock) d;
  ```

- Simulators perform two steps when encounter an intra assignment timing control statement.
  - Evaluate the RHS immediately.
  - Execute the assignment after a proper delay.
Intra-Assignment Timing Control

- Intra-assignment timing control can be accomplished by using the following constructs

<table>
<thead>
<tr>
<th>With intra-assignment construct</th>
<th>With intra-assignment construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = #10 b;</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>temp = b;</td>
</tr>
<tr>
<td></td>
<td>#10 a = temp;</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td>a = @(posedge clk) b;</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>temp = b;</td>
</tr>
<tr>
<td></td>
<td>@ (posedge clk) a = temp;</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td>a = repeat(3)@(posedge clk) b;</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>temp = b;</td>
</tr>
<tr>
<td></td>
<td>@ (posedge clk)</td>
</tr>
<tr>
<td></td>
<td>@ (posedge clk)</td>
</tr>
<tr>
<td></td>
<td>@ (posedge clk) a = temp;</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
Non-Blocking Procedural Assignment

- **Blocking procedural assignment.**
  
  rega = #100 regb;
  rega = @(posedge clk) regb;

- **Non-Blocking procedural assignment.**
  
  rega <= #100 regb;
  rega <= @(posedge clk) regb;

- **Schedule the assignment without blocking the procedural flow.**

- **Simulators perform two steps when encounter an non-blocking procedural assignment statement.**
  - Evaluate the RHS immediately.
  - Schedule the assignment at a proper time.
Blocking Procedural Assignment

Initial begin
   a = #10 1;
   $display("current time = %t a = %b", $time, a);
   evaluate at time = 10, a = 1
end
Non-Blocking Procedural Assignment

Initial begin
    a <= #10 1;
    $display(“current time = %t a = %b”, $time, a);
    ── evaluate at time = 0, a = x
end

event queues at each time stamp

$display(…)
Evaluate RHS (RHS = 1)
assign logic 1 to a

Current Simulation Time

t = 0

t = 1

t = 10

t = 11
Delay and Time Scales

- **Delays**
  - **Gate description**
    
    \[
    \text{buf } \#<\text{delay}> \text{ buf0}(X,A);
    \]
    
    where \(<\text{delay}>\) is:
    
    \(<\text{delay time}>\) or
    
    \(<\text{minimum delay}>:<\text{typical delay}>:<\text{maximum delay}>\)
    
    **example:** buf #(3:4:5) buf0(X,A);

  - **Modeling separate rise and fall delays**
    
    \[
    \text{not } \#<\text{delay}> \text{ not0}(X,A);
    \]
    
    where \(<\text{delay}>\) is
    
    \(<\text{rise delay}>,<\text{fall delay}>\)
    
    **example:** not #(2.23:2.33:2.66,3.33:3.47:3.9) not0(X,A);
Delay and Time Scales (cont.)

- Three-state drivers: include rise, fall, and turn off delays
  
  *example*: not #(2.2:2.3:2.6, 3.3:3.4:3.9, 0.2:0.2:0.2) u0(out, in);

*Timescales*

The `timescale compiler directive can be used to specify delays in explicit time units.

Syntax of the `timescale compiler directive:

```
timescale <unit>/<precision>
```

*example*: `timescale 1ns/10ps`

then the design will be simulated in units of 10 ps.

*example*: not #(2.337,3.472) not1(X, A); 2.337ns will be scaled to 234 ten pico-second units for simulation purposes.
User Defined Primitives

- UDPs permit the user to augment the set of pre-defined primitive elements.
- Use of UDPs may reduce the amount of memory required for simulation.
- Both level-sensitive and edge-sensitive behavior is supported.
UDP Definition

- Pure combinational Logic

```vhdl
primitive mux(o,a,b,s);
output o;
input a,b,s;

table
// a b s : o
0 ? 1 : 0;
1 ? 1 : 1;
? 0 0 : 0;
? 1 0 : 1;
0 0 x : 0;
1 1 x : 1;
endtable
endprimitive
```

- The output port must be the first port.
- UDP definitions occur outside of a module.
- All UDP ports must be declared as scalar inputs or outputs. UDP ports cannot be inout.
- Table columns are inputs in order declared in primitive statement-colon, output, followed by a semicolon.
- Any combination of inputs which is not specified in the table will produce an 'x' at the output.
UDP Definition (cont.)

- **Level-sensitive Sequential Behavior**

```verilog
primitive latch(q,clock,data);
output q;
reg q;
input clock,data;

table
// clock data : state_output : next_state
0   1   :   ?   :   1;
0   0   :   ?   :   0;
1   ?   :   ?   :   -;
endtable
endprimitive
```

- The '?' is used to represent don't care condition in either inputs or current state.
- The '-' in the output field indicates 'no change'.
UDP Definition (cont.)

**Edge-sensitive Sequential Behavior**

primitive d_edge_ff(q,clock,data);
output q;
reg q;
input clock,data;

```vhdl

endprimitive
```

// ignore negative edge of clock
(0?) : ? : ? : -;

// ignore data changes on steady clock
(? ?) : ? : -;

Note that the table now has edgeterms representing transitions on inputs.
Specify Blocks

- What is a specify block?
  - Specify block let us add timing specifications to paths across a module.

```
    a
   / 
  b   Full Adder
   
    cin
     
    sum
      
    carry
```

```
Tlh_a_to_Sum = 1.2     Tlh_b_to_Sum = 1.5
Thl_a_to_Sum = 2.0     Thl_b_to_Sum = 2.2
...
...```
Example of Specify Blocks

```verilog
module DFF (q, d, clk);
    input   d, clk;
    output  q;
    reg     notifier;
    UDP_DFF u0(q, d, clk notifier);

    specify
        specparam
            InCap$d = 0.024, Tsetup$d_cp = 0.41, Thold$d_cp = 0.2;
        ...
        (cp => q) = (0.390, 0.390);
        ...
        $setup(d, posedge cp, Tsetup$d_cp, notifier);
        $hold(posedge cp, d, Thold$d_cp, notifier);
    }
    perform timing check
    endspecify
endmodule
```

![Diagram of DFF with inputs d and clk, output q]
Trouble Shooting

- If \( a=b \) is triggered by some event, \( a \) must be declared as \texttt{reg}.
- A bus signal must be declared as \texttt{wire}.
- The negative value \texttt{should be sign-extended}.
- The port size and number of a module should match anywhere it is referred.
FIGURE 6.13  Actel logic cell
QuickLogic logic cell
Figure 1: Simplified Block Diagram of XC4000-Series CLB (RAM and Carry Logic functions not shown)
Figure 16: Simplified Block Diagram of XC4000E IOB
Figure 17: Simplified Block Diagram of XC4000EX IOB (shaded areas indicate differences from XC4000E)