Digital System Design

Fundamentals of Hardware Description Language

Lecturer: Yu-Hao Chen (陳郁豪)
Date: 2011.03.02

Based on: Ch.1-3 of the textbook
Review: Logic Design (Concepts of Propagation Delay)
Outline

- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification
Hardware Description Language (1/2)

✓ Hardware Description Language (HDL) is any language from a class of computer languages and/or programming languages for formal description of electronic circuits, and more specifically, digital logic.

✓ HDL can
  ✓ describe the circuit’s operation, design, organization
  ✓ verify its operation by means of simulation.

✓ Now HDLs usually merge Hardware Verification Language, which is used to verify the described circuits.

From Wikipedia
HDLs are used to write executable specifications of some piece of hardware.

- Designed to implement the semantics of the language statements with native supporting to simulate the progress of time.
- Being executable
- Provides the hardware designer the ability to model a piece of hardware before it is created physically.

Supporting discrete-event (digital) or continuous-time (analog) modeling, e.g.:

- SPICE, Verilog HDL, VHDL, SystemC
High-Level Programming Language

- It’s possible to describe a hardware (operation, structure, timing, and testing methods) in C/C++/Java, why do we use HDL?
- The efficiency (to model/verify) does matter.
- Native support to concurrency
- Native support to the simulation of the progress of time
- Native support to different model of signal states
- The required level of detail determines the language we use.
List of HDL for Digital Circuits

<table>
<thead>
<tr>
<th>HDL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog</td>
<td></td>
</tr>
<tr>
<td>VHDL</td>
<td></td>
</tr>
<tr>
<td>ABEL</td>
<td>Advanced Boolean Expression Language (ABEL)</td>
</tr>
<tr>
<td>AHDL</td>
<td>Altera HDL, a proprietary language from Altera</td>
</tr>
<tr>
<td>Atom</td>
<td>behavioral synthesis and high-level HDL based on Haskell</td>
</tr>
<tr>
<td>Bluespec</td>
<td>high-level HDL originally based on Haskell, now with a SystemVerilog syntax</td>
</tr>
<tr>
<td>Confluence</td>
<td>functional HDL; has been discontinued</td>
</tr>
<tr>
<td>CUPL</td>
<td>proprietary language from Logical Devices, Inc.</td>
</tr>
<tr>
<td>Handel-C</td>
<td>a C-like design language</td>
</tr>
<tr>
<td>C-to-Verilog</td>
<td>Converts C to Verilog</td>
</tr>
<tr>
<td>HDCCaml</td>
<td>based on Objective Caml</td>
</tr>
<tr>
<td>Hardware Join Java</td>
<td>based on Join Java</td>
</tr>
<tr>
<td>HML</td>
<td>based on SML</td>
</tr>
<tr>
<td>Hydra</td>
<td>based on Haskell</td>
</tr>
<tr>
<td>Impulse C</td>
<td>another C-like language</td>
</tr>
<tr>
<td>JHDL</td>
<td>based on Java; based on Haskell</td>
</tr>
<tr>
<td>Lola</td>
<td>a simple language used for teaching</td>
</tr>
<tr>
<td>MyHDL</td>
<td>based on Python</td>
</tr>
<tr>
<td>PALASM</td>
<td>for Programmable Array Logic (PAL) devices</td>
</tr>
<tr>
<td>Ruby</td>
<td>hardware description language</td>
</tr>
<tr>
<td>RHDL</td>
<td>based on the Ruby programming language SDL based on Tcl.</td>
</tr>
<tr>
<td>CoWareC</td>
<td>a C-based HDL by CoWare. Now discontinued in favor of SystemC</td>
</tr>
<tr>
<td>SystemVerilog</td>
<td>superset of Verilog, with enhancements to address system-level design and verification</td>
</tr>
<tr>
<td>SystemC</td>
<td>standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level</td>
</tr>
<tr>
<td>SystemTCL, SDL</td>
<td>based on Tcl.</td>
</tr>
</tbody>
</table>
About Verilog

- Introduction on 1984 by Phillip Moorby and Prabhu Goel in Automated Integrated Design System (renamed to Gateway Design Automation and bought by Cadence Design Systems)

- Open and Standardize (IEEE 1364-1995) on 1995 by Cadence because of the increasing success of VHDL (standard in 1987)

- Become popular and makes tremendous improvement on productivity
  - Syntax similar to C programming language, though the design philosophy differs greatly
History/Branch of Verilog

- **Verilog**
  - IEEE Std 1364-1995
- **Verilog-2001**
  - IEEE Std 1364-2001
- **Verilog-2005**
  - IEEE Std 1364-2005
- **SystemVerilog**
  - IEEE Std 1800-2005

Digital-signal HDL
Outline

- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification
Hierarchical Modeling Concept

- Introduce *top-down* and *bottom-up* design methodologies
- Introduce *module* concept and encapsulation for hierarchical modeling
- Explain differences between *modules* and *module instances* in Verilog
Top-down Design Methodology

- We define the top-level block and identify the sub-blocks necessary to build the top-level block.
- We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.
Bottom-up Design Methodology

- We first identify the building block that are available to us.
- We build bigger cells, using these building blocks.
- These cells are then used for higher-level blocks until we build the top-level block in the design.
Example: 16-bit Adder
Hierarchical Modeling in Verilog

- A Verilog design consists of a hierarchy of modules.

- Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports.

- Internally, a module can contain any combination of the following
  - net/variable declarations (wire, reg, integer, etc.)
  - concurrent and sequential statement blocks
  - instances of other modules (sub-hierarchies).
Design Encapsulation

- Encapsulate structural and functional details in a module

```vhdl
module <ModuleName> (<PortName List>);

// Structural part
<List of Ports>
<Lists of Nets and Registers>
<SubModule List>  <SubModule Connections>

// Behavior part
<Timing Control Statements>
<Parameter/Value Assignments>
<Stimuli>
<System Task>
endmodule
```

- Encapsulation makes the model available for instantiation in other modules
Module

- Basic building block in Verilog.

- Module
  1. Created by “declaration” (can’t be nested)
  2. Used by “instantiation”

- Interface is defined by ports

- May contain instances of other modules

- All modules run concurrently
Instances

- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.
Module Instantiation

```
module adder(out, in1, in2);
    output out;
    input in1, in2, sel;
    assign out = in1 + in2;
endmodule
```

```
module adder_tree (out0, out1, in1, in2, in3, in4);
    output out0, out1;
    input in1, in2, in3, in4;
    adder add_0 (out0, in1, in2);
    adder add_1 (out1, in3, in4);
endmodule
```
## Analogy: module ↔ class

As *module* is to *Verilog HDL*, so *class* is to *C++ programming language*.

| Syntax         | module $m_{Name}$( IO list );
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><code>endmodule</code></td>
</tr>
<tr>
<td></td>
<td>class $c_{Name}$ {</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><code>};</code></td>
</tr>
</tbody>
</table>
| Instantiation  | $m_{Name}$ $ins_{name}$ ( port
|                | connection list );         |
|                | $c_{Name}$ $obj_{name}$;    |
| Member         | $ins_{name}$.member_signal  |
|                | $obj_{name}$.member_data    |
| Hierachy       | instance.sub_instance.memb
|                | er_signal                  |
|                | object.sub_object.member_dat
|                | a                          |
Analogy: module ↔ class

Model AND gate with C++

```c
class c_AND_gate {
    bool in_a;
    bool in_b;
    bool out;
    void evaluate() { out = in_a && in_b; }
};
```

Model AND gate with Verilog HDL

```verilog
module m_AND_gate (in_a, in_b, out);
    input in_a;
    input in_b;
    output out;
    assign out = in_a & in_b;
endmodule
```

assign and evaluate() is simulated/called at each $T_{i+1} = T_i + t_{\text{resolution}}$
Outline

- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification
Cell-Based Design and Levels of Modeling

**Behavioral Level**
- Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools.

**Register Transfer Level (RTL)**
- Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level.

**Structural/Gate Level**
- Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library.

**Transistor/Physical Level**
- Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE.
Tradeoffs Among Modeling Levels

- Each level of modeling permits modeling at a higher or lower level of detail. More detail means more efforts for designers and the simulator.

- Always keep in mind which level of modeling is adopted.
An Example
1-bit Multiplexer

To “select” output

if (sel==0) out = in1;
else out = in2;

out = (sel’ & in1) + (sel & in2)
Gate Level Description

Gate Level: you see only netlist (gates and wires) in the code

```verilog
module mux2(out,in1,in2,sel);
    output out;
    input in1,in2,sel;
    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or o1(out,a1_o,a2_o);
endmodule
```
Behavioral Level/RTL Description

RTL: describe logic/ arithmetic function between input node and output node

Behavior: RTL in an event-driven behavior description construct
Outline

- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification
Verilog Language Rules

Verilog is a **case sensitive** language (with a few exceptions)

**Identifiers** (space-free sequence of symbols)
- upper and lower case letters from the alphabet
- digits \((0, 1, ..., 9)\)
- underscore ( \(_)\)
- \$ symbol (for system tasks)
- Max length of 1024 symbols

Terminate lines with semicolon **;**

Single line comments:
- // A single-line comment goes here

Multi-line comments:
- /* Do not */ nest multi-line comments */ like this */
Verilog Basis Cell

- Verilog Basis Components
  - parameter declarations
  - nets or reg declarations
  - port declarations
  - Continuous assignments
  - Module instantiations
  - Gate instantiations
  - Function definitions
  - always blocks
  - task statements
Port Declaration

- Three port types
  - Input port
    - `input a;`
  - Output port
    - `output b;`
  - Bi-direction port
    - `inout c;`
Data Types

- nets are further divided into several net types:
  - wire, wand, wor, tri, triand, trior, supply0, supply1
- registers – variable to store a logic value for event-driven simulation - reg
- integer - supports computation 32-bits signed
- time - stores time 64-bit unsigned
- real - stores values as real numbers
- realtime - stores time values as real numbers
Net Types

- The most common and important net types
  - Wire and tri
    - for standard interconnection wires
  - Supply 1 and supply 0

- Other wire types
  - Wand, wor, triand, and trior
    - for multiple drivers that are wired-anded and wired-ored
  - Tri0 and tri1
    - pull down and pull up
  - Trireg
    - for net with capacitive storage
    - If all drivers at z, previous value is retained
Register Types

- **reg**
  - any size, unsigned

- **integer (not synthesizable)**
  - integer a,b; // declaration
  - 32-bit signed (2’s complement)

- **time (not synthesizable)**
  - 64-bit unsigned, behaves like a 64-bit reg
  - $display("At \%t, value=\%d",$time,val_now)

- **real, realtime (not synthesizable)**
  - real c,d; //declaration
  - 64-bit real number
  - Defaults to an initial value of 0
Integer, Real, & Time

- Data types not for hardware description
  - For simulation control, data, timing extraction.

- integer counter;
  - initial counter = -1;

- real delta;
  - initial delta = 4e10;

- time sim_time;
  - initial sim_time = $time;
Wire & Reg

- `wire(wand, wor, tri)`
  - Physical wires in a circuit
  - Cannot assign a value to a wire within a function or a `begin.....end` block

- A wire does not store its value, it must be driven by
  - connecting the wire to the output of a gate or module
  - assigning a value to the wire in a continuous assignment

- An un-driven wire defaults to a value of `Z` (high impedance).
- Input, output, inout port declaration -- wire data type (default)
Wire & Reg

- **reg**
  - A event driven variable in Verilog

- Use of “reg” data type is **not** exactly stands for a really DFF.

- Use of wire & reg
  - When use “wire” usually use “assign” and “assign” **does not** appear in “always” block
  - When use “reg” only use “a=b”, always appear in “always” block

```verilog
module test(a,b,c,d);
input a,b;
output c,d;
reg d;
assign c=a;
always @(b)
d=b;
endmodule
```
Nets-Wired Logic

- The family of nets includes the types `wand` and `wor`
  - A `wand` net type resolves multiple driver as wired-and logic, e.g. open collector technology
  - A `wor` net type resolves multiple drivers as wired-or logic, e.g. emitter-coupled technology

- The family of nets includes `supply0` and `supply1`
  - `supply0` has a fixed logic value of 0 to model a ground connection
  - `supply1` has a fixed logic value of 1 to model a power connection

- Used when model at `transistor-level`
More about Nets: Tri

- **tri**: a data type identified distinctively to indicate that it will be thi-state (Z) in hardware, same functionality as **wire**
- **triand** and **trior**: similar as **wand** and **wor**
- **tri0**: resistive pull-down
- **tri1**: resistive pull-up
- **trireg**: a net models the charge stored on a physical net

- Used when model at **transistor-level**
Data Type - Examples

reg a; // scalar register
wand b; // scalar net of type “wand”
reg [3:0] c; // 4-bit register
tri [7:0] bus; // tri-state 8-bit bus
reg [1:4] d; // 4-bit
trireg (small) store; // specify logical strength (rare used)
Vector

- wire and reg can be defined vector, default is 1bit
- vector is multi-bits element
- Format: [High#:Low#] or [Low#:High#]
- Using range specify part signals

```vhdl
wire  a;  // scalar net variable, default
wire [7:0] bus;  // 8-bit bus
reg  clock;  // scalar register, default
reg [0:23] addr;  // Vector register, virtual address 24 bits wide

bus[7]  // bit #7 of vector bus
bus[2:0]  // Three least significant bits of vector bus
            // using bus[0:2] is illegal because the significant bit should
            // always be on the left of a range specification
addr[0:1]  // Two most significant bits of vector addr
```
Array

Arrays are allowed in Verilog for reg, integer, time, and vector register data types.

Multidimensional array are not permitted in Verilog.

```
integer count[0:7]; // An array of 8 count variables
reg bool[31:0]; // Array of 32 one-bit Boolean register variables
time chk_ptr[1:100]; // Array of 100 time checkpoint variables
reg [4:0] port_id[0:7]; // Array of 8 port_id, each port_id is 5 bits wide
integer matrix[4:0][4:0] // Illegal declaration

count[5] // 5th element of array of count variables
chk_ptr[100] // 100th time check point value
port_id[3] // 3rd element of port_id array. This is a 5-bit value
```
Memories

- In digital simulation, one often needs to model register files, RAMs, and ROMs.
- Memories are modeled in Verilog simply as an array of registers.
- Each element of the array is known as a word, each word can be one or more bits.
- It is important to differentiate between
  - n 1-bit registers
  - One n-bit register

```verilog
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
reg [7:0] mem1byte[0:1023]; // Memory mem1byte with 1K 8-bit words

mem1bit[255] // Fetches 1 bit word whose address is 255
Mem1byte[511] // Fetches 1 byte word whose address is 511
```
Arrays Extended after Verilog-2001

- In Verilog-1995, only reg, integer, and time can be declared as array. Array is limited to 1D.
- In Verilog-2001, arrays of real, realtime, and any type of net are allowed.

//bit
reg r_1bit;

//vector
reg [3:0] r_4bit_vec;

//1D array: 8bit×8
wire [7:0] w_net [7:0];

//2D array: 4bit×8×8
trireg [3:0] row_col_addr [0:7][0:7];

//1D array: memory: 32bit×8
reg [31:0] r_memory [7:0];

//3D array: 100×16×4 float variables
real float_array [0:99][1:16][10:13];
Strings

✓ String: a sequence of 8-bits ASCII values

module string;
    reg [8*14:1] strvar;
initial
begin
    strvar = “Hello World”; // stored as 000000486561…726c64
    strvar = “Hello World!!”; // stored as 00486561…726c642121
end
endmodule

✓ Special characters

\n È newline \t È tab character
\\ È \ character ” È “ character
%% È % character \abc È ASCII code
Four-Valued Logic

- Verilog’s nets and registers hold four-valued data:
  - 0 represent a logic zero or false condition
  - 1 represent a logic one or true condition
  - \( z \) represents an output of an undriven tri-state driver – high-impedance value
  - \( \bar{z} \) models case where nothing is setting a wire’s value
  - \( x \) models when the simulator can’t decide the value – uninitialized or unknown logic value
    - Initial state of registers
    - When a wire is being driven to 0 and 1 simultaneously
    - Output of a gate with \( z \) inputs
Logic System in Verilog

- Four values: 0, 1, x or X, z or Z  // Not case sensitive here
  - The logic value x denotes an unknown (ambiguous) value
  - The logic value z denotes a high impedance

- Primitives have built-in logic

- Simulators describe 4-value logic (see Appendix A in text)
Resolution of Contention Between Drivers

- The value on a wire with multiple drivers in contention may be \( x \)
Logic Strength Levels

- Types of strengths
  - Charge strength: trireg (large>medium>small)
  - Drive strength: <Net> (supply>strong>pull>weak)

- Syntax

```
<NetType> <Strength> <Range> <Delay> <Variables>
trireg (large) [1:4] #5 c1;
```

- Strength level

```
weakest strongest
highz small medium weak large pull strong supply
```
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 0, 1, x or z, will extend 0, 1, x or z (defined later!)
Number Representation

Examples:
- 6'b010_111 gives 010111
- 8'b0110 gives 00000110
- 4'bx01 gives xx01
- 16'H3AB gives 0000001110101011
- 24 gives 0...0011000
- 5'O36 gives 11100
- 16'Hx gives xxxxxxxxxxxxxxx
- 8'hz gives zzzzzzzzz
Vector Concatenations

A easy way to group vectors into a larger vector

<table>
<thead>
<tr>
<th>Representation</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>{cout, sum}</td>
<td>{cout, sum}</td>
</tr>
<tr>
<td>{b[7:4],c[3:0]}</td>
<td>{b[7], b[6], b[5], b[4], c[3], c[2], c[1], c[0]}</td>
</tr>
<tr>
<td>{a,b[3:1],c,2'b10}</td>
<td>{a, b[3], b[2], b[1], c, 1'b1, 1'b0}</td>
</tr>
<tr>
<td>{4{2'b01}}</td>
<td>8'b01010101</td>
</tr>
<tr>
<td>{{8{byte[7]}}.byte}</td>
<td>Sign extension</td>
</tr>
</tbody>
</table>
Parameter Declaration

- Parameters are not variables, they are constants. (hardware design view)
- Can be defined as a bit or a vector
- Typically parameters are used to specify conditions, states, width of vector, entry number of array, and delay

```verilog
module var_mux(out, i0, i1, sel);
    parameter width = 2, flag = 1'b1;
    output [width-1:0] out;
    input [width-1:0] v0, v1;
    input sel;

    assign out = sel==flag ? v1 : v0;
endmodule
```

- If sel = 1, then v1 will be assigned to out;
- If sel = 0, then v0 will be assigned to out;
Overriding the Values of Parameters

- Module instance parameter value assignment.
- Cannot skip any parameter assignment even you do not want to reassign it.

```verilog
module top;
    ......
    wire [1:0] a_out, a0, a1;
    wire [3:0] b_out, b0, b1;
    wire [2:0] c_out, c0, c1;
    var_mux U0(a_out, a0, a1, sel);
    var_mux #(4,2) U1(b_out, b0, b1, sel);
    var_mux #(3, ) U2(c_out, c0, c1, sel);
    ......
endmodule
```

The order of assign of parameters follows the order of declaration of Parameters in the module.

You cannot skip the delay parameter assignment.
You can use `defparam` to group all parameter value override assignment in one module.

```verilog
module top;
    ....
    wire [1:0] a_out, a0, a1;
    wire [3:0] b_out, b0, b1;
    wire [2:0] c_out, c0, c1;

    var_mux U0(a_out, a0, a1, sel);
    var_mux U1(b_out, b0, b1, sel);
    var_mux U2(c_out, c0, c1, sel);
    ....
endmodule

defparam
top.U0.width = 2;
top.U0.delay = 1;
top.U1.width = 4;
top.U1.delay = 2;
top.U2.width = 3;
top.U2.delay = 1;
```
Primitives

- Verilog has predefined logical elements called Primitives

- Smallest modeling block for simulator
  - Behavior as software execution in simulator, not hardware description

- Verilog build-in primitive gate
  - `and, or, not, buf, xor, nand, nor, xnor`
  - `prim_name inst_name( output, in0, in1,.... );`

- User defined primitive (UDP)
  - Building block defined by designer
Structural Models

- Verilog primitives encapsulate pre-defined functionality of common logic gates.
- The counterpart of a schematic is a structural model composed of Verilog primitives.
- Model structural detail by instantiating and connecting primitives.

```
module AOI_str (y_out, x_in1, x_in2, x_in3, x_in4, x_in5);
output y_out;
input x_in1, x_in2, x_in3, x_in4, x_in5;
wire y1, y2;
nor (y_out, y1, y2);
and (y1, x_in1, x_in2);
and (y2, x_in3, x_in4, x_in5);
endmodule
```
Wires in Verilog establish connectivity between primitives and/or modules.

Data type: nets (Example: `wire`)

The logic value of a `wire` (net) is determined dynamically during simulation by what is connected to the wire.

An undeclared identifier is treated by default as a `wire`.

Use nets to establish structural connectivity.

Port connection by name

- `Add_half_0_delay M1(.b(b),.c_out(w2),.a(a),.sum(w1));`

Port connection by place

- `Add_half_0_delay M1(w1, w2, a, b);`
Hierarchical Design Example

Model complex structural detail by instantiating modules within modules

```
module Add_full_0_delay (sum, c_out, a, b, c_in);
    input a, b, c_in;
    output c_out, sum;
    wire w1, w2, w3;
    Add_half_0_delay M1 (w1, w2, a, b);
    Add_half_0_delay M2 (sum, w3, c_in, w1);
    or (c_out, w2, w3);
endmodule
```

MODELING TIP
Use nested module instantiations to create a top-down design hierarchy.

MODELING TIP
The ports of a module may be listed in any order. The instance name of a module is required.
Gate and Switch Level Modeling

- Coding style: **Netlist, connect primitives/modules by nets (wire)**

- Primitives: bottom level of the hierarchy
  - Verilog Gate Level Primitives
  - User-defined Primitives (UDP): described by truth table

- Conventional modules
  - Behavior/RTL statements
  - Structural statements
  - Switch Level Modeling (using transistors)
### Verilog Built-in Primitives

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ideal MOS switch</th>
<th>Resistive gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
<td>nmos</td>
</tr>
<tr>
<td>nand</td>
<td>not</td>
<td>pmos</td>
</tr>
<tr>
<td>or</td>
<td>bufif0</td>
<td>cmos</td>
</tr>
<tr>
<td>nor</td>
<td>bufif1</td>
<td>tran</td>
</tr>
<tr>
<td>xor</td>
<td>notif0</td>
<td>tranif0</td>
</tr>
<tr>
<td>xnor</td>
<td>notif1</td>
<td>tranif1</td>
</tr>
</tbody>
</table>

- **Ideal MOS switch:**
  - and: buf
  - nand: not
  - or: bufif0
  - nor: bufif1
  - xor: notif0
  - xnor: notif1

- **Resistive gates:**
  - pullup
  - pulldown
  - tran
  - tranif0
  - tranif1
  - rtran
  - rtranif0
  - rtranif1
Switch Level Modeling

Two types of MOS switches can be defined with the keywords, **nmos** and **pmos**

- **nmos** is used to model NMOS transistors
  
  ```
  nmos n1(out, data, control);
  ```

- **pmos** is used to model PMOS transistors
  
  ```
  pmos p1(out, data, control);
  ```

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>1</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Z</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Z</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

H: stands for 1 or z
L: stands for 0 or z
CMOS Switches

- CMOS switches are declared with the keyword `cmos`.
- A `cmos` device can be modeled with a `nmos` and a `pmos` device.
  - `cmos c1(out, data, ncontrol, pcontrol);`
- The `cmos` gate is essentially a combination of two gates: one `nmos` and one `pmos`.
  - `nmos n1(out, data, ncontrol);`
  - `pmos p1(out, data, pcontrol);`
Bidirectional Switches

- NMOS, PMOS, CMOS gates conduct from drain to source.
- It is important to have devices that conduct in both directions.
- In such cases, signals on either side of the device can be the driver signal.
- Bidirectional switches are typically used to provide isolation between buses or signals.

- `tran t1(inout1, inout2);`
- `tranif0 t2(inout1, inout2, control);`
- `tranif1 t3(inout1, inout2, control);`
Power and Ground

- The power (Vdd, logic 1) and Ground (Vss, logic 0) sources are needed when transistor-level circuits are designed.
- Supply1 are equivalent to Vdd in circuits and place a logical 1 on a net.
- Supply0 are equivalent to ground or Vss in circuits and place a logical 0 on a net.

```verilog
supply1 vdd;
supply0 gnd;
assign a = vdd;  // connect a to vdd
assign b = gnd;  // connect b to gnd
```
Resistive Switches

- Resistive switches have the same syntax as regular switches.
- Resistive devices have a high source-to-drain impedance. Regular switches have a low source-to-drain impedance.
- Resistive switches reduce signal strengths when signals pass through. Regular switches retain strength levels of signals from input to output.

<table>
<thead>
<tr>
<th>Input Strength</th>
<th>Output Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply</td>
<td>Pull</td>
</tr>
<tr>
<td>Strong</td>
<td>Pull</td>
</tr>
<tr>
<td>Pull</td>
<td>Weak</td>
</tr>
<tr>
<td>Weak</td>
<td>Medium</td>
</tr>
<tr>
<td>Large</td>
<td>Medium</td>
</tr>
<tr>
<td>Medium</td>
<td>Small</td>
</tr>
<tr>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>
Switches Example

```
// Define our own nor gate: nor_sw
module nor_sw (out, a, b);
output out;
input a, b;
// internal wires
wire c;
// set up power and ground lines
supply1 pwr;
supply0 gnd;
// instantiate pmos switches
pmos (c, pwr, b);
pmos (out, c, a);
// instantiate nmos switches
nmos (out, gnd, a);
nmos (out, gnd, b);
endmodule
```
Gate Level Modeling

- Steps
  - Develop the boolean function of output
  - Draw the circuit with logic gates/primitives
  - Connect gates/primitives with net (usually wire)

- HDL: Hardware Description Language
  - Figure out architecture first, then write code.
Port Connection

```
module FA1 (CO, S, A, B, CI);
    output CO, S;
    input A, B, CI;

    assign {CO, S} = A + B + CI;
endmodule
```

- Connect module port by order list
  - `FA1 fa1(c_o, sum, a, b, c_i);`

- Not fully connected
  - `FA1 fa3(c_o, a, b, c_i);`

- Connect module port by name `.PortName( NetName )`
  - `FA1 fa2(.A(a), .B(b), .CO(c_o), .CI(c_i), .S(sum));`
  - Recommended
Case Study
1-bit Full Adder

<table>
<thead>
<tr>
<th>Ci</th>
<th>A</th>
<th>B</th>
<th>Co</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\text{co} &= (a \cdot b) + (b \cdot \text{ci}) + (\text{ci} \cdot a);
\end{align*}
\]
Case Study
1-bit Full Adder

\[ \text{sum} = a \oplus b \oplus c_i \]

\[ \text{module } FA\_sum \ ( \text{sum, a, b, ci} ); \]
\[ \text{input } a, b, ci; \]
\[ \text{output } \text{sum, co}; \]
\[ \text{xor } g1( \text{sum, a, b, ci} ); \]
\[ \text{endmodule} \]
Case Study
1-bit Full Adder

Full Adder Connection
- Instance ins_c from FA_co
- Instance ins_s from FA_sum

```verilog
testbench
module FA_gatelevel( sum, co, a, b, ci );

input a, b, ci;
output sum, co;

FA_co ins_c( co, a, b, ci );
FA_sum ins_s( sum, a, b, ci );

endmodule testbench
```
Functional verification of hardware is used to verify functionality of the designed circuit.

However, blocks in real hardware have delays associated with the logic elements and paths in them.

To model these delay, we use timing / delay description in Verilog: #

Then we can check whether the total circuit meets the timing requirements, given delay specifications of the blocks.
Delay Specification in Primitives

Delay specification defines the propagation delay of that primitive gate.

\[
\text{not } \#10 (\text{out, in});
\]
Delay Specification in Primitives

- Verilog supports (rise, fall, turn-off) delay specification.
Delay Specification in Primitives

All delay specification in Verilog can be specified as 
(minimum : typical : maximum) delay

Examples

(min:typ:max) delay specification of all transition
ū or #(3.2:4.0:6.3) U0(out, in1, in2);

(min:typ:max) delay specification of RISE transition and
FALL transition
ū nand #(1.0:1.2:1.5,2.3:3.5:4.7) U1(out, in1, in2);

(min:typ:max) delay specification of RISE transition, FALL
transition, and turn-off transition
ū bufif1 #(2.5:3:3.4,2:3:3.5,5:7:8) U2(out, in, ctrl);
Types of Delay Models

- Distributed Delay
  - Specified on a per element basic
  - Delay value are assigned to individual in the circuit

```verilog
module and4(out, a, b, c, d);
  ...
  and #5 a1(e, a, b);
  and #7 a2(f, c, d);
  and #4 a3(out, e, f);
endmodule
```
Types of Delay Models

- Lumped Delay
  - They can be specified as a single delay on the output gate of the module.
  - The cumulative delay of all paths is lumped at one location.

```
module and4(out, a, b, c, d);
    ... ...
    and a1(e, a, b);
    and a2(f, c, d);
    and #11 a3(out, e, f);
endmodule
```
Types of Delay Models

- Pin-to-Pin Delay
  - Delays are assigned individually to paths from each input to each output.
  - Delays can be separately specified for each input/output path.

![Diagram]

Path a-e-out, delay = 9
Path b-e-out, delay = 9
Path c-f-out, delay = 11
Path d-f-out, delay = 11
Path Delay Modeling

- Specify blocks
  - Assign pin-to-pin timing delay across module path
  - Set up timing checks in the circuits
- Define \texttt{specparam} constants

```verilog
module and4(out, a, b, c, d);
    ...
// specify block with path delay statements
specify
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
endspecify

// gate instantiations
and    a1(e, a, b);
and    a2(f, c, d);
and    a3(out, e, f);
endmodule
```
Parallel/Full Connection

(a[0] => out[0]) = 9;
(a[1] => out[1]) = 9;
(a[2] => out[2]) = 9;
(a[3] => out[3]) = 9;

(a => out) = 9;

(b => out) = 9;
(c => out) = 11;
(d => out) = 11;

(a,b *=> out) = 9;
(c,d *=> out) = 11;
specparam Statement

- Special parameters can be declared for use inside a `specify` block.
- Instead of using hardcoded delay numbers to specify pin-to-pin delays

```verilog
module and4(out, a, b, c, d);
    ...
    // specify block with path delay statements
    specify
        specparam delay1 = 9;
        specparam delay2 = 11;
        (a,b *> out) = delay1;
        (c,d *> out) = delay2;
    endspecify
    ...
endmodule
```
Rise, Fall, and Turn-off Delays

Pin-to-pin timing can also be expressed in more detail by specifying rise, fall, and turn-off delay values.

// specify one delay statements
specparam t_delay = 9;
(clk => q) = t_delay;

// specify two delay statements
specparam t_rise = 9;
specparam t_fall = 13;
(clk => q) = (t_rise, t_fall);

// specify three delay statements
specparam t_rise = 9;
specparam t_fall = 13;
specparam t_turnoff = 11;
(clk => q) = (t_rise, t_fall, t_turnoff);

// specify six delay statements
specparam t_01= 9, t_10 = 13;
specparam t_0z = 11, t_z1 = 9;
specparam t_1z = 11, t_z0 = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);

// specify twelve delay statements
specparam t_01= 9, t_10 = 13;
specparam t_0z = 11, t_z1 = 9;
specparam t_1z = 11, t_z0 = 13;
specparam t_0x= 9, t_x1 = 13;
specparam t_1x = 11, t_x0 = 9;
specparam t_xz = 11, t_xz = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0, t_0x, t_x1, t_1x, t_x0, t_xz, t_xz);
Min, Max, and Typical Delays

- Min, max, and typical delay value were discussed earlier for gates.
- Can also be specified for pin-to-pin delays.

```vhdl
// specify two delay statements
specparam t_rise = 8:9:10;
specparam t_fall = 12:13:14;
specparam t_turnoff = 10:11:12
(clk => q) = (t_rise, t_fall, t_turnoff);
```
Timing Checks - For Testbench

✓ setup time and hold time checks

- setup time
- hold time

= clock
data

= specify
  $setup(data, posedge clock, 3);
  endspecify

= specify
  $hold(posedge clock, data, 5);
  endspecify

✓ Width check

- Sometimes it is necessary to check the width of a pulse.

= clock

= specify
  $width(posedge clock, 6);
  endspecify

= width of the pulse

p. 85

Foundamentals of HDL, 2011.3.2
Yu-Hao
Outline

✓ Overview and History
✓ Hierarchical Design Methodology
✓ Levels of Modeling
  ✓ Behavioral Level Modeling
  ✓ Register Transfer Level (RTL) Modeling
  ✓ Structural/Gate Level Modeling
✓ Language Elements
  ✓ Logic Gates
  ✓ Data Type
  ✓ Timing and Delay
✓ Simulation & Verification
Verification Methodology

- **Task**: systematically verify the functionality of a model.
- **Approaches**: Simulation and/or formal verification

**Simulation**:
1. detect syntax violations in source code
2. simulate behavior
3. monitor results
Components of a Simulation

The output results are verified by console/waveform viewer

The output results are verified by testbench or stimulus block
Verilog Simulator

Circuit Description

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

Testfixture

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

Verilog Simulator

Verilog Parser

Simulation Engine

User Interface

Graphical Simulation Result

Text Mode Simulation Result

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>16.00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>......</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testbench Template

Consider the following template as a guide for simple testbenches:

```vhdl
module t_DUTB_name ();  // substitute the name of the UUT(Unit Under Test)
reg ...;               // Declaration of register variables for primary inputs of the UUT
wire ...;             // Declaration of primary outputs of the UUT
parameter time_out = // Provide a value
UUT_name M1_instance_name ( UUT ports go here);

initial $monitor ( );  // Specification of signals to be monitored and displayed as text
initial #time_out $stop;  // (Also $finish) Stopwatch to assure termination of simulation
initial
begin
// Behavioral statements generating waveforms
// to the input ports, and comments documenting
// the test. Use the full repertoire of behavioral
// constructs for loops and conditionals.
end
endmodule
```

Foundamentals of HDL, 2011.3.2
Yu-Hao
p. 90
Example: Testbench

```verilog
module t_Add_half();

    wire sum, c_out;
    reg a, b; // Variable for stimulus waveforms

    Add_half_0_delay M1 (sum, c_out, a, b); //UUT

    initial begin
        // Time Out
        #100 $finish; // Stopwatch
    end

    initial begin
        // Stimulus patterns
        #10 a = 0; b = 0; // Statements execute in sequence
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
    end

endmodule
```
Behaviors for Abstract Models

- Verilog has three types of behaviors for composing abstract models of functionality:
  - Continuous assignment (Keyword: `assign`) -- later
  - Single pass behavior (Keyword: `initial`) -- Note: only use in testbenches
  - Cyclic behavior (Keyword: `always`) -- later

- Single pass and cyclic behaviors execute procedural statements like a programming language
- The procedural statements execute sequentially
- A single pass behavior expires after the last statement executes
- A cyclic behavior begins executing again after the last statement executes
Signal Generators

- Use cyclic behaviors to describe stimulus generators
- Statements in a behavior may be grouped in `begin` … `end` blocks
- Execution begins at $t_{sim} = 0$
- # delay control operator temporarily suspends execution of a behavior
- The operator `=` denotes procedural assignment (also called blocking assignment)

MODELING TIP

Use procedural assignments to describe stimulus patterns in a testbench.
Simulation Results

MODELING TIP

A Verilog simulator assigns an *initial* value of *x* to all variables.
Propagation Delay

Gate propagation delay specifies the time between an input change and the resulting output change.

Transport delay describes the time-of-flight of a signal transition.

Verilog uses an inertial delay model for gates and transport delay for nets.

Inertial delay suppresses short pulses (width less than the propagation delay value).

Inertial delay: the amount of time that the inputs must be stable in order to generate an output.

MODELING TIP

All primitives and nets have a default propagation delay of 0.
Example: Propagation Delay

Unit-delay simulation reveals the chain of events

```verilog
module Add_full (sum, c_out, a, b, c_in);
  output sum, c_out;
  input a, b, c_in;
  wire w1, w2, w3;

  Add_half M1 (w1, w2, a, b);
  Add_half M2 (sum, w3, w1, c_in);
  or #1 M3 (c_out, w2, w3);
endmodule

module Add_half (sum, c_out, a, b);
  output sum, c_out;
  input a, b;

  xor #1 M1 (sum, a, b);
  and #1 M2 (c_out, a, b);
endmodule
```
Compiler Directives

- **define**
  - `define RAM_SIZE 16
  - Defining a name and gives a constant value to it.
  - the identifier `RAM_SIZE will be replaced by 16

- **include**
  - `include adder.v
  - Including the entire contents of other verilog source file.
  - seldom use, replaced by specifying files to simulator in console

- **timescale**
  - `timescale 100ns/1ns
  - `timescale <reference_time_unit> / <time_precision>
  - Setting the reference time unit and time precision of your simulation.
System Tasks

- **Displaying information**
  - `$display(“ID of the port is %b”, port_id);`
  - ID of the port is 00101

- **Monitoring information**
  - `$monitor($time, “Value of signals clk = %b rst = %b”, clk, rst);`
  - 0 Value of signals clk = 0 rst = 1
  - 5 Value of signals clk = 1 rst = 1
  - 10 Value of signals clk = 0 rst = 0

- **Stopping and finishing in a simulation**
  - `$stop;` // provided to stop during a simulation
  - `$finish;` // terminates the simulator
Simulation Schemes

- There are 3 categories of simulation schemes
  - Time-based: Simulation on real time scale, used by SPICE simulators
  - Event-based: Simulation on events of signal transition, used by Verilog simulators. Note each event must occur at discrete time specified by the testbench.
  - Cycle-based: Used by system/platform level verification, less used in cell-based IC designing.
Time Wheel Concept in Event-Based Simulation

A current event scheduling another event