Section 2

Sample Design

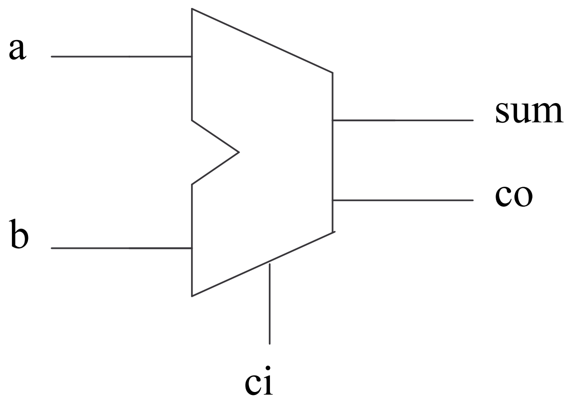
The Verilog Module

- Modules are the basic building blocks in Verilog.
- Every module starts with the keyword ***module***, a unique name, and ends with ***endmodule***.
- The descriptions of the logic being modeled are placed within the module.
- Modules can represent anything from a low level logic block to a high level system.
- Sample Design: 4-bit adder

- **module** *name* (ports);
- // port declarations
- // data type declarations
- // functionality, instantiations, timing, etc.
- **endmodule**

Every level of hierarchy uses the ***module/endmodule*** statements, no matter what the functionality is inside. The ***module*** is the building block for all designs in Verilog.

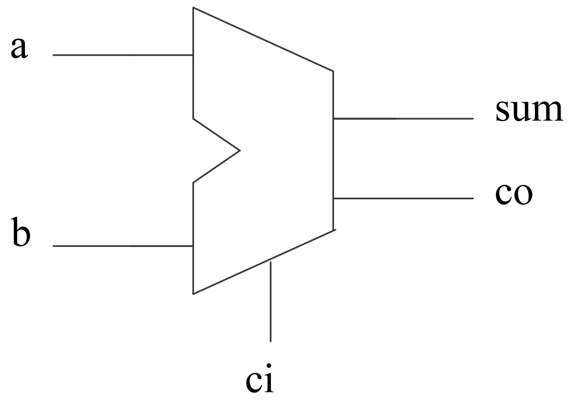
1 Bit Adder



A	B	CI	CO	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Here's the functionality of a 1-bit adder. Now let's design it in Verilog.

1 Bit Adder Module



- Module definition and port list.

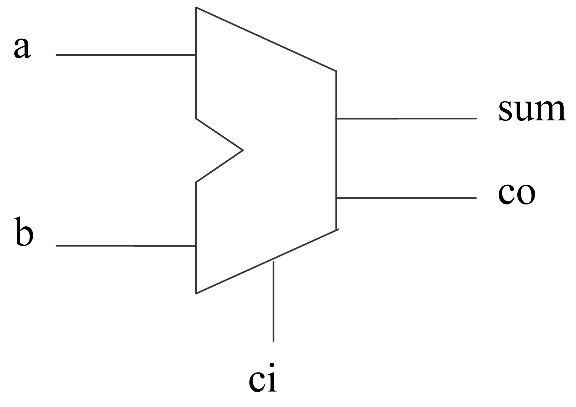
```
module addbit (a, b, ci, sum, co);
   // port declarations
   // data type declarations
   // functionality,
endmodule
```

module is a keyword in Verilog

addbit is a user-defined name (identifier), we'll discuss the rules for identifiers a little later in the course.

The port list (a, b, ci, sum, co) can be in any order. Also, the port names follow the same "identifier" rules.

1 Bit Adder Port Declarations



- Port declarations (any order)

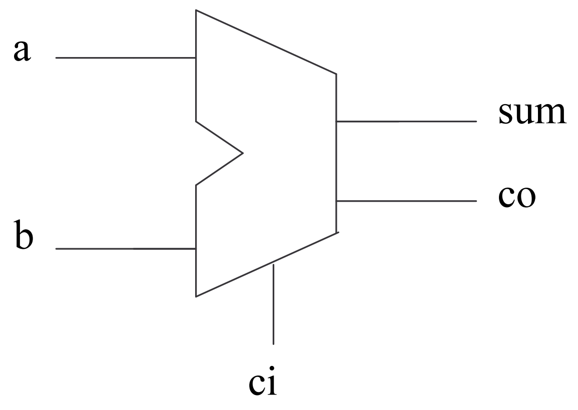
```
module addbit (a, b, ci, sum, co);
   output sum, co;
   input a, b, ci;
   // data type declarations
   // functionality,
endmodule
```

The port type declaration (input, output, or inout) can be in any order. Also, the ports can be in any order within the declaration. For example, the following would be legal port declarations:

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

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

1 Bit Adder Data Type Declarations



- Data type declarations (any order)

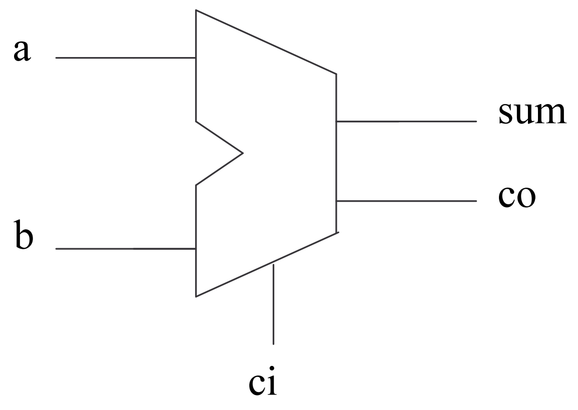
```
module addbit (a, b, ci, sum, co);
   output sum, co;
   input a, b, ci;
   wire a, b, ci;
   reg sum, co;
   // functionality,
endmodule
```

Data type declarations can be an in any order. As a general rule, they follow the port declarations and are declared before they're used.

The ***wire*** data type is the default in Verilog. I have added here to show that we're declaring all data types. However, if I did not include the line: ***wire*** *a, b, ci;* Verilog would still default the data type for a, b, and ci to a 1-bit ***wire***. Any signal/input that is wider than 1-bit has to be explicitly declared. For example, if 'a' is a 4-bit wide bus, it would have to be declared as such.

The ***reg*** data type declaration is required for any signal/output that is the LHS of an assignment in a procedural block. There will be much more on this topic later. For now just remember that if a signal/output is on the LHS of an assignment in a procedure block, you have to declare it as a ***reg*** first.

1 Bit Adder Functionality



- Functionality

```
module addbit (a, b, ci, sum, co);
  output sum, co;
  input a, b, ci;
  wire a, b, ci;
  reg sum, co;
  always @(a or b or ci) begin
    {co, sum} = a + b + ci;
  end
endmodule
```

-Note: '{}' means concatenation

The concatenation construct "{ }", saves steps in making the assignment to co and sum. Without it, you would have to declare in intermediate 2-bit variable, and make individual assignments:

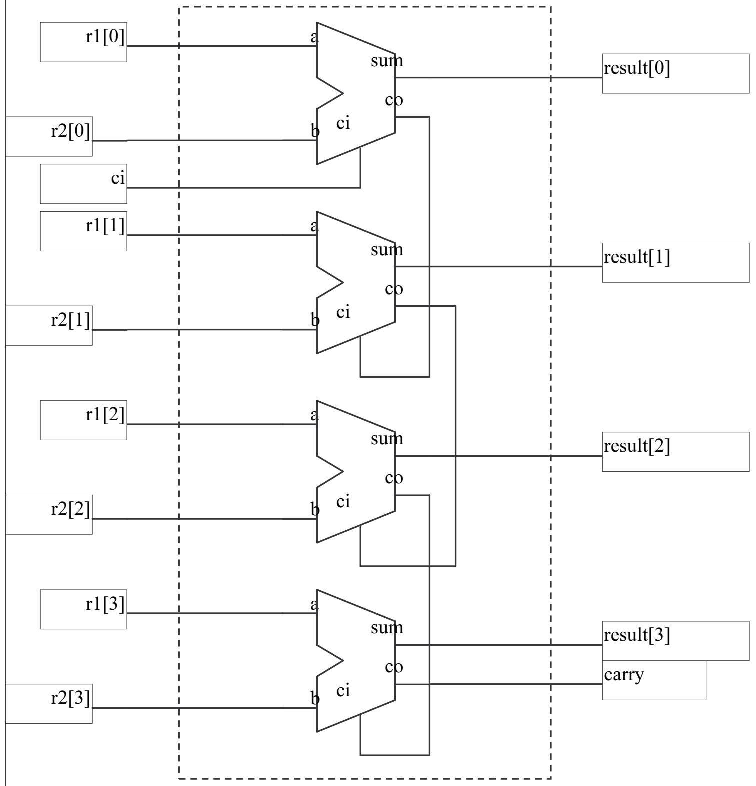
```
module addbit (a, b, ci, sum, co);
  output sum, co;
  input a, b, ci;

  wire a, b, ci;
  reg sum, co;
  reg [1:0] temp;

  always @(a or b or ci) begin
    temp = a + b + ci;
    co = temp[1];
    sum = temp[0];
  end

endmodule
```

4 Bit Adder Module Instantiation



- Instantiate 4 1-bit adders to create a 4-bit adder

```
module add4bits (result, carry, r1, r2, ci);
  input [3:0] r1, r2;
  input ci;
  output [3:0] result;
  output carry;
  wire [3:0] r1, r2, result;
  wire ci, carry, c1, c2, c3;
     addbit u1 (r1[0], r2[0], ci, result[0], c1);
     addbit u2 (r1[1], r2[1], c1, result[1], c2);
     addbit u3 (r1[2], r2[2], c2, result[2], c3);
     addbit u4 (r1[3], r2[3], c3, result[3], carry);
endmodule
```

Notice that each instance of *addbit* has a unique “instance name” such as u1, u2, etc. This is a requirement in Verilog; whenever you instantiate a lower level module you must use an “instance name” to define each instance. The instance name follows the rules of Verilog identifiers.

Notice that *r1, r2,* and result are 4-bit wide busses. The “[ ]” nomenclature signifies each individual bit of the bus.

Named vs. Positional Port Connections

- There are two methods for making connections to modules during instantiation:
  - Ordered List: The signals to be connected must appear in the module instantiation in the **same order** as the port order in the module's definition.
  - Port Named List: The signals to be connected are specified using the **port name** in the module definition. Order is irrelevant using this method.

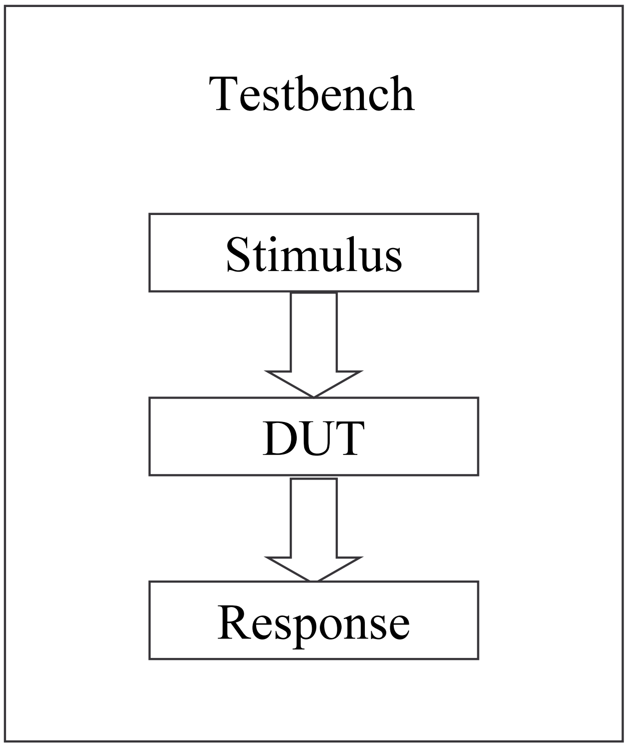
```
Given:      module aa21 (Q, A, B);

Instances:  aa21 u1 (my_q, my_a, my_b);                    // ordered list
            aa21 u2 (.A(my_a), .B(my_b), .Q(my_q));        // port named list
```

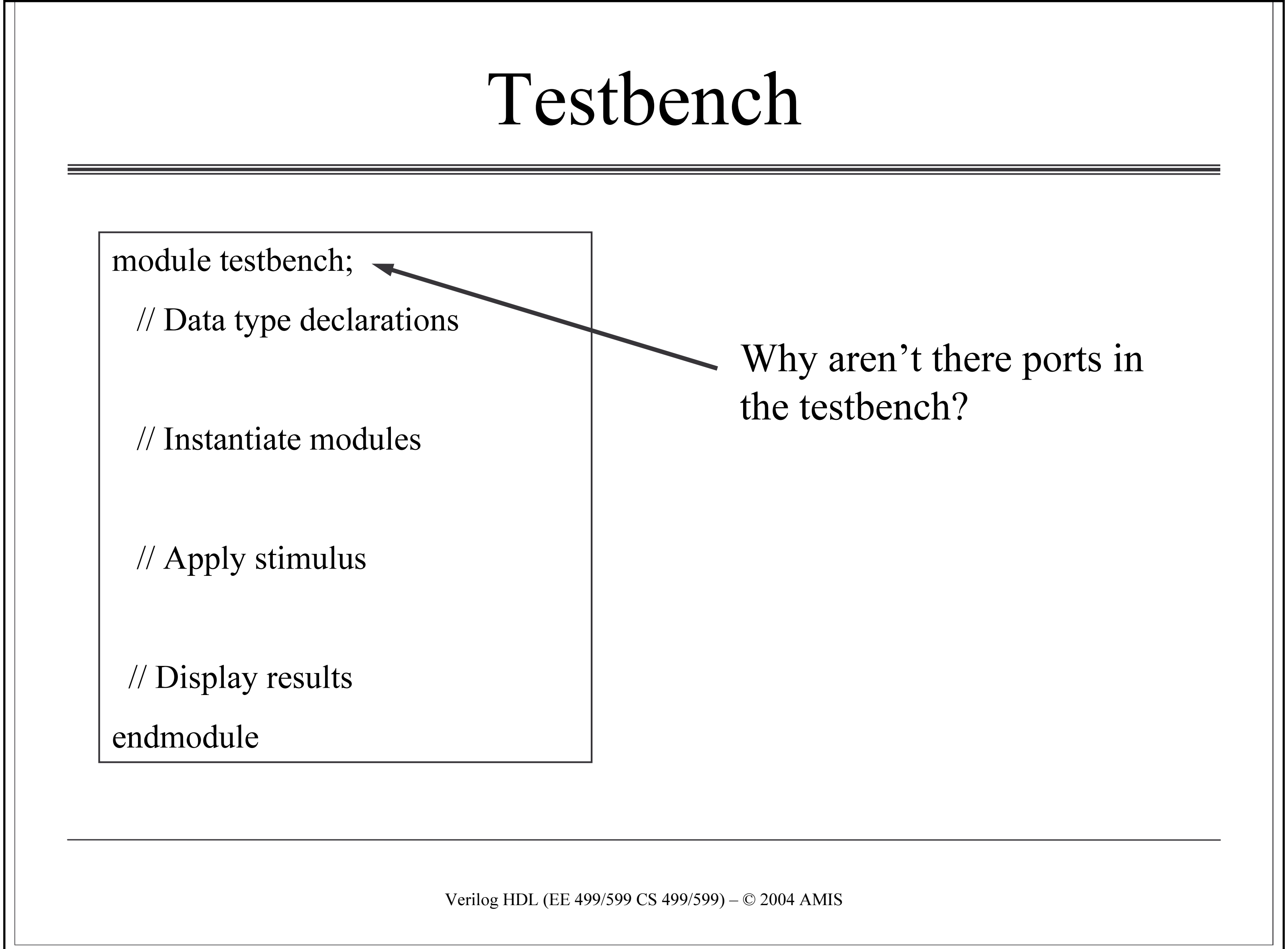
The fastest way to instantiate a lower level module is to use the *Ordered List* method. Sometimes, the fastest isn't always the safest way. The safest way to ensure that all ports are connected and connected correctly is to use the *Port Named List* method.

Simulating in Verilog

- Required components
  - Stimulus
  - Device Under Test (DUT)
  - Response Generation and Verification



Our DUT in this case is the 4-bit Adder that we've built from 1-bit Adders.



Testbench

```
module testbench;
   // Data type declarations


   // Instantiate modules


   // Apply stimulus


  // Display results
endmodule
```

Why aren’t there ports in the testbench?

The testbench is always the top level ***module*** in your design. In a sense, it surrounds your design. The testbench provides stimulus to the design and receiving feedback from the design. In this example we’ll show how a simple testbench works.

The name “testbench” is a user defined name. The name you use for your testbench can be any legal identifier.

Testbench

- Data type declarations
- Instantiate module(s)

```
• module testbench;
•     // Data type declaration
•     // Instantiate modules (DUT)
•     // Apply stimulus
•     // Display results
• endmodule
```

```
module testbench;
reg [3:0] r1, r2;
reg ci;
wire [3:0] result;
add4bits u1 (result, carry, r1, r2, ci);
   // Apply stimulus
   // Display results
endmodule
```

Why isn’t ‘carry’ declared in the data type declarations?

Again, the default data type for Verilog is a 1-bit wire. A wire of 1-bit does not need to be declared. Any other data type, or any other vectored wire must be declared. Notice that we have to declare *result* because it is wider than 1-bit.

Also, notice that we again have used an “instance name” for our instantiation of *add4bits*.

Testbench

- The pound sign (#) character denotes the delay specification for both gate instances and procedural statements.
- The *initial* block is a (behavioral) "procedural block" that executes only once.
- The dollar sign ($) denotes Verilog system tasks or PLI routines.
- $finish causes the simulation to finish and exit.

- Apply stimulus

```
initial
  begin
    r1 = 0; r2 = 0; ci = 0;
    #10  r1 = 5;
    #10  r2 = 10;
    #10  ci = 1;
    #10  r1 = 8;
    #10  ci = 0;
    #10 $finish;
  end
```

The ***initial*** block starts executing at simulation time zero and steps through the stimulus based on the timing control in the procedural block. The simulation will finish at 60 time units. We'll discuss how to give the time units some meaning (i.e. nanoseconds, picoseconds, etc.) in the next few slides.

Procedural Blocks

Executes once

```
initial
  begin
    r1 = 0; r2 = 0; ci = 0;
    #10  r1 = 5;
    #10  r2 = 10;
    #10  ci = 1;
    #10  r1 = 8;
    #10  ci = 0;
    #10 $finish;
  end
```

Executes at every rising clock edge

```
always @(posedge clock)
  begin
    r1 <= 1;
   r2 <= 0;
   ci <= 1;
  end
```

initial and ***always*** are the ONLY procedural blocks in Verilog.

Testbench (cont.)

- module testbench;
- // Data type declaration
- // Instantiate modules
- // Apply stimulus
- // Display results
- endmodule

```
module testbench;
. . .
add4bits u1 (result, carry, r1, r2, ci);
initial begin
  r1 = 0; r2 = 0; ci = 0;
  #10  r1 = 5;
  #10  r2 = 10;
  #10  ci = 1;
  #10  r1 = 8;
  #10  ci = 0;
  #10 $finish;
end
endmodule
```

“. . .” means that something irrelevant to the example has been left out. It is not legal Verilog syntax.

Display Simulation Results to Screen

```
initial begin
  $timeformat(-9,2,“ ns”, 10);
  $monitor(“time=%t  r1=%b  r2=%b  ci=%b  result=%b  carry=%b”, $realtime, r1, r2, ci, result, carry);
end
```

- The $timeformat system task globally specifies how time values will be displayed using the %t formatter. (<units>, <precision>, <suffix>, <min_width>)
- The $monitor system task displays the values of the argument list whenever any of the arguments change (except $time, $stime, $realtime)
- The $realtime system task returns the current simulation time as a real number.

NC-Verilog and VerilogXL requires the `timescale compiler directive to be used whenever the $timeformat command is used. This will be discussed later in the course.

Viewing Simulation Results - VCD

- The $dumpfile and $dumpvars system tasks will work with any Verilog simulator.
- The commands will open a file called “dump.vcd” and place all of the simulation data in that file. The file can then be opened by various waveform viewers and other tools.

```
initial begin
  $dumpfile (“dump.vcd”);
  $dumpvars;
end
```

Testbench (cont.)

- module testbench;
- // Data type declaration
- // Instantiate modules
- // Apply stimulus
- // Display results
- endmodule

```
module testbench;
reg [3:0] r1, r2;
reg ci;
wire [3:0] result

add4bits u1 (result, carry, r1, r2, ci);

initial begin
  . . .
end
initial begin
 $timeformat(-9,2,“ ns”, 10);
 $monitor(“time=%t  r1=%b  r2=%b  ci=%b
      result=%b  carry=%b”, $realtime, r1, r2, ci,
      result, carry);
end
endmodule
```

Simulation results to screen

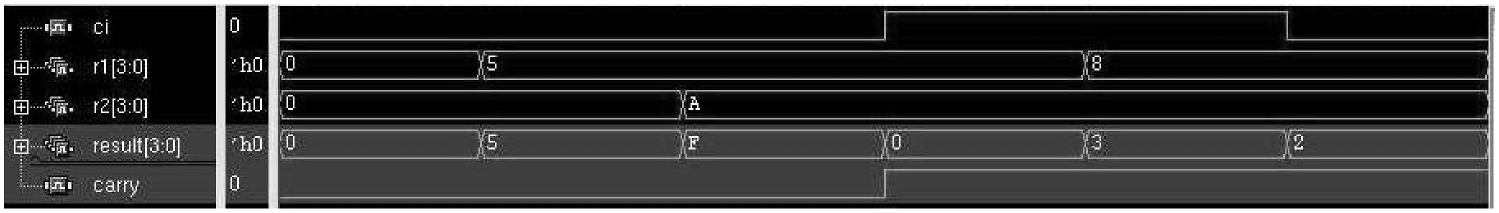
- To Simulate the design we have to include all the files containing the design modules.
  - For example, our files are named: ***testbench.v add4bits.v addbit.v***
- Simulation results:

```
Compiling source file ”testbench.v"
Compiling source file "add4bits.v"
Compiling source file "addbit.v"
Highest level modules:
testbench
   0.00 ns  r1=0000  r2=0000  ci=0  result=0000  carry=0
  10.00 ns  r1=0101  r2=0000  ci=0  result=0101  carry=0
  20.00 ns  r1=0101  r2=1010  ci=0  result=1111  carry=0
  30.00 ns  r1=0101  r2=1010  ci=1  result=0000  carry=1
  40.00 ns  r1=1000  r2=1010  ci=1  result=0011  carry=1
  50.00 ns  r1=1000  r2=1010  ci=0  result=0010  carry=1
L17 ”testbench.v": $finish at simulation time 60.00 ns
```

The various Verilog simulators require slightly different commands to compile and simulate the design. The ModelSim and Cadence software demonstrations will show how to load, compile, and simulate your designs.

Simulation results - Waveform

- Simulators generally provide a way to view the simulation waveforms.
- The waveform simulation results were stored in the VCD file.



Both ModelSim and Cadence have a waveform viewer built into the simulation tool.

Review

- What are the two methods for making connections to modules during instantiation?
- What is an “Instance Name?”
- What are the three elements to a testbench?
- What is the difference between the ‘initial’ and ‘always’ procedural blocks?

Homework

- Homework assignment #1 is on the class web page.
- Homework assignment #1 is due Wednesday, January 21, 2004.