




Design with Verilog 

Chap 6: Introduction to HDL - 1



Credit to: MD Rizal Othman
Faculty of Electrical & Electronics Engineering
Universiti Malaysia Pahang

Ext: 6036



What is “HDL”?

- HDL = Hardware Description Language
- A text-based method for describing hardware to a *synthesis tool*



Hardware Description Languages (HDL)

- Verilog
 - Gateway Design Automation (1983; proprietary)
 - Acquired by Cadence 1989
 - IEEE standard in 1995 and 2001
- VHDL
 - Origins in DoD VHSIC program (1980's)
 - IEEE standard in 1987





Discussion

→ Verilog vs. VHDL



HDL Advantages Over Schematic Entry

- Produce correct designs in less time
- Produce larger and more complex systems per unit time
- Shifts focus to specifying functionality
- Synthesis tools automate details of connecting gates and devices



Key Advantages of HDL-Based Design Methodology

- Operate at higher level of abstraction
- Can debug earlier (behavioral simulator)
- Parameterized design, easy to make wholesale modifications to a design (e.g., bus width)



Key Advantages of HDL-Based Design Methodology

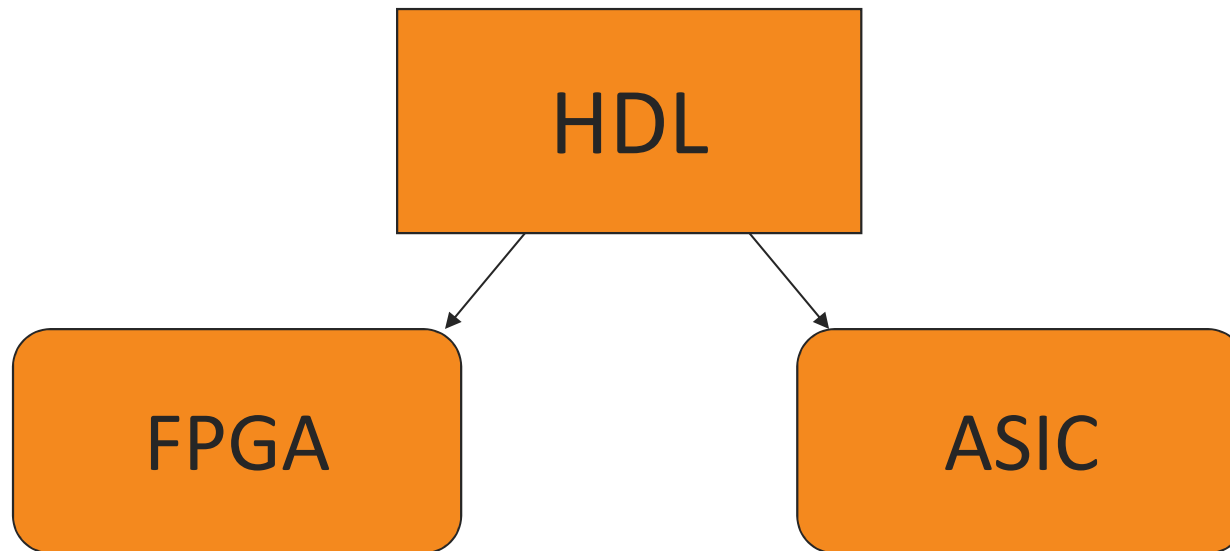
- Can quickly specify desired behavior
 - Example: Up-counter with reset

```
if (reset == 1)
    count <= 0;
else
    count <= count + 1;
```



Key Advantages of HDL-Based Design Methodology

- Can easily target multiple devices (eases product migration)



Key Advantages of HDL-Based Design Methodology

- HDL is more universal than schematic tools
- Promotes *design reuse*
- Promotes integration of third party designs, or *IP (intellectual property)*

What HDL is **NOT**:

HDL is **not** a programming language
(HDL is a *description* language)

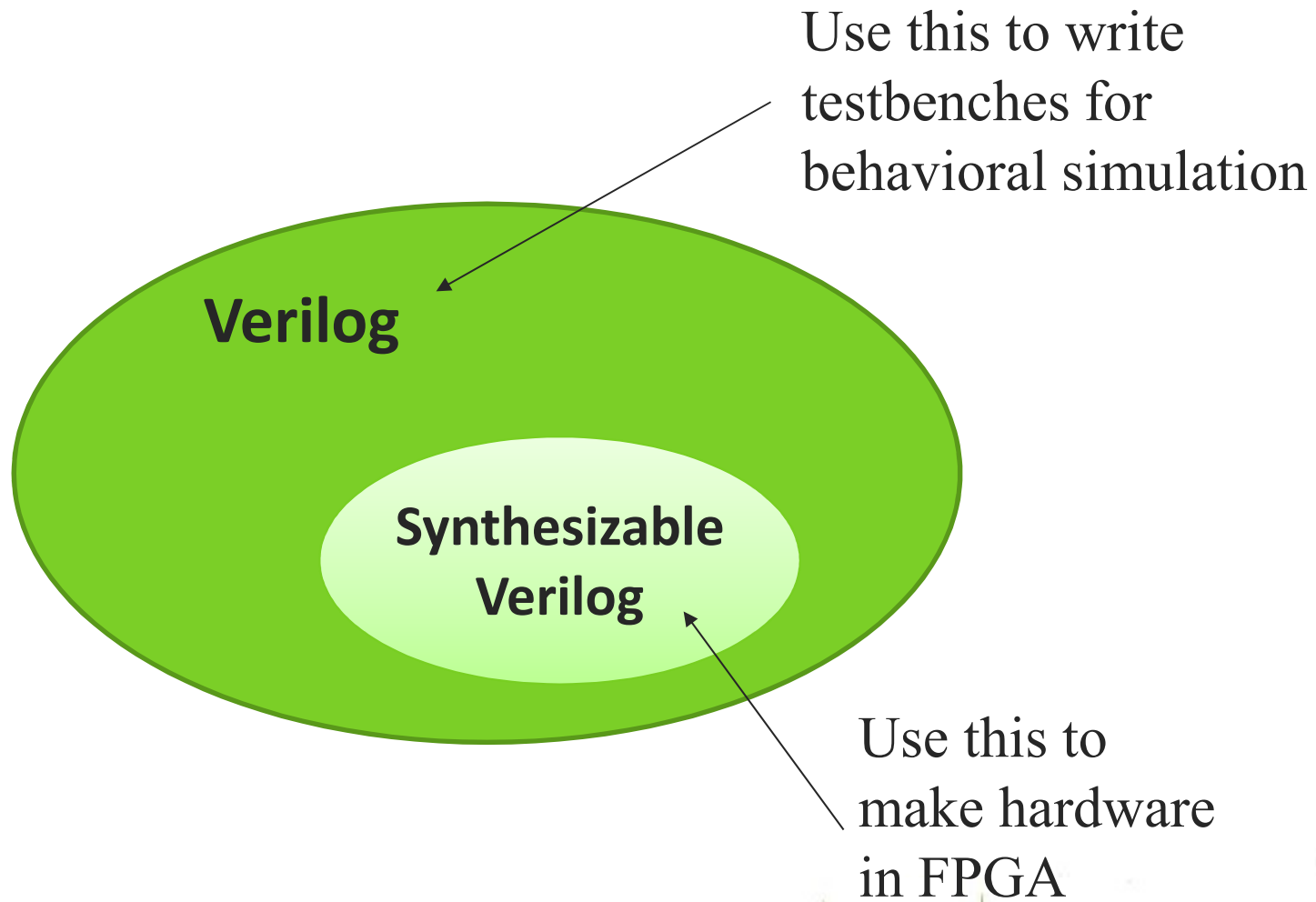


Synthesizable Subset

- Verilog (and VHDL) began as *simulation* and *modeling* tools
- Hardware synthesis developed during the 1990's
- Need to use a subset of Verilog and specific coding styles to allow synthesis tool to *realize* correct hardware



Synthesizable Subset



Design Methodologies

- Two main types of design methodologies:
 - 1) Top-down design
 - 2) Bottom-up design
- 1. Top-down
 - First, top-level block is identified
 - Then, the blocks in the next lower level are defined
 - This process continues until all level in the structure have been defined



2. Bottom-up design

- The smaller cells as building blocks is created first
- Then, tie the smaller blocks together to form a building block of the next level
- This process continues until the top level is reached



Ex. Four-bit Ripple Adder

- Another example of top-down methodology



BASIC LEXICAL ELEMENTS AND DATA TYPES

1. Identifier

1. *identifier is an unique name to an object, such as eq1, x, y or out*
 - composed of letters, digits, the underscore character(-), and the dollar sign (\$)
 - the first character of an identifier **must** be a letter or underscore
 - a good practice to give an object a descriptive name. e.g, **mem-addr-en** is more meaningful than **mae** for a memory address enable signal
 - Verilog is a *case-sensitive language*, thus data-bus, Data-bus, and DATA_BUS refer to three different objects
 - To avoid confusion, refrain from using the case to create different identifiers



2. Keywords

- predefined identifiers that are used to describe language constructs
- Use boldface/blue color for Verilog keywords
- **Ex: module, wire**

```
module eq1
    // I/O ports
    (
        input wire i0, i1,
5      output wire eq
    );

    // signal declaration
    wire p0, p1;
10

    // body
    // sum of two product terms
    assign eq = p0 | p1;
    // product terms
15  assign p0 = ~i0 & ~i1;
    assign p1 = i0 & i1;

endmodule
```

3. White space

- White space, which includes the space, tab, and newline characters, is used to separate identifiers and can be used freely in the Verilog code
- use proper white spaces to format the code and make it more readable

Bad Code : Never write code like this
module addbit(a,b,ci,sum,co); input
a,b,ci;output sum co; wire
a,b,ci,sum,co;endmodule

Good Code : Nice way to write code

```
module addbit ( a, b, ci, sum, co);  
    input a;  
    input b;  
    input ci;  
    output sum;  
    output co;  
    wire a;  
    wire b;  
endmodule
```

4. Comments

- for documentation purposes and will be ignored by software
- Verilog has two forms of comments.
 - A one-line comment starts with //

// This is a comment

- A multiple-line comment is encapsulated between /* and */,

/ This is comment line 1.*

This is comment line 2.

*This is comment line 3. */*



DATA TYPES

1. Four-value system

- Four basic values are used in most data types:
 - 0: for "logic 0", or a false condition
 - 1: for "logic 1", or a true condition
 - z: for the high-impedance state
 - x: for an unknown value
- z value corresponds to the output of a tri-state buffer
- x value is usually used in modeling and simulation, representing a value that is not 0, 1, or z, such as an uninitialized input or output conflict



2. Data type groups

Verilog has two main groups of data types: *net and variable*

2.1 net group

- The data types in the net group represent the physical connections between hardware components
- used **as the outputs** of *continuous assignments* and **as the connection signals** between different modules
- **wire** is the most commonly used data type in this group



-
- A wire represents a physical wire in a circuit and is used to connect gates or modules.
 - A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.
 - *Syntax*
 - wire [msb:lsb] wire_variable_list;
 - wand [msb:lsb] wand_variable_list;
 - wor [msb:lsb] wor_variable_list;
 - tri [msb:lsb] tri_variable_list;



-
- The **wire data type** represents a **1-bit signal**

```
wire p0, p1; // two 1-bit signals
```

- Representing a collection of signals is grouped into a bus

```
wire [31:0]addr; // 32-bit address
```

```
wire [0:7] revers-data; // ascending index should be avoided
```

- Representing a two-dimensional array e.g memory

```
wire [3:0] mem1 [31:0] ; // 32-by-4 memory
```



2.2 variable group

- The data types in the variable group represent abstract storage in behavioral modeling
- used in the outputs of *procedural assignments*
- There are five data types in this group: **reg**, **integer**, **real**, **time**, and **realtime**
- The most commonly used data type in this group is **reg** and it can be synthesized
- The last four data types can only be used in modeling and simulation



-
- Declare type **reg** for all data objects on the left hand side of expressions in procedural assignment (initial and always block), or functions.
 - A **reg** is the data type that must be used for latches, flip-flops and memory.
 - In multi-bit registers, data is stored as *unsigned* numbers and no sign extension is done.
 - **Syntax**
 - **reg [msb:lsb] reg_variable_list;**
 - **Examples**

```
reg a;           // single 1-bit register variable  
reg [7:0] tom;  // an 8-bit vector; a bank of 8 registers.  
reg [5:0] b, c; // two 6-bit variables
```

3. Number representation

- A number/constant in Verilog can be represented in various formats. Its general form is
- [sign] [size] ' [base] [value]
- The [base] term specifies the base of the number, which can be the following:
 - b or B: binary
 - o or O: octal
 - h or H: hexadecimal
 - d or D: decimal



-
- The [value] term specifies the value of the number in the corresponding base. The underline character () can be included for clarity
 - The [size] term specifies the number of bits in a number. It is optional. The number is known as a *sized number* when a [size] term exists and is known as an *unsized number* otherwise.



Sized number

- A sized number specifies the number of bits explicitly. If the size of the value is smaller than the [size] term specified, zeros are padded in front to extend the number. except in several special cases.
- The z or x value is padded if the MSB of the value is z or x, and the MSB is padded if the signed data type is used. Several sized number examples are shown in the top portion of Table 1.2.



Unsigned number

- An unsigned number omits the [size] term.
- Its actual size depends on the host computer but must be at least 32 bits. Assume that 32 bits are used in the host machine.
- The ' [base] term can also be omitted if the number is in decimal format.
- Several unsigned number examples are shown in the bottom portion of Table 1.2.



number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	0000000000000000000000000000000011010	extended to 32 bits
'hee	0000000000000000000000000000000011101110	extended to 32 bits
1	0000000000000000000000000000000000000001	extended to 32 bits
-1	11	extended to 32 bits

Program Skeleton

- As its name indicates, HDL used to describe hardware. When we develop or examine a Verilog code, it is much easier to comprehend if we think in terms of “hardware organization” rather than “sequential algorithm.”
- The basic program skeleton for Verilog coding as Code 1.1:

```
module eq1
    // I/O ports
    (
        input wire i0, i1,
        output wire eq
    );

    // signal declaration
    wire p0, p1;

    // body
    // sum of two product terms
    assign eq = p0 | p1;
    // product terms
    assign p0 = ~i0 & ~i1;
    assign p1 = i0 & i1;

endmodule
```

Code 1.1

Port declaration

- The module declaration and port declaration of Code 1.1 are:

```
module eq1
(
input wire iO, il,
output wire eq
);
```

- The i/o declaration specifies the modes, data types, and names of the module's i/o ports. The simplified syntax is

```
module [module-name]
(
[mode] [data-type] [port-names] ,
[mode] [data-type] [port-names] ,
...
[mode] [data-type] [port-names]
);
```

- The [mode] term can be **input**, **output**, or **inout**, which represent the input, output, or bidirectional port, respectively. Note that there is no comma in the last declaration.
- The [data-type] term can be omitted if it is wire.

Program body

- Unlike a program in the C language, in which the statements are executed sequentially, the program body of a synthesizable Verilog module can be thought of as a collection of circuit parts. These parts are operated in parallel and executed concurrently.
- There are several ways to describe a part:
 - Continuous assignment
 - "Always block"
 - Module instantiation



- The first way to describe a circuit part is by using a continuous assignment. It is useful for simple combinational circuits. Its simplified syntax is:

```
assign [signal_name] = [expression] ;
```

- Each continuous assignment can be thought as a circuit part. The signal on the left-hand side is the output and the signals used in the right-hand-side expression are the inputs.
- The expression describes the function of this circuit. For example, consider the statement:

```
assign eq = p0 | p1;
```

- It is a circuit that performs the or operation. When p0 or p1 changes its value, this statement is activated and the expression is evaluated. The new value is assigned to eq after



- The second way to describe a circuit part is by using an *always block*. More abstract *procedural assignments* are used inside the *always block* and thus it can be used to describe more complex circuit operation. The always block will be discussed in next section.
- The third way to describe a circuit part is by using module instantiation. Instantiation creates an instance of another module and allows us to incorporate predesigned modules as subsystems of the current module. Instantiation will be discussed in next section.



Signal declaration

- The declaration portion specifies the internal signals and parameters used in the module.
- The internal signals can be thought of as the interconnecting wires between the circuit parts,
- as shown in Figure 1.1.
- The simplified syntax of signal declaration is
- **[data_type] [port_names];**
- Two internal signals are declared in Code 1.1 :

```
wire p0, p1;
```

