Digital System Design

Logic Design at Register-Transfer Level

Lecturer: Chihhao Chao (趙之昊)
Date: 2009.3.4
Outline

- Levels of Abstraction
- Register-Transfer
- Syntax for RT-level design in Verilog
- Operators
- Assignments
Various Abstraction of Verilog HDL

❖ Switch level Verilog description of a inverter

```verilog
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
```

```plaintext
in out
pwr gnd
```
Various Abstraction of Verilog HDL

- Gate level Verilog description of a full-adder

```
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

- **RT-Level (RTL) 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:

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

- Behavioral level Verilog description of a full adder

```verilog
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
dendmodule
```

![Diagram of full adder circuit](image)
What is Register Transfer Level?

- In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a synchronous digital circuit.

- In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers, and the logical operations performed on those signals.
Description at RT-Level

- Register (memory, usually D-Flip/Flops)
- Transfer (operation, combinational)
Procedures to Design at RT-Level

1. Partition design into register part (synchronous DFFs) and transfer part (combinational circuit)
2. Declare net variables for input signals and output signals of transfer part
3. Describe transfer by logical/arithmetic/conditional operators on input signals and use continuous assignment operator on output signals
Example

wire [7:0] in1;
wire [7:0] in2;
wire [7:0] in3;

wire [7:0] out1;
wire [7:0] out2;
wire [7:0] out3;

assign out1 = in1 & in2;
assign out2 = in1 + in3;
assign out3 = in1[0]==1’b1 ? in2 : in3;

input signals of transfer part
output signals of transfer part

transfer description
## Operators

- 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{ }}

*limited support for synthesis*

**define and lecture later**
Operators

Arithmetic Operators: unsigned representation

<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
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>

\(\sim(101011)=010100\)
\((010101)\&(001100)=000100\)
\((010101)\|(001100)=011101\)
\((010101)^{(001100)}=011001\)
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>

\( & (10101010) = 1'b0 \)
\( | (10101010) = 1'b1 \)
\( & (10\times0\times0\times) = 1'b0 \)
\( | (10\times01010) = 1'b1 \)
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”
Operator

Logical operators

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

<table>
<thead>
<tr>
<th><code>reg_in</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reg_in</code></td>
<td><code>6'b011100</code></td>
</tr>
<tr>
<td><code>reg_in&lt;&lt;3</code></td>
<td><code>100000</code></td>
</tr>
<tr>
<td><code>reg_in&gt;&gt;3</code></td>
<td><code>000011</code></td>
</tr>
</tbody>
</table>
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'd00)?i0:
                   (sel==2'd01)?i1:
                   (sel==2'd10)?i2:
                   (sel==2'd11)?i3:
                   4'dbx;
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 +, -, *, /, %, &, |, ^, ^~:
  - Arithmetic binary and bitwise
  - Bit width = max (width(x), width(y))
Expression Bit Widths (continued)

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

- $x ? y : z$
  - Conditional
  - Bit width = max(width(y), width(z))

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

- ${x{y, \ldots, z}}$
  - Replication
  - Bit width = $x \times (\text{width}(y) + \ldots + \text{width}(z))$
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:
    
    \[
    f(1,1) = 1, \quad f(0,0) = 0, \quad \text{otherwise } x.
    \]
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>wire, tri</td>
</tr>
<tr>
<td>Procedural Assignment</td>
<td>Register</td>
</tr>
<tr>
<td></td>
<td>reg, integer, real</td>
</tr>
</tbody>
</table>

Left Hand Side = LHS
Right Hand Side = RHS
Assignments

- **Continuous assignment**

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

- **Procedural assignment**

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

```verbatim
module 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

```plaintext
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=",pwr_on, " STABLE=",pwr_stable," OK=",Ok," FIRE=",Fire," GOOD=",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

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

```
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

- 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.

- 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
  ```
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

```vhdl
wire [3:0] a;
wire [3:0] b;
assign a = b + a;
```