

Computer Organization

Lecture 11 - Verilog - Comb. and Structural Modeling

Reading: C.4, Verilog Handout

Assignment: Project 1

Today's Outline

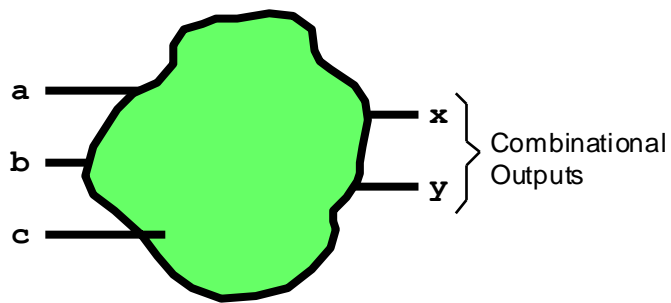
- ▶ **Verilog Language Basics**
- ▶ **Combinational Design with Verilog**
- ▶ **Structural Design - Module Instantiation**
- ▶ **Behavioral Simulation - Testbenches**
- ▶ **Discuss Project 1 - Adder/ALU Design**

Goals of HDL-Based Design

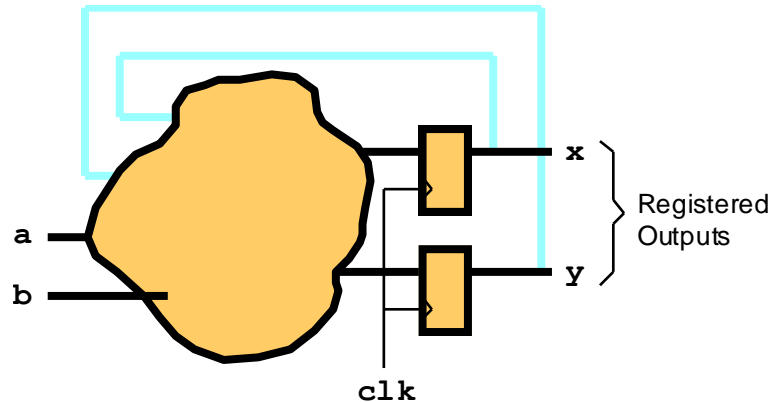
- ▶ Model **hardware** for
 - ▶ Simulation - predict how hardware will behave
 - ▶ Synthesis - generate optimized hardware for FPGA, ASIC
- ▶ Provide a concise text description of circuits
- ▶ Support design of very large systems

Important Points about Verilog

- ▶ **Key point:** Verilog code describes **hardware**
- ▶ Verilog code is ***not like software!***
 - ▶ Hardware is parallel
 - ▶ Software is sequential
- ▶ When writing Verilog code, think of **hardware**
- ▶ Follow **coding guidelines** to avoid pitfalls



Combinational Logic



Registered Logic

Verilog Language Basics

- ▶ **Syntax - See Quick Reference Card**
- ▶ **Major elements of language:**
 - ▶ **Lexical Elements (“tokens” and “token separators”)**
 - Numbers
 - Identifiers
 - Operators
 - ▶ **Data Types and Values**
 - ▶ **Modules**

Lexical Elements

- ▶ **Whitespace** - ignored except as token separators
- ▶ **Comments**
 - ▶ Single-line comments `//`
 - ▶ Multi-line comments `/* ... */`
- ▶ **Numbers** (special syntax to specify bit-width)
- ▶ **Identifiers** (e.g. module names, port names, etc.)
- ▶ **Operators**- unary, binary, ternary
 - ▶ Unary `a = ~b;`
 - ▶ Binary `a = b && c;`
 - ▶ Ternary `a = (b < c) ? b : c;`

Numbers

- ▶ Sized numbers - with specific bitwidth and radix

16' d255

Size in bits

Radix:

d or D - decimal

H or h - hexadecimal

O or o - octal

b or B - binary

Value in consecutive digits:

0 - 9

a - f (for hexadecimal)

x - unknown

z - high impedance

4' b1101

12' h7af

12' b0111_1010_1111

Spacers

Numbers (continued)

- ▶ **Unsigned numbers (decimal, usually 32 bits)**

0	3547	65535
---	------	-------

- ▶ **Negative numbers**

-3547	-12' h3eb	-16' d1
-------	-----------	---------

Strings

- ▶ Character sequence in quotes (on single line)
- ▶ Treated as a sequence of 1-byte ASCII values
- ▶ Special characters - C-like (e.g., \n)

`"This is a string"` `"a/b"` `"a\nb"`

Identifiers

- ▶ **Standard C-like identifiers**
 - ▶ First character alphabetic or ‘_’
 - ▶ Following characters: alpha, numeric, or ‘_’
 - ▶ Identifiers are case sensitive
 - ▶ Identifiers can’t be reserved words
- ▶ **"Escaped" identifiers:**
 - ▶ start with backslash
 - ▶ follow with any non-whitespace ASCII
 - ▶ end with whitespace character
- ▶ **Examples**

John paul _0george **r1ng0

Reserved Words



always	and	assign	begin	buf	bufif0	bufif1	case
casex	casez	cmos	deassign	default	defparam	disable	edge
else	end	endcase	endfunction		endmodule		
endprimitive		endspecify		endtable	endtask	event	for
force	forever	fork	function	highz0	highz1	if	ifnone
initial	inout	input	integer	join	large	macromodule	
medium	module	nand	negedge	nmos	nor	not	
notif0	notif	or	output	parameter	pmos		
posedge	primitive	pull0	pull1	pulldown	pullup	rcmos	
real	realtime	reg	release	repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small	specify	specparam	strong0	
strong1	supply0	supply1	table	task	time	tran	tranif0
tranif1	tri	tri0	tri1	triand	trior	triereg	vectored
wait	wand	weak0	weak1	while	wire	wor	xnor
xor							

Data Types

- ▶ **nets** - describe “wire” connections

- ▶ general purpose: **wire**
- ▶ special purpose: `supply0, supply1,`
`tri0, tri1, triand, trior,`
`triereg, wand, wor`

- ▶ **registers** - variables

(assigned values by procedural statement)

- ▶ **reg** - basic binary values
- ▶ **integer** - binary word (≥ 32 bits - machine dependent)
- ▶ **real** - floating point (not supported by synthesis)
- ▶ **time** - simulation time (not supported in synthesis)
- ▶ **realtime** - simulation time (not supported in synthesis)

More about Data Types

- ▶ **Vectors - Multiple-Bit Signals (net or register)**

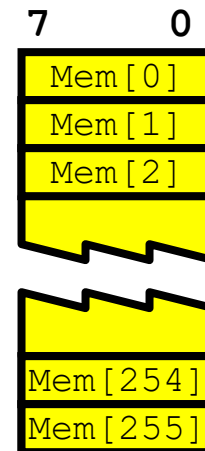
```
wire [31:0] sum;  
reg  [7:0]  avg;
```

- ▶ **Arrays - used for memories**

```
reg  [7:0] mem [0:255];
```

word size

memory size

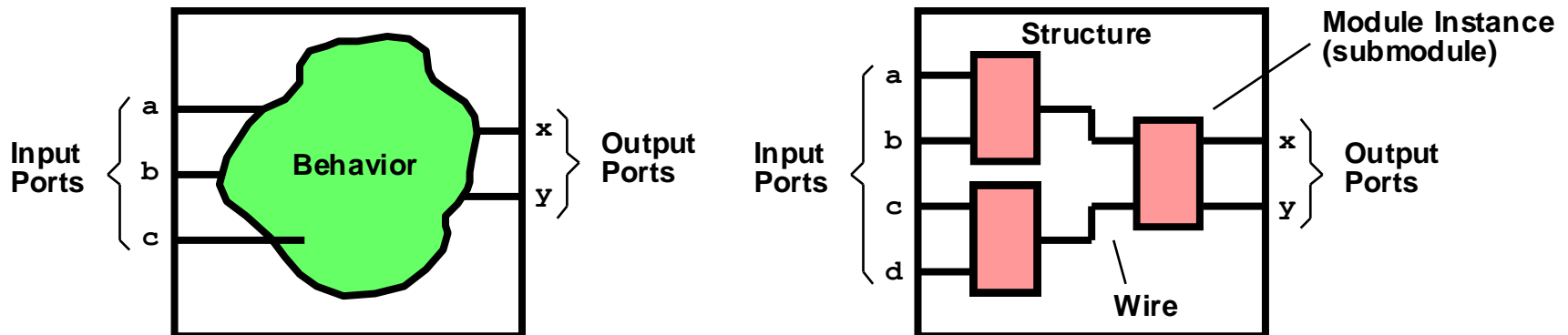


Logic Values

- ▶ **Each bit can take on 4 values:**
 - ▶ **0** - Standard binary “FALSE”
 - ▶ **1** - Standard binary “TRUE”
 - ▶ **x** - UNKNOWN
 - ▶ **z** - High Impedance
- ▶ **During simulation, all values are originally x**
- ▶ **Complication: x & z sometimes used as “wildcards” (e.g. casex, casez)**

Verilog Modules

- ▶ **module** - key building block of language
 - ▶ **Interface** - input and output ports
 - ▶ **Body** - specifies contents of "black box"
 - behavior - what it does
 - structure - how it's built from other "black boxes"



Syntax: See Quick Reference Card Section 1

Module Declaration

```
module modulename ( port1, port2, ... );  
    port1 direction declaration;  
    port2 direction declaration;  
    reg declarations;  
    wire declarations;  
  
    module body - "parallel" statements:  
        assign, always, initial, etc.  
  
endmodule
```


Module Example

```
module full_adder(a, b, cin, sum, cout);  
    input a,  $\bar{b}$ , cin;  
    output sum, cout;  
  
    assign sum = a ^ b ^ cin;  
    assign cout = a & b | a & cin | b & cin;  
endmodule
```

Ports

} Port Declarations

Port direction:

input

output

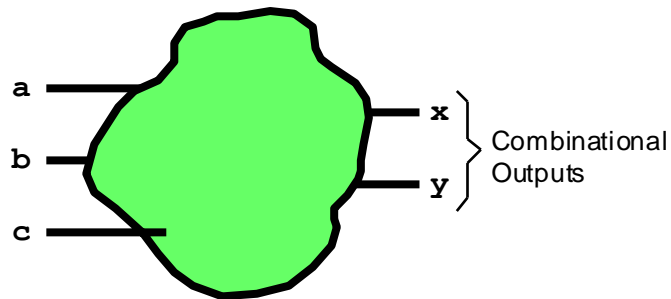
inout

Module Body - “Parallel” Statements

- ▶ Describe **parallel** behavior - **unlike** C/Java!
- ▶ Types of Parallel Statements:
 - ▶ **assign** - simple combinational logic with an expression
 - ▶ **always** - complex behavior for combinational or registered logic with **procedural statements**
 - ▶ module **instantiation** - used for structure
 - ▶ **initial** - used to specify startup behavior
(not supported in synthesis - but **useful in simulation!**)
 - ▶ ... and other features useful only in simulation

Combinational Logic

- ▶ Describe using Verilog “Parallel Statements”
 - ▶ Simple Expression - `assign`
 - ▶ Procedural Code - `always`



```
wire x, y;  
assign x = (a | b) & c;  
assign y = (a & b) | c;
```

```
reg x, y;  
always @(a or b or c)  
begin  
    x = (a | b) & c;  
    y = (a & b) | c;  
end
```

Continuous Assignment

- ▶ **General Form:**

assign *wire* = *expression*

- ▶ **Example:**

```
module full_adder(a, b, cin, sum, cout);  
    input a, b, cin;  
    output sum, cout;  
  
    assign sum = a ^ b ^ cin;  
    assign cout = a & b | a & cin | b & cin;  
endmodule
```

Verilog Operators

▶ Equivalent to C/C++/Java Operators

- ▶ Arithmetic: + - * / &
- ▶ Comparison: == != < <= > >=
- ▶ Shifting: << >>

▶ Example:

```
module adder(a, b, y) ;  
    input  [31:0]    a, b;  
    output [31:0]    y;  
  
    assign y = a + b;  
endmodule
```

Operators and Precedence

- ▶ Same precedence as Java/C/C++ (plus a few extras)
- ▶ Override with parentheses () when needed

~	Highest
*, /, %	
+, -	
<<, >>	
<, <=, >, >=	
=, ==, !=	
&, ~&	
^, ~^	
, ~	
?:	Lowest

Bitwise Operators in Expressions

- ▶ Basic bitwise operators: identical to C/C++/Java

```
module inv(a, y);  
  input  [3:0] a;  
  output [3:0] y; } 4-bit Ports
```

```
  assign y = ~a;  
endmodule
```

↑
Unary Operator: NOT

Reduction Operators in Expressions

- ▶ Apply a single logic function to multiple-bit inputs

```
module and8(a, y);  
    input  [7:0] a;  
    output          y;
```

```
    assign y = &a;  
endmodule
```



Reduction Operator: AND & (also supports OR |, EXOR ^)

equivalent to:

`a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[2] & a[2] & a[0]`

Conditional Operators in Expressions

▶ Like C/C++/Java Conditional Operator

```
module mux2(d0, d1, s, y);  
    input  [3:0] d0, d1;  
    input          s;  
    output [3:0] y;  
  
    assign y = s ? d1 : d0;  
    // output d1 when s=1, else d0  
endmodule
```

▶ “if” statements **not allowed** in assign

Concatenation

- ▶ `{ }` combines bit vectors

```
module cat(input [7:0] a, input [7:0] b,  
           output [15:0] w);
```

```
    assign w = {a, b};  
endmodule
```

Concatenates 8 bits + 8 bits => 16 bits

```
module adder(input [31:0] a, input [31:0] b,  
             output [31:0] y, output cout);
```

```
    assign {cout, y} = a + b;  
endmodule
```



Concatenates 1 bits + 32 bits => 33 bits

32-bit add produces 33-bit result

More Operators: Replication

- ▶ `{ n {pattern} }` replicates a pattern *n* times

```
module sign_extend(a, y);  
    input  [15:0]    a;  
    output [31:0]    y;  
  
    assign y = {16{a[15]}, a[15:0]};  
endmodule
```

 **Copies sign bit 16 times**  **Lower 16 Bits**

Internal Signals

► Declared using the **wire** keyword

```
module full_adder(a, b, cin, s, cout);  
    input      a, b, cin;  
    output s, cout;
```

```
    wire      prop;
```

```
    {  
        assign prop = a ^ b;  
        assign s = prop ^ cin;  
        assign cout = (a & b) | (cin & (a | b));  
    }  
endmodule
```

→ Important point: these statements “execute” in parallel

Combinational always blocks

► Motivation

- assign statements are fine for simple functions
- More complex functions require procedural modeling

► Basic syntax:

`always (`*sensitivity-list*`)`



Signal list - change activates block

statement ← Sequential statement (=, if/else, etc.)

or

`always (`*sensitivity-list*`)`

`begin`

statement-sequence

`end`

} Compound Statement -
sequence of statements
(which execute sequentially)

Combinational Modeling with always

► Example: 4-input mux behavioral model

```
module mux4(d0, d1, d2, d3, s, y);  
    input [31:0] d0, d1, d2, d3;  
    input [1:0] s;  
    output [31:0] y;  
    reg [31:0] y; // declare y as a variable
```

Sensitivity List (activates on change)

```
always @ (d0 or d1 or d2 or d3 or s)
```

```
    case (s)
```

```
        2'd0 : y = d0;
```

```
        2'd1 : y = d1;
```

```
        2'd2 : y = d2;
```

```
        2'd3 : y = d3;
```

```
        default : y = 1'bx;
```

```
    endcase
```

```
endmodule
```

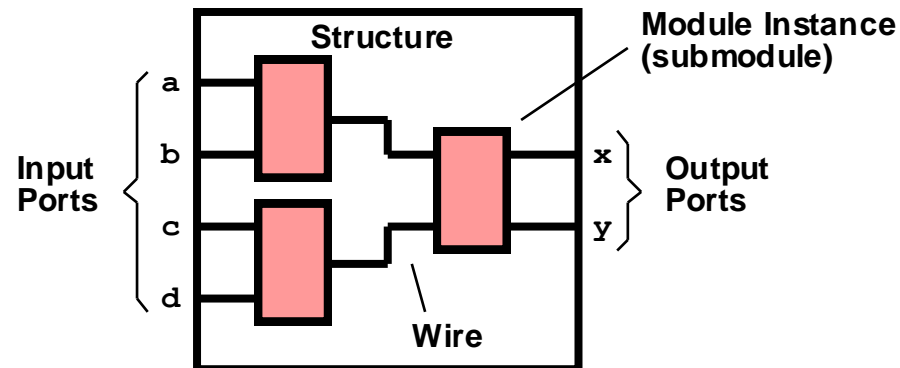
Blocking assignments
(immediate update)

Guideline: Assign outputs (in comb. logic) using blocking assignment.

Modeling with Hierarchy

- ▶ Create instances of submodules
- ▶ Example: Create a 4-input Mux using mux2 module
- ▶ Original mux2 module:

```
module mux2(d0, d1, s, y);  
    input  [3:0] d0, d1;  
    input          s;  
    output [3:0] y;  
    assign y = s ? d1 : d0;  
endmodule
```



Module Instantiation (Positional)

- ▶ Create instances of submodules
- ▶ Example: Create a 4-input Mux using mux2 module

```
module mux4(d0, d1, d2, d3, s, y);  
    input  [3:0] d0, d1, d2, d3;  
    input  [1:0] s;  
    output [3:0] y;  
  
    wire    [3:0] low, high;  
  
    mux2 U_LOW(d0, d1, s[0], low);  
    mux2 U_HIGH(d2, d3, s[0], high);  
    mux2 U_FINAL(low, high, s[1], y);  
endmodule
```

↑
Instance Names

↑
Connections (Positional)

Module Instantiation (Explicit)

► Preferable for long port lists

```
module mux4 (d0, d1, d2, d3, s, y);  
    input    [3:0]  d0, d1, d2, d3;  
    input    [1:0]  s;  
    output   [3:0]  y;  
  
    wire     [3:0]  low, high;  
  
    mux2 U_LOW(.d0(d0), .d1(d1), .s(s[0]), .y(low));  
    mux2 U_HIGH(.d0(d2), .d1(d3), .s(s[0]), .y(high));  
    mux2 U_FINAL(.d0(low), .d1(high), .s(s[1]), .y(y));  
endmodule
```

Guideline: Use the explicit connection style when instantiating modules with more than 3 ports.

Larger Example: Ripple Adder

► Recall Full Adder Example

```
module full_adder(a, b, cin, sum, cout);  
    input a,  $\bar{b}$ , cin;  
    output sum, cout;  
  
    assign sum = a ^ b ^ cin;  
    assign cout = a & b | a & cin | b & cin;  
endmodule
```

Example: 8-Bit Ripple Adder (Positional)

```
module add8(a, b, sum, cout);
    input [7:0] a, b;
    output [7:0] sum;
    output cout;

    wire [7:0] c; // used for carry connections

    assign c[0]=0;
    full_adder f0(a[0], b[0], c[0], sum[0], c[1]);
    full_adder f1(a[1], b[1], c[1], sum[1], c[2]);
    full_adder f2(a[2], b[2], c[2], sum[2], c[3]);
    full_adder f3(a[3], b[3], c[3], sum[3], c[4]);
    full_adder f4(a[4], b[4], c[4], sum[4], c[5]);
    full_adder f5(a[5], b[5], c[5], sum[5], c[6]);
    full_adder f6(a[6], b[6], c[6], sum[6], c[7]);
    full_adder f7(a[7], b[7], c[7], sum[7], cout);
endmodule
```

Example: 8-Bit Ripple Adder (Explicit)

```
module add8(a, b, sum, cout);
    input [7:0] a, b;
    output [7:0] sum;
    output cout;

    wire [7:0] c; // used for carry connections

    assign c[0]=0;
    full_adder f0(.a(a[0]), .b(b[0]), .cin(c[0]),
                  .sum(sum[0]), .cout(c[1]));
    full_adder f1(.a(a[1]), .b(b[1]), .cin[c1]),
                  .sum(sum[1]), .cout[c2]);
    full_adder f2(.a(a[2]), .b(b[2]), .cin[c2]),
                  .sum(sum[2]), .cout[c3]);
    . . .

    full_adder f6(.a(a[6]), .b(b[6]), .cin[c6]),
                  .sum(sum[6]), .cout[c7]);
    full_adder f7(.a(a[7]), .b(b[7]), .cin[c7]),
                  .sum(sum[7]), .cout(cout));
endmodule
```

Hierarchical Design with Gate Primitives

- ▶ “Built-In” standard logic gates

`and or not xor nand nor xnor`

- ▶ Using Gate Primitives:

```
and g1(y, a, b, c, d);
```



Output

Inputs (variable number)

- ▶ How are the different from operators (&, |, ~, etc.)?
 - ▶ Operators specify function
 - ▶ Gate primitives specify structure

Gate Primitives Example

▶ 2-1 Multiplexer

```
module mux2s(d0, d1, s, y);  
    wire sbar, y0, y1;  
    not inv1(sbar, s);  
    and and1(y0, d0, sbar);  
    and and2(y1, d1, s);  
    or or1(y, y0, y1);  
endmodule;
```

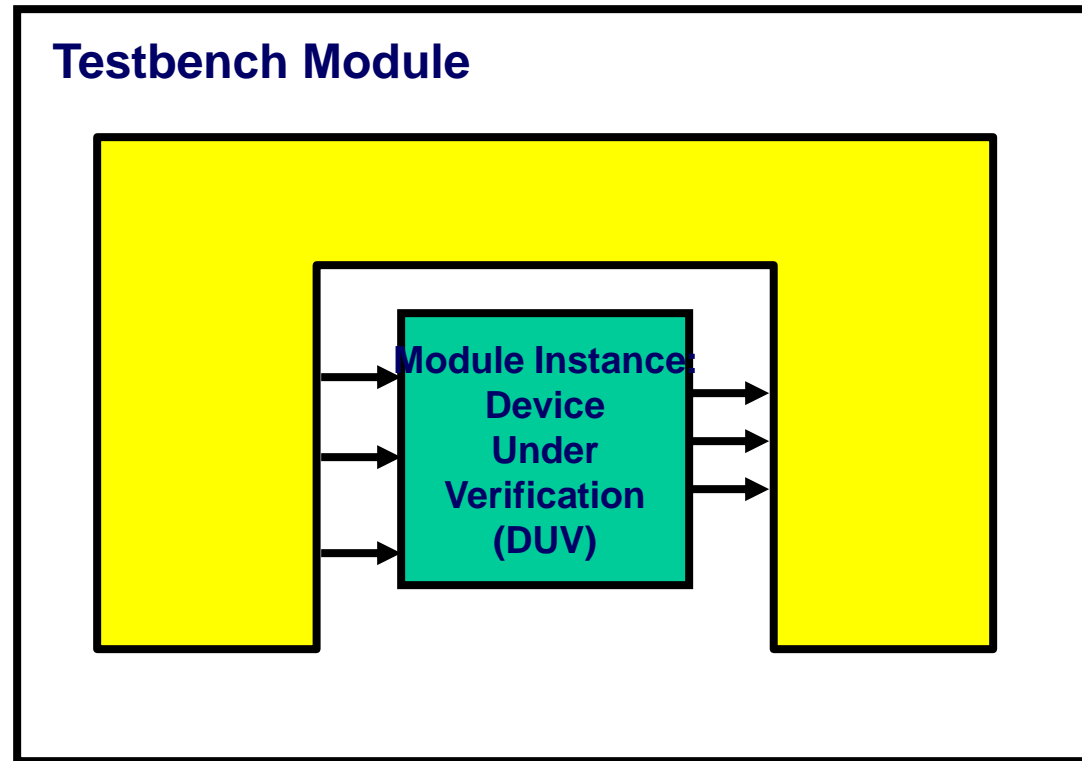
▶ Why shouldn't we use gate primitives?

- ▶ Requires “low-level” implementation decisions
- ▶ It's often better to let synthesis tools make these

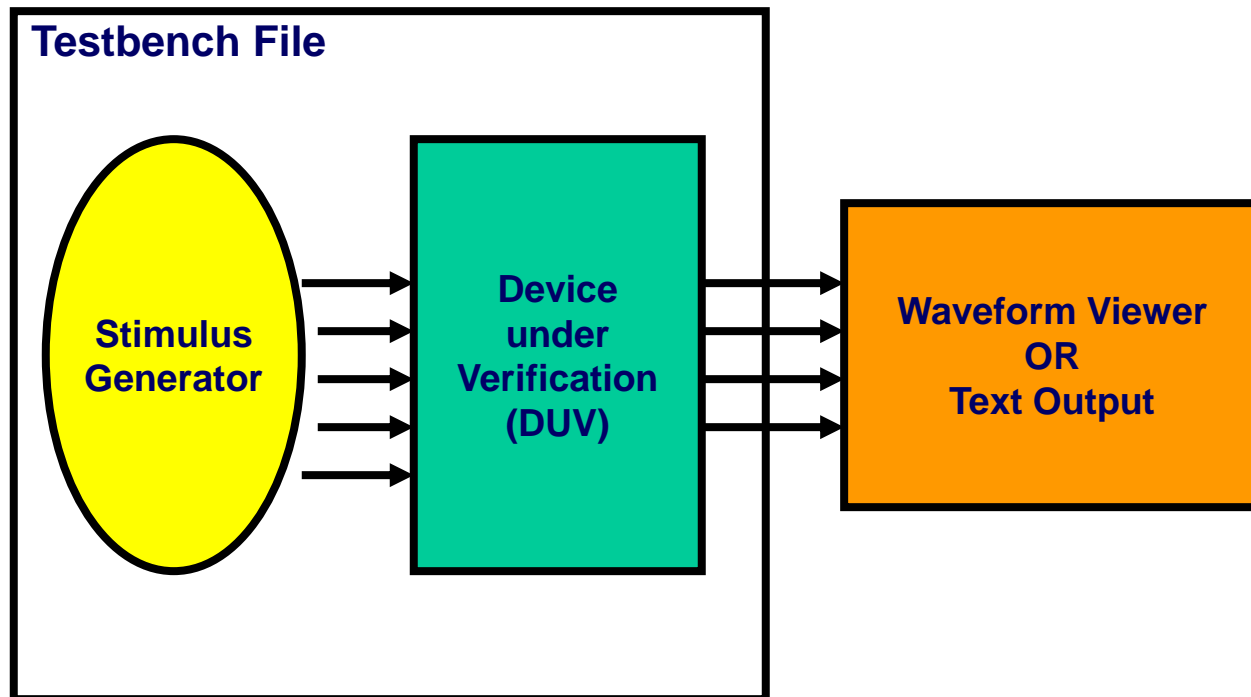
Testbenches

- ▶ A **testbench** (test fixture) is HDL code to **verify** a module
 - ▶ Apply input vectors to module inputs
 - ▶ Check module outputs
 - ▶ Report errors to user
- ▶ Why use a testbench instead of a GUI timing diagram?
 - ▶ Portability - testbench will work on any HDL simulator
 - ▶ Automatic checking - don't have to interpret waveform
 - ▶ Expressability - can use the full semantics of HDL to:
 - generate input vectors (possibly from input file)
 - **check** output vectors
 - control simulation

Coding Testbenches in Verilog HDL



Testbench Approaches - Stimulus-Only



Building a Testbench for Simulation

- ▶ **Use the New Source “Test Fixture” wizard in ISE**
- ▶ **Edit the resulting code to add input stimulus**
- ▶ **Run the simulator and observe the output**
- ▶ **Demo – Using ISE/ISIM**

Example: Ripple Adder Testbench

```
module ripple_bench;

    // Inputs
    reg [7:0] a;
    reg [7:0] b;

    // Outputs
    wire [7:0] sum;
    wire cout;

    // Instantiate the UUT
    ripple_adder uut (
        .a(a),
        .b(b),
        .sum(sum),
        .cout(cout)
    );

    initial begin
        // Initialize Inputs
        a = 0;
        b = 0;

        // Wait 100 ns for glob...
        #100;

        // Add stimulus here
        a = 8'h5;
        b = 8'h6;
        #100;

        a = 8'hFE;
        b = 8'hA;
        #100;
        $stop;
    end
endmodule
```

Assignment - Project 1

► Modeling and Simulating Arithmetic Circuits

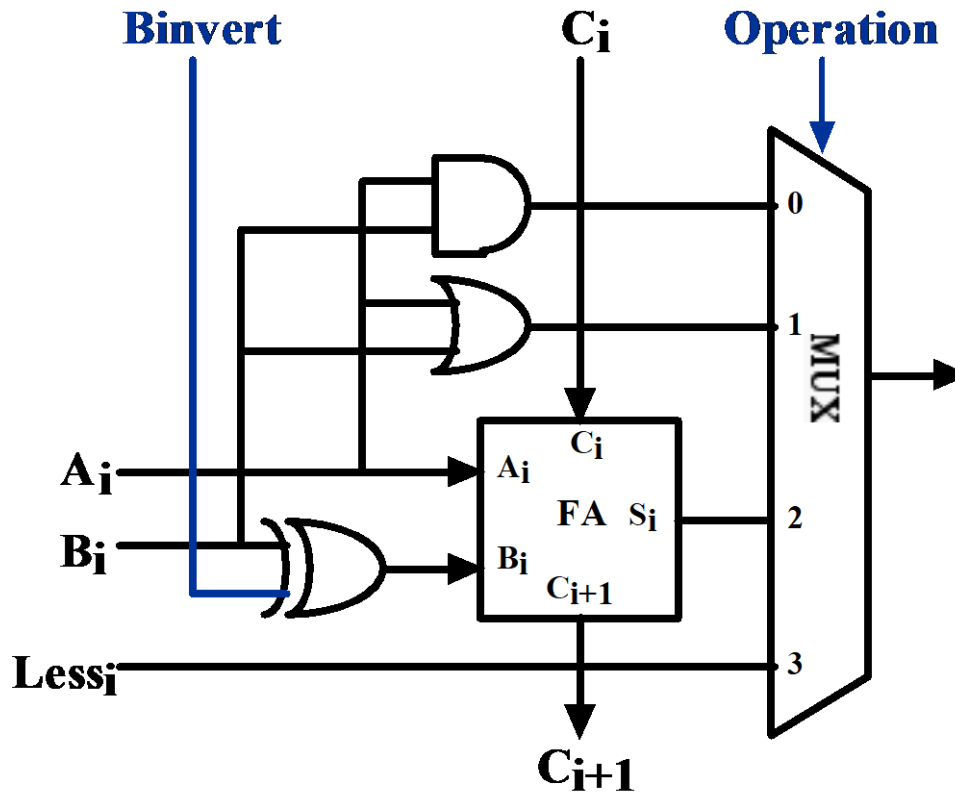
► Part 1

- Basic Ripple Adder - expand 8 bit to 16 bit
- Simulate with several different values to exercise fully including carry function!
- Hand in listings, annotated simulation timing diagrams

► Part 2

- Basic ALU Slice
- **16-bit** ALU (note **no** carry in)
- Simulate with several different values to exercise fully
- Hand in listings, annotated simulation timing diagrams

Review - ALU Slice





Coming Up

- ▶ **Floating Point**
- ▶ **Sequential Logic in Verilog**