Unit 2: SystemVerilog for Design

Adam Waksman
Simha Sethumadhavan
Hardware Description Languages (HDLs)

- Hardware life-cycle (8 steps)
  - Specification
    - High level specification
    - Architecture
    - Microarchitecture
  - HDL Design
  - HDL Validation
  - Synthesis
  - Layout
  - Fabrication (physical process)
Hardware Description Languages (HDLs)

- HDLs serve two different purposes
  - **Hardware Design**
    - Code defines the functionality of the hardware design
  - **Design Validation**
    - Creates a binary executable for validation/simulation
- Commercial tools automatic part of the process
  - **Synthesis**
    - Done automatically by compiler (Synopsys VCS)
  - **Layout**
    - Done with automated tools (Synopsys or Cadence)
Flavors of SystemVerilog

- **Structural SystemVerilog**
  - Low level, specify logic gates
  - Guaranteed to synthesize

- **Behavioral SystemVerilog**
  - Higher level language constructs
  - Not guaranteed to synthesize

For this class
- **Use behavioral SystemVerilog**
- Be careful of synthesizability
- All code should either:
  - Be synthesizable
  - Be explicitly for validation

```verilog
modulemux(s,x0,x1,y);
    input s;
    input x0;
    input x1;
    output y;
    wire sel_0;
    wire sel_1;

    AND2 XLI_1 (.I0(s),.I1(x1),.O(sel_1));
    AND2B1 XLI_2 (.I0(s),.I1(x0),.O(sel_0));
    OR2 XLI_3 (.I0(sel_1),.I1(sel_0),.O(y));
endmodule
```

```verilog
add_cost <= resize(Mux_out(0), 32);
add_cost_1 <= resize(Mux_out(1), 32);
add_temp <= resize(add_cost, 33) + resize(add_cost_1, 33);
sum_1 <= (31 => '0', OTHERS => '1') WHEN add_temp(32) == '0' AND add_temp(31) /= '0'
     ELSE (31 => '1', OTHERS => '0') WHEN add_temp(32) == '1' AND add_temp(31) /= '1'
     ELSE (add_temp(31 DOWNTO 0));
add_cost_2 <= sum_1;
add_cost_3 <= resize(Mux_out(2), 32);
add_temp_1 <= resize(add_cost_2, 33) + resize(add_cost_3, 33);
sum_2 <= (31 => '0', OTHERS => '1') WHEN add_temp_1(32) == '0' AND add_temp_1(31) /= '0'
     ELSE (31 => '1', OTHERS => '0') WHEN add_temp_1(32) == '1' AND add_temp_1(31) /= '1'
     ELSE (add_temp_1(31 DOWNTO 0));
```
Language Basics Outline

• Basic Types and Style
  • SystemVerilog Primitives
  • Basic Data Types
  • Assign and Always

• Building Larger Components
  • Parameters and Instantiation
  • Conditional Statements
  • Advanced Data Types
  • Advanced Example
SystemVerilog Primitives (pg. 37)

- Each primitive represents a signal carried by a wire
  - 0: Clear digital 0
  - 1: Clear digital 1
  - X: Means either “don’t know” or “don’t care”
    - Useful for debugging
    - Also useful for ‘don’t care’ bits in logic
  - Z: High impedance, non-driven circuit
    - Value is not clearly 0 or 1
    - Useful for testing, debugging, and tri-state logic
SystemVerilog Primitives (pg. 37)

- Each primitive represents a signal carried by a wire
  - 0: Clear digital 0
  - 1: Clear digital 1
  - X: Means either “don’t know” or “don’t care”
    - Useful for debugging
    - Also useful for ‘don’t care’ bits in logic
  - Z: High impedance, non-driven circuit
    - Value is not clearly 0 or 1
    - Useful for testing, debugging, and tri-state logic

- Constants/Multi-bit primitives
  - All wires carrying the same value
    - ‘1, ‘0, ‘z, ‘x
  - Specific values
    - 16'b1100101011111110
    - 16’d51966
    - 16’hcafe
  - Sets – example, odd numbers
    - 16’bxxxxxxxxxxxxxxxxx1
Basic Data Types: Wire (pg. 43)

- Wire (4-state variable)
  - Generic wire, can carry any signal (0, 1, x, z)
  - No semantic type safety between wires
    - Any wire can connect to any wire
  - Almost anything will compile
    - Including dangling wires, fused wires

- Use “assign” keyword to store a value
  - wire x;
  - assign x = '1;

- Can assign one wire to another
  - Wire x;
  - Assign x = y;
Basic Data Types: Wire (pg.43)

- **Assignment**
  - Assignments are permanent (think physical wires)
  - All assignments happen continuously and in parallel

```verilog
define wire a, b;
assign a = 1'b1;
assign b = a;
assign a = 1'b0;
```

- **This is incorrect code**
  - Cannot assign two values to ‘wire a’
  - SystemVerilog is not declarative
- **The compiler won’t warn you**
  - Validation might catch this
Basic Data Types: Wire (pg.43)

• Splitting wires, accessing subsets
  • Wires can be arbitrarily wide
  • Subsets accessed with [x:y] (0-based)

    wire [31:0] myInt;
    wire [63:0] myLong;
    assign myLong [31:0] = myInt;
    assign myLong [63:32] = '0;

• Can access a region or a single bit

    wire [31:0] value;
    wire fifthBit; // 0-based (this is the sixth bit 1-based)
    assign fifthBit = value[5];

• Example: Decoder

    wire [31:0] instruction;
    wire [5:0] opcode;
    assign opcode = instruction[31:26];
Basic Data Types: Input, Output, Module  
(pg. xxv)

- **Inputs and outputs**
  - Inputs are wires that come in from the “outside world”
  - Outputs are wires that go out to the “outside world”

- **Module**
  - One discrete piece of hardware
  - Can be instanced multiple times

```verilog
module adder(a, b, cin, cout, s);

input a, b, cin;
output cout, s;

assign s = a ^ b ^ cin;
assign cout = (a & b) | (a & cin) | (b & cin);

endmodule
```
Wire Concatenation and Replication

• The \{ \} operator does literal concatenation
  • Example, wire doubler

module doubler (in, out);
  input [31:0] in;
  output [63:0] out;
  assign out = \{in, in\};
endmodule

0xdecafbad -> 0xdecafbaddecafbad

• The {{ }} operator does replication
  • Example, sign extension

module signExtend (in, out);
  input [31:0] in;
  output [63:0] out;
  assign output[31:0] = in;
  assign output[63:32] = \{32{in[31]}\};
endmodule

0xdecafbad -> 0xffffffffdecafbad
Basic Data Types: logic (pg.43)

• Logic
  – Can be assigned values with the = operator
  – Do not synthesis actual state, only logic

• Always blocks (always_comb)
  – Allow assignment to logic variables
  – Signify an action that should happen continuously

Declare module
  Declare inputs
  Declare outputs
  Declare logic
  Declare wires

  always_comb begin
    combinatorial logic
  end

  Declare assignments
endmodule
Basic Data Types: logic (pg.43)

- **Logic**
  - Can be assigned values with the = operator
  - Do not synthesis actual state, only logic

- **Always blocks (always_comb)**
  - Allow assignment to logic variables
  - Signify an action that should happen continuously

```plaintext
Declare module
  Declare inputs
  Declare outputs
  Declare logic
  Declare wires

  always_comb begin
    combinatorial logic
  end

  Declare assignments
endmodule

module adder(a_i, b_i, out_o);
  input a_i, b_i;
  output out_o;
  logic sum;

  always_comb begin
    sum = a_i + b_i;
  end

  assign out_o = sum;
endmodule
```
Combinatorial vs. Sequential (flip-flop) (pg. xxv)

- **Combinatorial**
  - Logic/arithmetic that has no notion of time
  - Use ‘logic’

- **Sequential**
  - Logic that uses state to occur over multiple timesteps
  - Use ‘reg’

```verilog
module adder(a_i, b_i, out_o);
  input a_i, b_i;
  output out_o;
  logic sum;
  always_comb begin
    sum = a_i + b_i;
  end
  assign out_o = sum;
endmodule
```

```verilog
module flip_flop(clk, data_i, data_o);
  input data_i, clk;
  output data_o;
  reg data;
  always_ff @(posedge clk) begin
    data <= data_i;
  end
  assign data_o = data;
endmodule
```
Combinatorial vs. Sequential (latch) (pg. xxv)

- **Combinatorial**
  - Logic/arithmetic that has no notion of time
  - Use ‘logic’

- **Sequential**
  - Logic that uses state to occur over multiple timesteps
  - Use ‘reg’

```verilog
module adder(a_i, b_i, out_o);
  input a_i, b_i;
  output out_o;
  logic sum;

  always_comb begin
    sum = a_i + b_i;
  end

  assign out_o = sum;
endmodule
```

```verilog
module latch(clk, data_i, data_o);
  input data_i, clk;
  output data_o;
  reg data;

  always_latch begin
    if (clk) data <= data_i;
  end

  assign data_o = data;
endmodule
```
Style Conventions

• Logic vs. Reg
  • Use logic for combinatorial logic
  • Use reg only for memory

• Make a separate flip-flop file
  • You usually only need to declare a reg once per project
  • In other files use always_comb

```verilog
module uses_ff(clk, data_i, data_o);
    input data_i, clk;
    output data_o;

    flip_flop(clk, data_i, data_o);
endmodule
```
Example: Simple MUX

- Exercise: Design a 2-to-1 multiplexer

```verilog
module mux (data0_i, data1_i, select_i, data_o);

// Your code

endmodule
```
Example: Simple MUX

• In structural SystemVerilog

```systemverilog
class_module mux (data0_i, data1_i, select_i, data_o);
  input data0_i, data1_i, select_i;
  output data_o;

  assign data_o = (select_i & data1_i) | (!select_i & data0_i);
endmodule
```
Example: Simple MUX

• In behavioral SystemVerilog (with conditionals)

```verbatim
module mux (data0_i, data1_i, select_i, data_o);
    input data0_i, data1_i, select_i;
    output data_o;

    assign data_o = select_i ? data1_i : data0_i;
endmodule
```

• If/Else statements are C-style
  – Only work inside of always blocks
Example: Simple MUX

- Using an `always_comb` block and `case` statement

```vhdl
module mux (data0_i, data1_i, select_i, data_o);
  input data0_i, data1_i, select_i;
  logic data;
  output data_o;

  always_comb begin
    case(select_i)
      '0: data = data0_i;
      '1: data = data1_i;
    endcase
  end

  assign data_o = data;
endmodule
```
Example: Simple MUX

- Concise version

```verilog
module mux (data0_i, data1_i, select_i, data_o);
  input data0_i, data1_i, select_i;
  output logic data_o;

  always_comb begin
    case(select_i)
      '0: data_o = data0_i;
      '1: data_o = data1_i;
    endcase
  end

endmodule
```
Basic Compilation

- Log on a Columbia (CLIC) machine
  
  ```bash
  ssh -X YOUR_UNI@clic-lab.cs.columbia.edu
  ```

- Set up environment (can use ~/.bashrc)
  
  ```bash
  Source /sim/synopsys64/env_castl64.sh
  ```

- Compile a module called MODULE_NAME
  
  ```bash
  vcs -sverilog MODULE_NAME.sv -o EXECUTABLE_NAME
  ```

- This compiles a binary executable for simulation
- Synthesis will be discussed later in the course
• What are the basic data types in SystemVerilog?
  input, output, wire, reg, logic, module.

• What is the key difference between assignment in SystemVerilog and assignment in a procedural language (like C)?
  SystemVerilog assignments are continuous and occur in parallel.

• What is the difference between sequential logic and combinatorial logic?
  Sequential logic occurs over multiple clock cycles in a synchronized fashion. Combinatorial logic is a single logical function.
Questions

"Are they fresh?"
Language Basics Outline

• Basic Types and Style
  • SystemVerilog Primitives
  • Basic Data Types
  • Assign and Always

• Building Larger Components
  • Parameters and Instantiation
  • Conditional Statements
  • Advanced Data Types
  • Advanced Example
Module Instantiation (pg. 224)

• Instantiating a module
  • All inputs and outputs must be defined
  • Abstraction: analogous to a constructor in Java or C++

• Example
  • A four bit ripple-carry adder from 4 separate full-adders (FA)
  • The module we’re instancing is a user-defined type

module fourbitadder
(
  input [3 : 0] a,
  input [3 : 0] b,
  output c4,
  output [3 : 0] s,
);
  wire c0, c1, c2, c3;
  assign c0 = 1’b0;
  FA bit0 (a[0], b[0], c0, c1, s[0]);
  FA bit1 (a[1], b[1], c1, c2, s[1]);
  FA bit2 (a[2], b[2], c2, c3, s[2]);
  FA bit3 (a[3], b[3], c3, c4, s[3]);
endmodule
The `. Operator

- Allows for reordering of arguments
- A REQUIRED stylistic choice

Example

- Same four bit ripple-carry adder from 4 separate full-adders (FA)

```verilog
module 
    (input [3 : 0] a,
     input [3 : 0] b,
     output c4,
     output [3 : 0] s,
    );
    wire c0, c1, c2, c3;
    assign c0 = 1'b0;
    FA bit0 (.a(a[0]), .b(b[0]), .cin(c0), .cout(c1), .s(s[0]));
    FA bit1 (.a(a[1]), .b(b[1]), .cin(c1), .cout(c2), .s(s[1]));
    FA bit2 (.a(a[2]), .b(b[2]), .cin(c2), .cout(c3), .s(s[2]));
    FA bit3 (.a(a[3]), .b(b[3]), .cin(c3), .cout(c4), .s(s[3]));
endmodule
```
Module Instantiation (pg. 224)

- **Auto-instantiation**
  - Corresponding variables must have the same name and size
  - Useful when instantiating one instance of something

- **Example**
  - Using a MUX inside of a module

```verilog
module contains_MUX 
  ( 
    input data0_i, 
    input select_i, 
    output data_o 
  ); 

  wire data1_i; 
  assign data1_i = '1; 
  MUX myMux (.data0_i, .data1_i, select_i, .data_o); 
endmodule
```
Module Instantiation (pg. 224)

- Can pass literals

- Can pass arguments in any order

module contains_MUX
(
    input data0_i,
    input select_i,
    output data_o
);

    MUX myMux (.data1_i(‘1), .data0_i, .data_o, .select_i);
endmodule
Parameters and Style

- Sets a value when the module is instanced
  - Equivalent to a constructor argument

module FF #(parameter WIDTH = 1)
(
    input clk,
    input [WIDTH – 1 : 0] data_i,
    output [WIDTH – 1 : 0] data_o
);

    reg [WIDTH – 1: 0] data;

always_ff @(posedge clk) begin
    data <= data_i;
end

assign data_o = data;
endmodule
Parameters and Style

- Sets a value when the module is instanced
  - Equivalent to a constructor argument

```verbatim
module FF #(parameter WIDTH = 1) (input clk, input [WIDTH – 1 : 0] data_i, output [WIDTH – 1 : 0] data_o);
  reg [WIDTH – 1: 0] data;
  always_ff @(posedge clk) begin
    data <= data_i;
  end
  assign data_o = data;
endmodule
```

```verbatim
module register (input clk, input [31 : 0] data_i, output [31 : 0] data_o);
  FF #( .WIDTH(32)) ff32 (clk, data_i, data_o);
endmodule
```
Conditional Statements (pg. 195)

• Usually implies a multiplexer
  • Syntactic sugar
  • Can always be done equivalently with structural Verilog

• Combinatorial always block
  • Changes whenever the input changes
  • The logic here does not create memory

module priorityEncoder
(  
  input [7:0] data_i;
  output logic [2:0] data_o;
);
always_comb begin
  if (data_i[0]) data_o = '0;
  else if (data_i[1]) data_o = 3'b001;
  else if (data_i[2]) data_o = 3'b010;
  else if (data_i[3]) data_o = 3'b011;
  else if (data_i[4]) data_o = 3'b100;
  else if (data_i[5]) data_o = 3'b101;
  else if (data_i[6]) data_o = 3'b110;
  else if (data_i[7]) data_o = 3'b111;
  else data_o = 'x;
end
endmodule
• Usually implies a multiplexer
  • Syntactic sugar
  • Can always be done equivalently with structural Verilog

module priorityEncoder
(
    input [7:0] data_i;
    output logic [2:0] data_o
);
always_comb begin
  casex(data_i)
    8'bxxxxxxxx1: data_o = '0;
    8'bxxxxxx10: data_o = 3'b001;
    8'bxxxxx100: data_o = 3'b010;
    8'bxxxx1000: data_o = 3'b011;
    8'bxxx10000: data_o = 3'b100;
    8'bx100000: data_o = 3'b101;
    8'b1000000: data_o = 3'b110;
    8'b10000000: data_o = 3'b111;
    default: data_o = 'x;
  endcase
endmodule
Advanced Data Types (pg. 44)

• Two-state variables
  – Can synthesize into wires but take on only 1 or 0
  – We’ll only use for non-synthesizing elements
    
    bit [3:0] a_nibble; // same as 4 wires but can only be 0s or 1s
    byte a_byte; // an 8-bit wire
    shortint an_int; // 16-bit wire
    int a_normal_int; // 32-bit wire
    longint a_long; // 64-bit wire

• Introduces typedefs (similar to C)
  
  typedef int [3 : 0] four_ints;
  fourInts my_four_ints;
• Basic arrays are similar to C

reg [255 : 0] my_registers; // 256 storage bits
reg [255 : 0] my_registers [31 : 0]; // 256 storage ints

• Parameters allow variable sizes for both dimensions

parameter STORAGE_SIZE = 256;
parameter INT_SIZE = 32;

reg [STORAGE_SIZE – 1 : 0] my_registers; // 256 storage bits
reg [STORAGE_SIZE – 1 : 0] my_registers [INT_SIZE – 1 : 0]; // 256 storage ints

• Associative arrays

typedef reg [ADDRESS_WIDTH – 1 : 0] address;
reg [DATA_SIZE – 1 : 0] memory [address];
Advanced Data Types: Structs (pg. 97)

- Essentially the same as in C
  
  ```
  struct {
    bit sign;
    bit [10:0] exponent;
    bit [51:0] mantissa;
  } float;
  ```

- Can be used in array types, other structs, or typedefs
  - “packed” signifies adjacency

  ```
  typedef struct packed {
    bit sign;
    bit [10:0] exponent;
    bit [51:0] mantissa;
  } float;
  
  typedef float [255:0] float_array;
  
  float_array my_float_array;
  ```
Real Design: 1 KB RAM
module eff #(parameter WIDTH = 1) 
( 
    input clk,
    input write_enable_i,
    input read_enable_i,
    input [WIDTH - 1 : 0] data_i,
    output logic [WIDTH - 1 : 0] data_o
);

reg [WIDTH - 1: 0] data;

always_ff @(posedge clk) begin
    if (write_enable_i) data <= data_i;
end

always_comb begin
    data_o = read_enable_i ? data : '0;
end
endmodule
Real Design: 1 KB RAM

Decoder

MUX
Decoder for RAM: For loops

module ram_decoder #(parameter DATA_WIDTH = 32,
   parameter ADDR_WIDTH = 8,
   parameter DEPTH = (1 << ADDR_WIDTH),
   parameter SIZE = (DEPTH * DATA_WIDTH))
(
   input write_enable_i,
   input [ADDR_WIDTH - 1 : 0] address_i,
   output logic [DEPTH - 1 : 0] write_enable_o,
   output logic [DEPTH - 1 : 0] read_enable_o
);

logic [DEPTH - 1 : 0] enable;
always_comb begin
   for (int iter = 0; iter < DEPTH; iter++) begin
      enable[iter] = (iter == address_i);
      read_enable_o[iter] = enable[iter];
      write_enable_o[iter] = enable[iter] & write_enable_i;
   end
end
endmodule
Real Design: 1 KB RAM

DECODER

MUX
MUX for RAM: For loops

module ram_mux #(parameter DATA_WIDTH = 32,
                  parameter ADDR_WIDTH = 8,
                  parameter DEPTH = (1 << ADDR_WIDTH),
                  parameter SIZE = (DEPTH * DATA_WIDTH))
(  
  input [SIZE - 1 : 0] data_i,
  output logic [DATA_WIDTH - 1 : 0] data_o
);

always_comb begin
  for(int bit_in_word = 0; bit_in_word < DATA_WIDTH; bit_in_word++) begin
    data_o[bit_in_word] = '0;
    for(int bit_location = bit_in_word; bit_location < SIZE; bit_location += DATA_WIDTH) begin
      if (data_i[bit_location]) data_o[bit_in_word] = '1;
    end
  end
endmodule
Real Design: 1 KB RAM

Decoder

MUX

MUX


generate
  for (genvar iter = 0; iter < DEPTH; iter++) begin
    eff #(.WIDTH(DATA_WIDTH)) memory_word
      (.clk,
       .data_i,
       .write_enable_i(write_enable[iter]),
       .read_enable_i(read_enable[iter]),
       .data_o(all_data[(iter * DATA_WIDTH) + DATA_WIDTH - 1 -: DATA_WIDTH]))
    end
  endgenerate
module ram #(parameter DATA_WIDTH = 32,
    parameter ADDR_WIDTH = 8,
    parameter DEPTH = (1 << ADDR_WIDTH),
    parameter SIZE = (DEPTH * DATA_WIDTH))
(
    input clk,
    input write_enable_i,
    input [ADDR_WIDTH - 1 : 0] address_i,
    input [DATA_WIDTH - 1 : 0] data_i,
    output logic [DATA_WIDTH - 1 : 0] data_o
);
Top Level Module: Body

// MUX
wire [SIZE - 1 : 0] all_data;
ram_mux mux (.data_i(all_data), .data_o);

// Decoder
wire [DEPTH - 1 : 0] read_enable;
wire [DEPTH - 1 : 0] write_enable;
ram_decoder decoder (.write_enable_i, .read_enable_o(read_enable), .write_enable_o(write_enable));

// Memory
generate
  for (genvar iter = 0; iter < DEPTH; iter++) begin
    eff #(DATA_WIDTH) memory_word (.clk, .data_i, .write_enable_i(write_enable[iter]), .read_enable_i(read_enable[iter]), .data_o(all_data[(iter * DATA_WIDTH) + DATA_WIDTH - 1 -: DATA_WIDTH]));
  end
endgenerate
endmodule
Questions

"Are they fresh?"
Appendix: Inout

• Inout refers to a wire for tri-state logic
  – Also called a bidirectional pin

    inout bidirectional;

• Tri-state wires can carry 0, 1, or z
  – When the value is z, it functions as an input wire
  – When the value is 0 or 1, it's an output wire

    assign bidirectional = outputEnable ? input : z ;
System Verilog: Synthesizeable Set

Processes
- assign statements
- always_comb, always_ff

Operators
- . , .* Operator
- Basic logic operations

Datatypes and Literals
- Logic (4 state)
- Typedef (user defined types)
- Enumerations
- Structures
- Literals

Disallow SET
- # delays
- Initialization
- No tasks and functions
- Auto increment, decrement
- Statically unknown bounds

Interfaces
- Generic Interface
- Interface ports
- Interface modports
- Parameterized Interfaces

For loop, Generate (with caution)
Typedef/Struct Example

Verilog

```verilog
module fifo (clk, rstp, din_src, din_dst, din_data, readp, writep, dout_src, dout_dst, dout_data, emptyp, fullp);
input clk;
input rstp;
input [7:0] din_src;
input [7:0] din_dst;
input [31:0] din_data;
input readp;
input writep;
output [7:0] dout_src;
output [7:0] dout_dst;
output [31:0] dout_data;
output emptyp;
output fullp;
.
.
```

SystemVerilog

```systemverilog
typedef struct {
  logic [7:0] src, dst;
  logic [31:0] data;
} packet_t;

module fifo (clk, rstp,
input packet_t din,
input readp;
input writep;
output packet_t dout;
output logic emptyp;
output logic fullp
);
```

Define Once

Use many times

packet_t is a port
Interfaces

Signals, direction specified in ALL modules

Signals, direction specified ONCE inside interface

(c) Synopsys
Simple Design Skeleton

Synthesized Design:

Source Code:

```vhdl
module CPU ();
    ...
endmodule

module RAM ();
    ...
endmodule

module Top;
    CPU CPU();
    RAM RAM();
endmodule
```
Interface definition

Synthesized Design:

```
interface chip_bus ()
    wire request, grant, ready;
    wire [47:0] address;
    wire [63:0] data;
endinterface

module CPU ();
    ...
endmodule

module RAM ();
    ...
endmodule

module Top;
    chip_bus a();
    CPU CPU();
    RAM RAM();
endmodule
```
Passing Interface as a Port

Synthesized Design:

```
interface chip_bus ();
    wire request, grant, ready;
    wire [47:0] address;
    wire [63:0] data;
endinterface

module CPU ( chip_bus io  );
endmodule

module RAM ( chip_bus pins );
endmodule

module Top;
    chip_bus a();
    CPU CPU(a);
    RAM RAM(a);
endmodule
```

Connection details are in the interface

Ports specified for modules

Modules do not duplicate connection detail
Interface Modports

Synthesized Design:

```
interface chip_bus ();
  logic request, grant, ready;
  wire [47:0] address;
  wire [63:0] data;

  modport MPl(output request, input grant, ...);

endinterface

module CPU (chip_bus.MPl io);
endmodule

module RAM ();
endmodule

module Top;
  chip_bus a();
  CPU CPU(a.MPl);
  RAM RAM(
endmodule
```
Interface Modports (2)

```verilog
interface chip_bus();
logic request, grant, ready;
wire [47:0] address;
wire [63:0] data;

modport MP1(output request, input grant, ...);
modport MP2(input request, output grant, ...);
endinterface

module CPU (chip_bus.MP1 io);
endmodule

module RAM (chip_bus.MP2 pins);
endmodule

module Top;
chip_bus a();
CPU CPU(a.MP1);
RAM RAM(a.MP2);
endmodule
```
Ports

NEW PORT CONNECTION STYLES

- Top level design have too many ports that need to be connected to lower level modules.
- Using .* and .name styles simplifies Top-Level code size – less typing and less bugs!

```verilog
module dffdff1(clk, q, d, rst);

endmodule
```
Coding Guidelines

• Use synthesizeable subset of SystemVerilog
• One module definition per file
• The name of the file should be same as the module name
• Do not infer latches (separate out logic and memory)
• Use non-blocking assignment for sequential memory and blocking assignment for combinational memory
• Use always_comb for combinational logic and always_ff for sequential memory
• Do not use latches (unless explicitly permitted)
• Use pre-built design blocks whenever possible
• Include a header file that includes: filename, last modified date and ownership date.
• Use _i for inputs and _o for outputs
Inputs and Outputs must be clearly marked along with bit widths
Also customary to provide the same in a tabular form
Specify Unit Functionality

- A CAM is a special type of memory array used for providing fast search functionality. Typically, memories are organized as tables for storing information and are used as random access memories (RAMs) i.e., you can retrieve a value at a location in a RAM “table” by specifying an index. A CAM, on the other hand, returns the index of the location containing a value that is supplied to the CAM. In a simple CAM, a supplied value is searched against all entries in the CAM simultaneously and the index of the first fully matching location is returned. CAMs find use in internet network routers for IP lookup, microprocessor structures such as TLBs, load/store queues and issue queues to name a few.

- Describe functions in English and “C” style pseudo code

```c
// int CAM[32];
// bool found;
// int key;

search(int key) {
    int I;
    for (I = 0; I < 32; I++) {
        if (CAM[I] == key) {
            found = 1;
            break;
        } else {
            found = 0;
        }
    }
    return found, I;
}

read(int i) {
    return CAM[i];
}

write (int I, int data) {
    CAM[I] = data;
}
```
Hints

(Try design partitioning, and block diagrams before next lectures)
What “parts” do you have to build the CAM?

Step 1: We need a memory element to hold the values.
    We know how to write to FF, read from a FF.

Step 2: Enhance Flip Flop to check if the stored bit is same
    as searched bit.

Step 3: We are searching for words not bits,
    so create a line from the cell.

Step 4: We need multiple words, so stack
    Cam lines.
Step 5: Need ability to read/write to a particular entry in CAM

Step 6: Need ability to select first matching entry through the CAM