Unit-1
Introduction to HDLs
&
Basic Concepts

By
Smitashree Mohapatra

History of Verilog® HDL

- Beginning: 1983
  - "Gateway Design Automation" company
  - Simulation environment
    - Comprising various levels of abstraction
      - Switch (transistors), gate, register-transfer, and higher levels

History of Verilog® HDL (cont'd)

- Three factors to success of Verilog
  - Programming Language Interface (PLI)
    - Extend and customize simulation environment
  - Close attention to the needs of ASIC foundries
    - "Gateway Design Automation" partnership with Motorola, National, and UTMC in 1987-89
  - Verilog-based synthesis technology
    - "Gateway Design Automation" licensed Verilog to Synopsys
    - Synopsys introduced synthesis from Verilog in 1987

Overview of Digital Design Using Verilog

- Evolution of Computer-Aided Digital Design
- Emergence of HDLs
- Typical Design Flow
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

Evolution of Computer-Aided Digital Design

- SSI: Small scale integration
  - A few gates on a chip
- MSI: Medium scale integration
  - Hundreds of gates on a chip
- LSI: Large scale integration
  - Thousands of gates on a chip
  - CAD: Computer-Aided Design
    - CAD vs. CAE
  - Logic and circuit simulators
  - Prototyping on bread board
  - Layout by hand (on paper or a computer terminal)

Evolution of Computer-Aided Digital Design (cont'd)

- VLSI: Very Large Scale Integration
  - Hundred thousands of gates
  - Not feasible anymore:
    - Bread boarding
    - Manual layout design
  - Simulator programs
  - Automatic place-and-route
  - Bottom-Up design
    - Design small building blocks
    - Combine them to develop bigger ones
  - More and more emphasis on logic simulation

Emergence of HDLs

- The need to a standardized language for hardware description
  - Verilog® and VHDL
- Simulators emerged
  - Usage: functional verification
  - Path to implementation: manual translation into gates
- Logic synthesis technology
  - Late 1980s
  - Dramatic change in digital design
    - Design at Register-Transfer Level (RTL) using an HDL

Typical VLSI Design Flow

1. Design specification
2. Behavioral description
3. RTL description
4. Functional verification and testing
5. Logic synthesis
6. Gate-level netlist
7. Logical verification and testing
8. Floor planning, automatic place & route
9. Physical layout
10. Layout verification
11. Implementation

Typical Design Flow (cont'd)

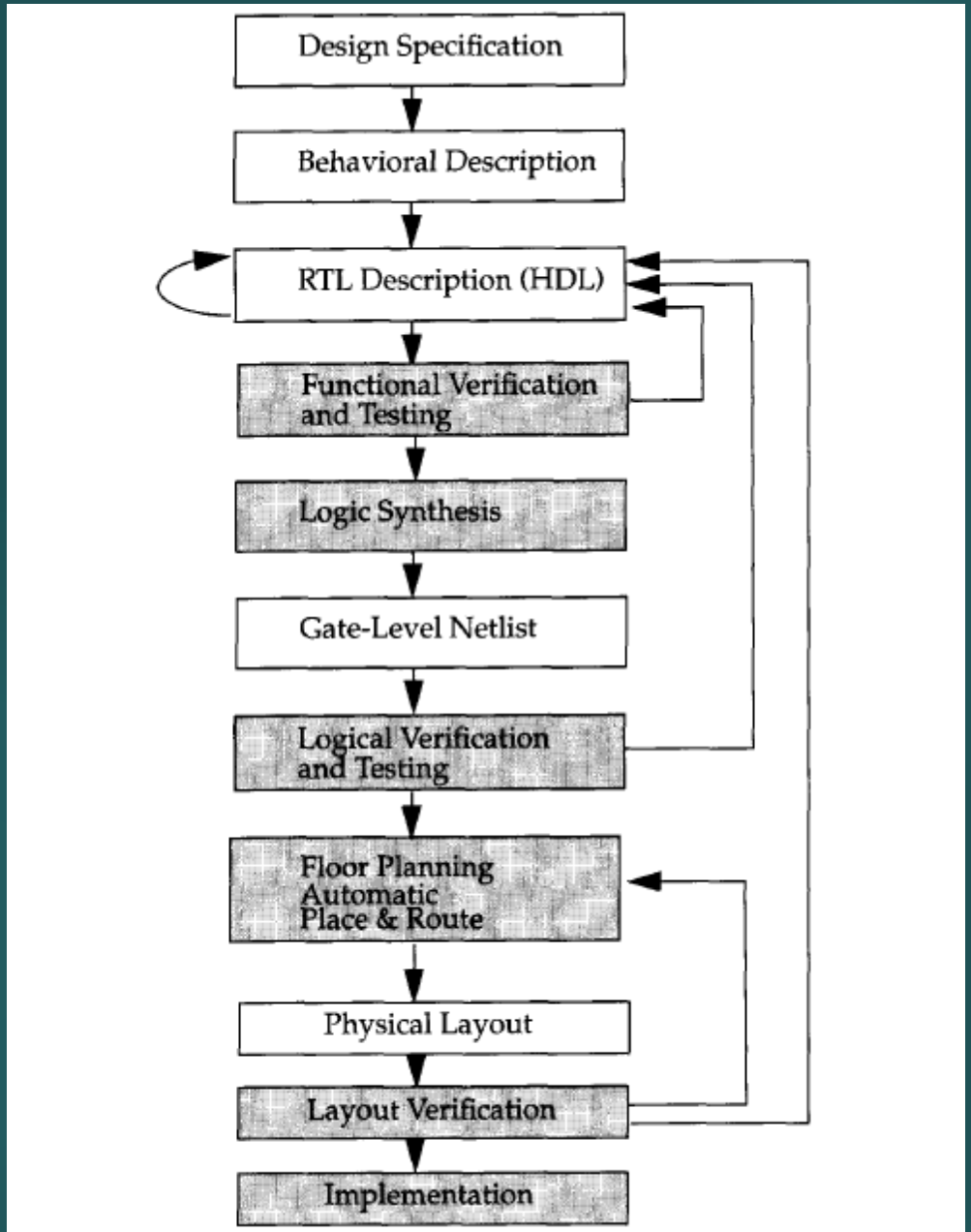
- Most design activity
  - In 1996:
    - Manually optimizing the RTL design
    - CAD tools take care of generating lower-level details
    - Reducing design time to months from years
  - Today
    - Still RTL is used in many cases
    - But, synthesis from behavioral-level also possible
    - Digital design now resembles high-level computer programming

Popularity of Verilog HDL

- Verilog HDL
  - General-purpose
  - Easy to learn, easy to use
  - Similar in syntax to C
  - Allows different levels of abstraction and mixing them
  - Supported by most popular logic synthesis tools
  - Post-logic-synthesis simulation libraries by all fabrication vendors
  - PLI to customize Verilog simulators to designers' needs

Trends in HDLs

- Design at behavioral level
- Formal verification techniques
- Very high speed and time critical circuits
  - e.g. microprocessors
  - Mixed gate-level and RTL designs
- Hardware-Software Co-design
  - System-level languages: SystemC, SpecC, …



Typical Design Flow

Hierarchical Modeling Concepts

Design Methodologies

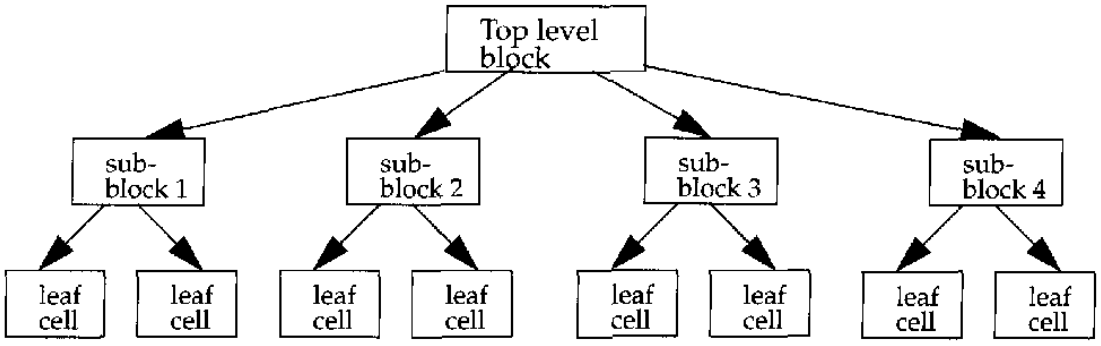


Figure 2-1 Top-down Design Methodology

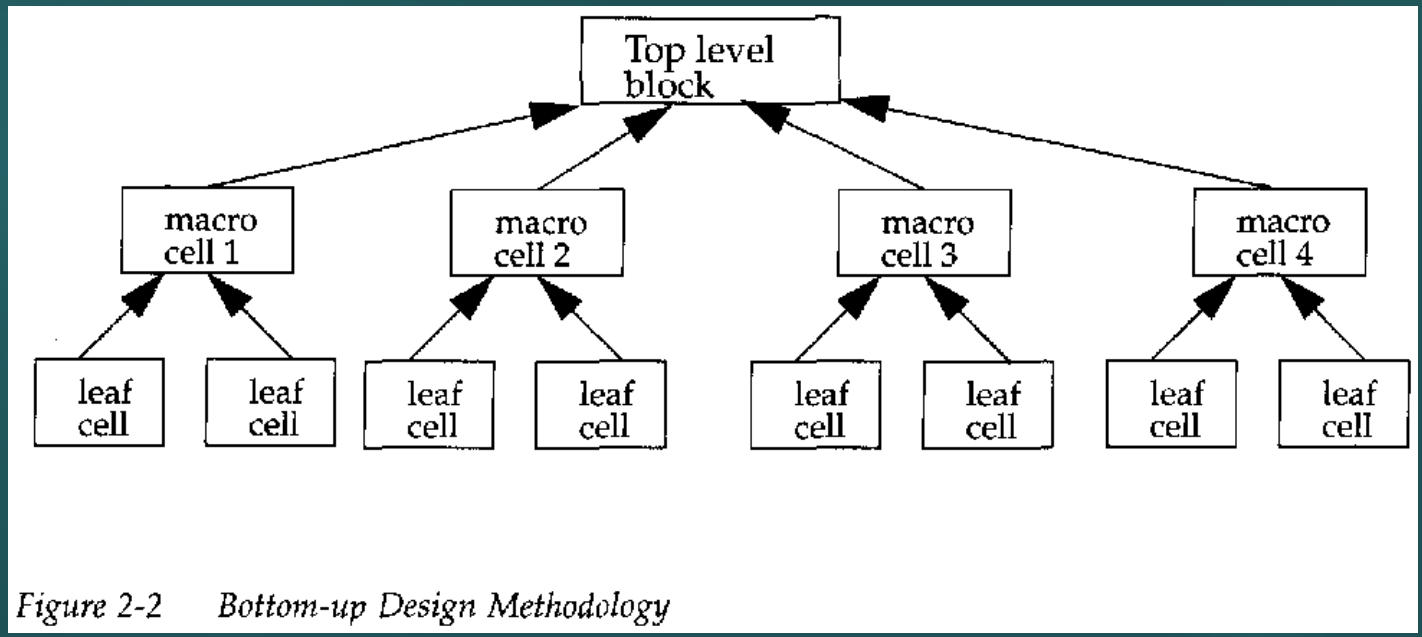


Figure 2-2 Bottom-up Design Methodology

4-bit Ripple Carry Counter

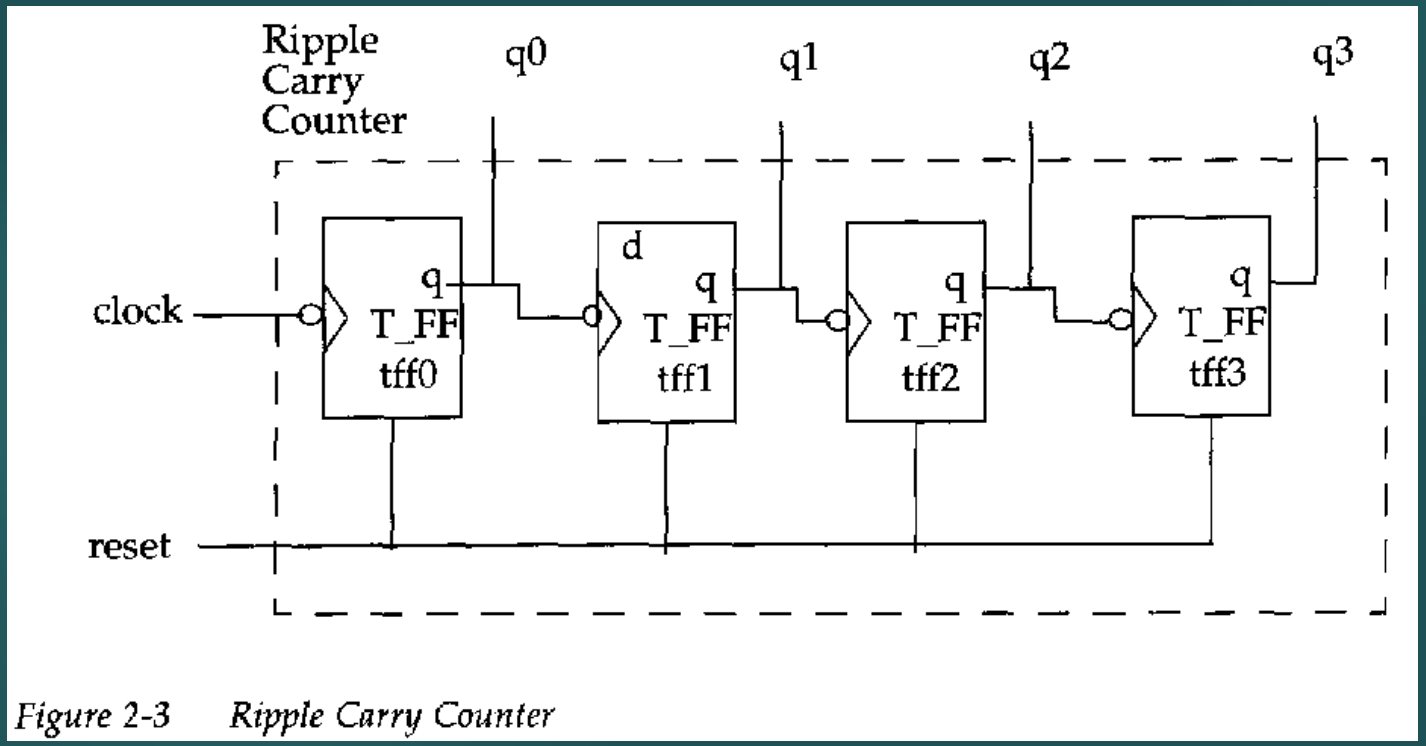


Figure 2-3 Ripple Carry Counter

T-flipflop and the Hierarchy

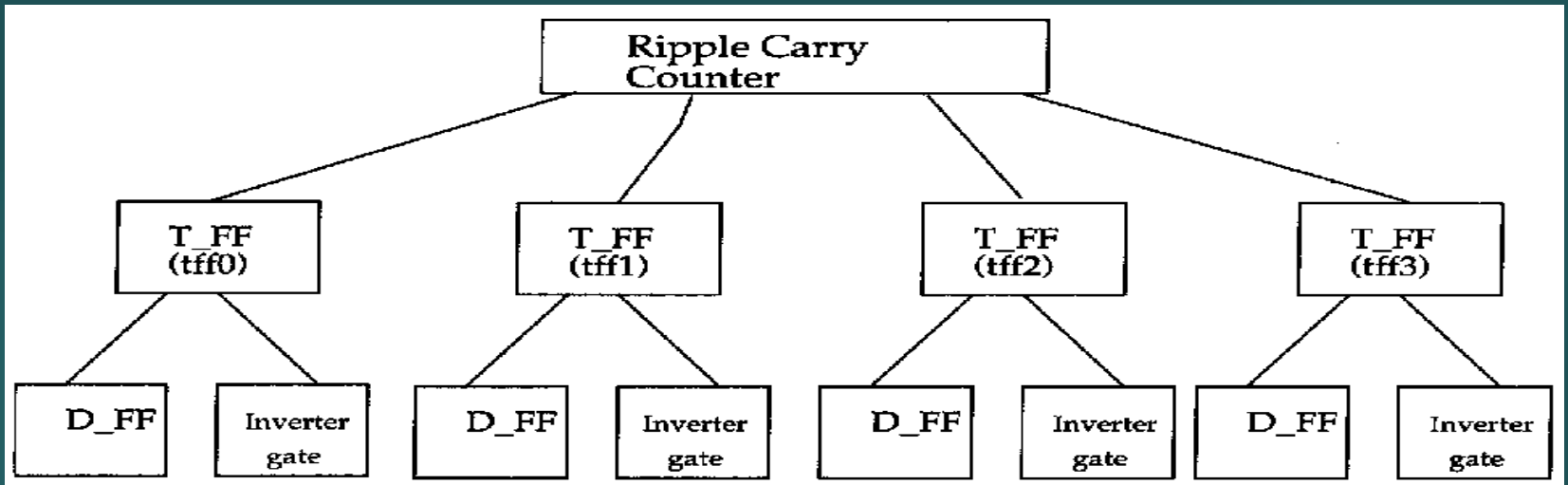


Figure 2-5 Design Hierarchy

Modules


```
module
   <module_name>(<module_terminal_list>);
      ...
      <module internals>
      ...
endmodule
```

- Example:

```
module T_ff(q, clock, reset);
      ...
      <functionality of T_flipflop>
      ...
endmodule
```

Modules (cont'd)

- Verilog supported levels of abstraction
  - Behavioral (algorithmic) level
    - Describe the algorithm used
    - Very similar to C programming
  - Dataflow level
    - Describe how data flows between registers and is processed
  - Gate level
    - Interconnect logic gates
  - Switch level
    - Interconnect transistors (MOS transistors)
- Register-Transfer Level (RTL)
  - Generally known as a combination of behavioral+dataflow that is synthesizable by EDA tools

Basic Concepts
Lexical Conventions

- Very similar to C
  - Verilog is case-sensitive
  - All keywords are in lowercase
  - A Verilog program is a string of tokens
    - Whitespace
    - Comments
    - Delimiters
    - Numbers
    - Strings
    - Identifiers
    - Keywords

Lexical Conventions (cont'd)

- Whitespace
  - Blank space (\b)
  - Tab (\t)
  - Newline (\n)
- Whitespace is ignored in Verilog except
  - In strings
  - When separating tokens

- Comments
  - Used for readability and documentation
  - Just like C:
    - // single line comment
    - /* multi-line comment */

```
/* Nested comments
 /* like this */ may
 not be acceptable
 (depends on Verilog
 compiler) */
```

Lexical Conventions (cont'd)

- Operators
  - Unary

```
a = ~b;
```

  - Binary

```
a = b && c;
```

  - Ternary

```
a = b ? c : d; // the only ternary operator
```

Lexical Conventions (cont'd)

- Number Specification
  - Sized numbers
  - Unsized numbers
  - Unknown and high-impedance values
  - Negative numbers

Lexical Conventions (cont'd)

- Sized numbers
  - General syntax: *<size>'<base><number>*
    - *<size>* number of bits (in decimal)
    - *<number>* is the number in radix *<base>*
    - *<base>* :
      - d or D for decimal (radix 10)
      - b or B for binary (radix 2)
      - o or O for octal (radix 8)
      - h or H for hexadecimal (radix 16)
    - Examples:
      - `4'b1111`
      - `12'habc`
      - `16'd255`
- Unsized numbers
  - Default base is decimal
  - Default size is at least 32 (depends on Verilog compiler)
  - Examples
    - `23232`
    - `'habc`
    - `'o234`

Lexical Conventions (cont'd)

- X or Z values
  - Unknown value: lowercase `x`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - High-impedance value: lowercase `z`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - Examples
    - `12'h13x`
    - `6'hx`
    - `32'bz`
  - Extending the most-significant part
    - Applied when `<size>` is bigger than the specified value
      - Filled with x if the specified MSB is x
      - Filled with z if the specified MSB is z
      - Zero-extended otherwise
    - Examples:
      - `6'hx`

Lexical Conventions (cont'd)

- Negative numbers
  - Put the sign before the `<size>`
  - Examples:
    - `-6'd3`
    - `4'd-2// illegal`
  - Two's complement is used to store the value
- Underscore character and question marks
  - Use '`_`' to improve readability
    - `12'b1111_0000_1010`
    - Not allowed as the first character
  - '`?`' is the same as '`z`' (only regarding numbers)
    - `4'b10?? // the same as 4'b10zz`

Lexical Conventions (cont'd)

- Strings
  - As in C, use double-quotes
  - Examples:
    - `"Hello world!"`
    - `"a / b"`
    - `"text\tcolumn1\bcolumn2\n"`
- Identifiers and keywords
  - identifiers: alphanumeric characters, '_', and '$'
    - Should start with an alphabetic character or '_'
    - Only system tasks can start with '$'
  - Keywords: identifiers reserved by Verilog
  - Examples:
    - `reg value;`
    - `input clk;`

Lexical Conventions (cont'd)

- Escaped identifiers
  - Start with '\'
  - End with whitespace (space, tab, newline)
  - Can have any printable character between start and end
  - The '\' and whitespace are not part of the identifier
  - Examples:
    - `\a+b-c // a+b-c is the identifier`
    - `\**my_name** // **my_name** is the identifier`
  - Used as name of modules

Data Types

- Value set and strengths
- Nets and Registers
- Vectors
- Integer, Real, and Time Register Data Types
- Arrays
- Parameters
- Strings

Value Set

Value level	HW Condition
0	Logic zero, false
1	Logic one, true
x	Unknown
z	High imp., floating

Nets

- Used to represent connections between HW elements
  - Values continuously driven on nets
- Keyword: `wire`
  - Default: One-bit values
    - unless declared as vectors
  - Default value: `z`
    - For `trireg`, default is `x`
  - Examples
    - `wire a;`
    - `wire b, c;`
    - `wire d=1'b0;`

Registers

- Registers represent data storage elements
  - Retain value until next assignment
  - NOTE: this is not a hardware register or flipflop
  - Keyword: `reg`
  - Default value: `x`
  - Example:

```
reg reset;
initial
begin
   reset = 1'b1;
   #100 reset=1'b0;
end
```

Vectors

- Net and register data types can be declared as vectors (multiple bit widths)
- Syntax:
  - `wire/reg [msb_index : lsb_index] data_id;`
- Example

```
wire a;
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg clock;
reg [0:40] virtual_addr;
```

Vectors (cont'd)

- Consider

```
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg [0:40] virtual_addr;
```

- Access to bits or parts of a vector is possible:

```
busA[7]
bus[2:0] // three least-significant
  bits of bus
```

Integer

- Integer
  - Keyword: `integer`
  - Very similar to a vector of `reg`
    - `integer` variables are signed numbers
    - `reg` vectors are unsigned numbers
    - Designer can also specify a width:

```
integer [7:0] tmp;
```

  - Examples:

```
integer counter;
initial
   counter = -1;
```

Arrays

- Only one-dimensional arrays supported
- Allowed for `reg`, `integer`, `time`
  - Not allowed for `real` data type
- Syntax:

```
<data_type> <var_name>[start_idx : end_idx];
```

- Examples:

```
integer count[0:7];
reg bool[31:0];
time chk_point[1:100];
reg [4:0] port_id[0:7];
integer matrix[4:0][4:0]; // illegal
```

Parameters

- Similar to `const` in C
  - But can be overridden for each module at compile-time
- Syntax:

```
parameter <const_id>=<value>;
```

- Gives flexibility
  - Allows to customize the module
- Example:

```
parameter port_id=5;
parameter cache_line_width=256;
parameter bus_width=8;
wire [bus_width-1:0] bus;
```

Strings

- Strings are stored in `reg` variables.
- 8-bits required per character
- The string is stored from the least-significant part to the most-significant part of the `reg` variable
- Example:

```
reg [8*18:1] string_value;
initial
    string_value = "Hello World!";
```

System Tasks

- System Tasks: standard routine operations provided by Verilog
  - Displaying on screen, monitoring values, stopping and finishing simulation, etc.
- All start with `$`

System Tasks (cont'd)

- `$display`: displays values of variables, strings, expressions.
  - Syntax: `$display(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or expression
  - Adds a new-line after displaying `pn` by default
  - Format specifiers:
    - `%d, %b, %h, %o`: display variable respectively in decimal, binary, hex, octal
    - `%c, %s`: display character, string
    - `%e, %f, %g`: display real variable in scientific, decimal, or whichever smaller notation
    - %v: display strength
    - %t: display in current time format
    - %m: display hierarchical name of this module

System Tasks (cont'd)

- `$display` examples:
  - `$display("Hello Verilog World!");`

    Output: `Hello Verilog World!`

  - `$display($time);`

    Output: `230`

  - `reg [0:40] virtual_addr;`
  - `$display("At time %d virtual address is %h", $time, virtual_addr);`

    Output: `At time 200 virtual address is 1fe000001c`

System Tasks (cont'd)

- `reg [4:0] port_id;`
- `$display("ID of the port is %b", port_id);`

  Output: `ID of the port is 00101`

- `reg [3:0] bus;`
- `$display("Bus value is %b", bus);`

  Output: `Bus value is 10xx`

- `$display("Hierarchical name of this module is %m");`

  Output: `Hierarchical name of this module is top.p1`

- `$display("A \n multiline string with a %% sign.");`

  Output:
  ```
  A
          multiline string with a % sign.
  ```

System Tasks (cont'd)

- `$monitor`: monitors a signal when its value changes
- Syntax: `$monitor(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or signal names
  - Format specifiers just as `$display`
  - Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.

System Tasks (cont'd)

- `$stop`: stops simulation
  - Simulation enters interactive mode when reaching a `$stop` system task
  - Most useful for debugging
- `$finish`: terminates simulation
- Examples:

```
initial
begin
   clock=0;
   reset=1;
   #100 $stop;
   #900 $finish;
end
```

Compiler Directives

- General syntax:

```
`<keyword>
```

- `` `define ``: similar to `#define` in C, used to define macros
- `` `<macro_name> `` to use the macro defined by `` `define ``
- Examples:

```
`define WORD_SIZE 32
`define S $stop

`define WORD_REG reg [31:0]
`WORD_REG a_32_bit_reg;
```

Compiler Directives (cont'd)

- `` `include ``: Similar to `#include` in C, includes entire contents of another file in your Verilog source file
- Example:

```
`include header.v
...
<Verilog code in file design.v>
...
```