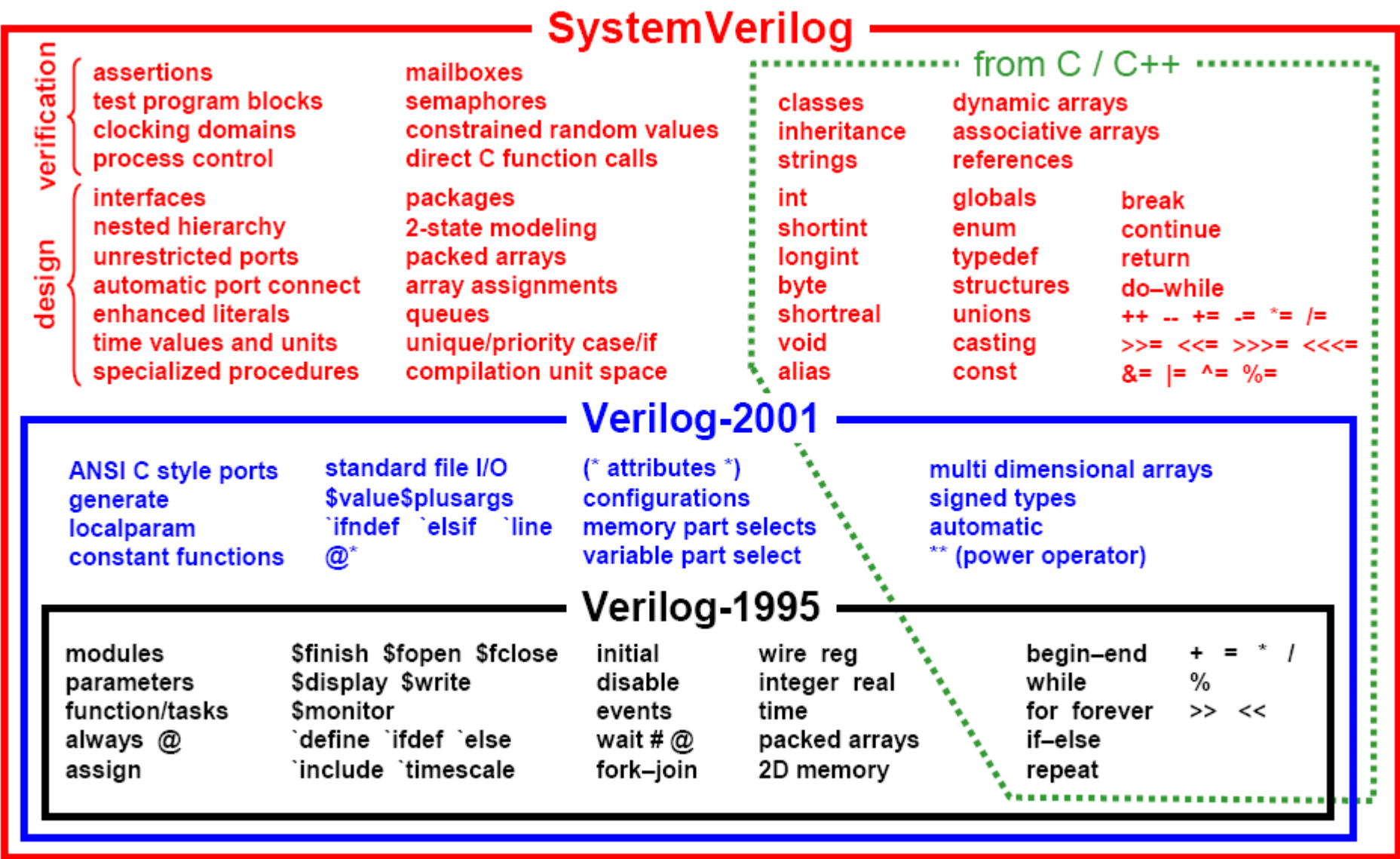


Outline

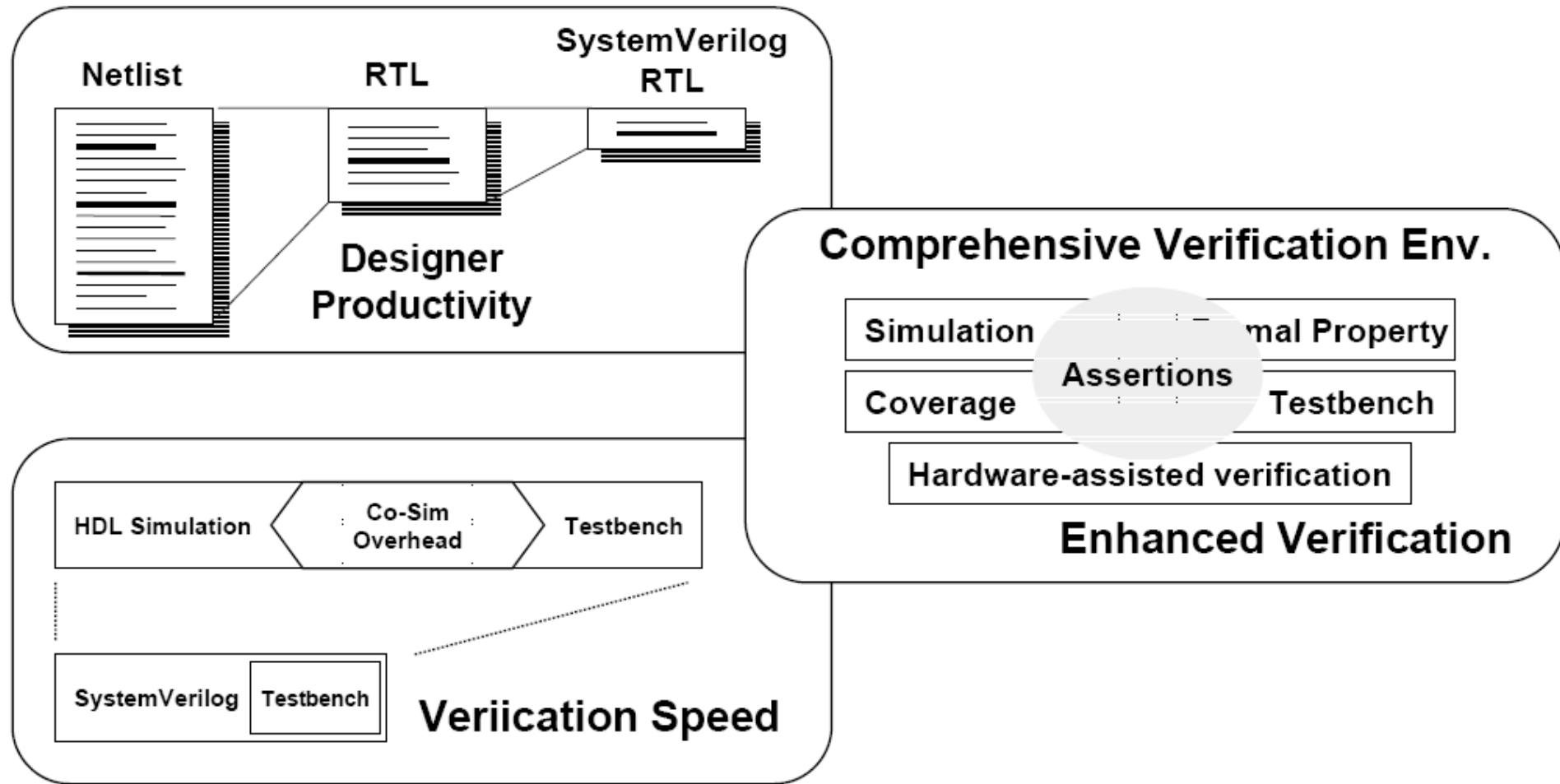
- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1 (Functional viewpoints)
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法 2 (Language viewpoints)
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- **SystemVerilog**
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

SYSTEMVERILOG DESIGN ENHANCEMENT (PARTIAL)

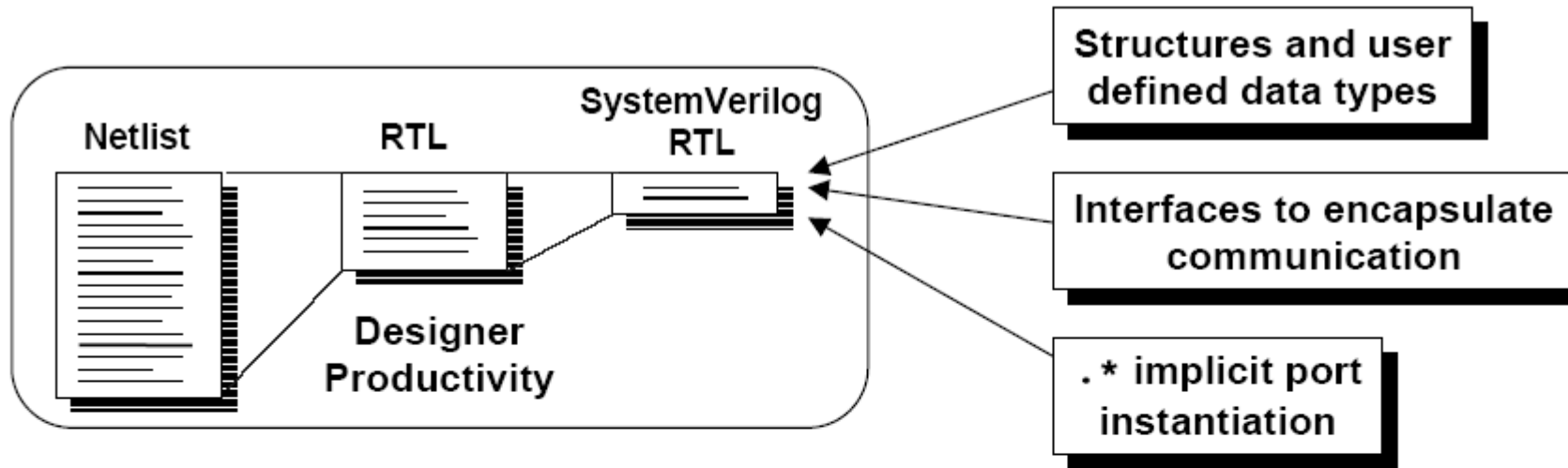
SystemVerilog is a Huge Extension of Verilog



SystemVerilog Means Productivity



Increase Designer Productivity

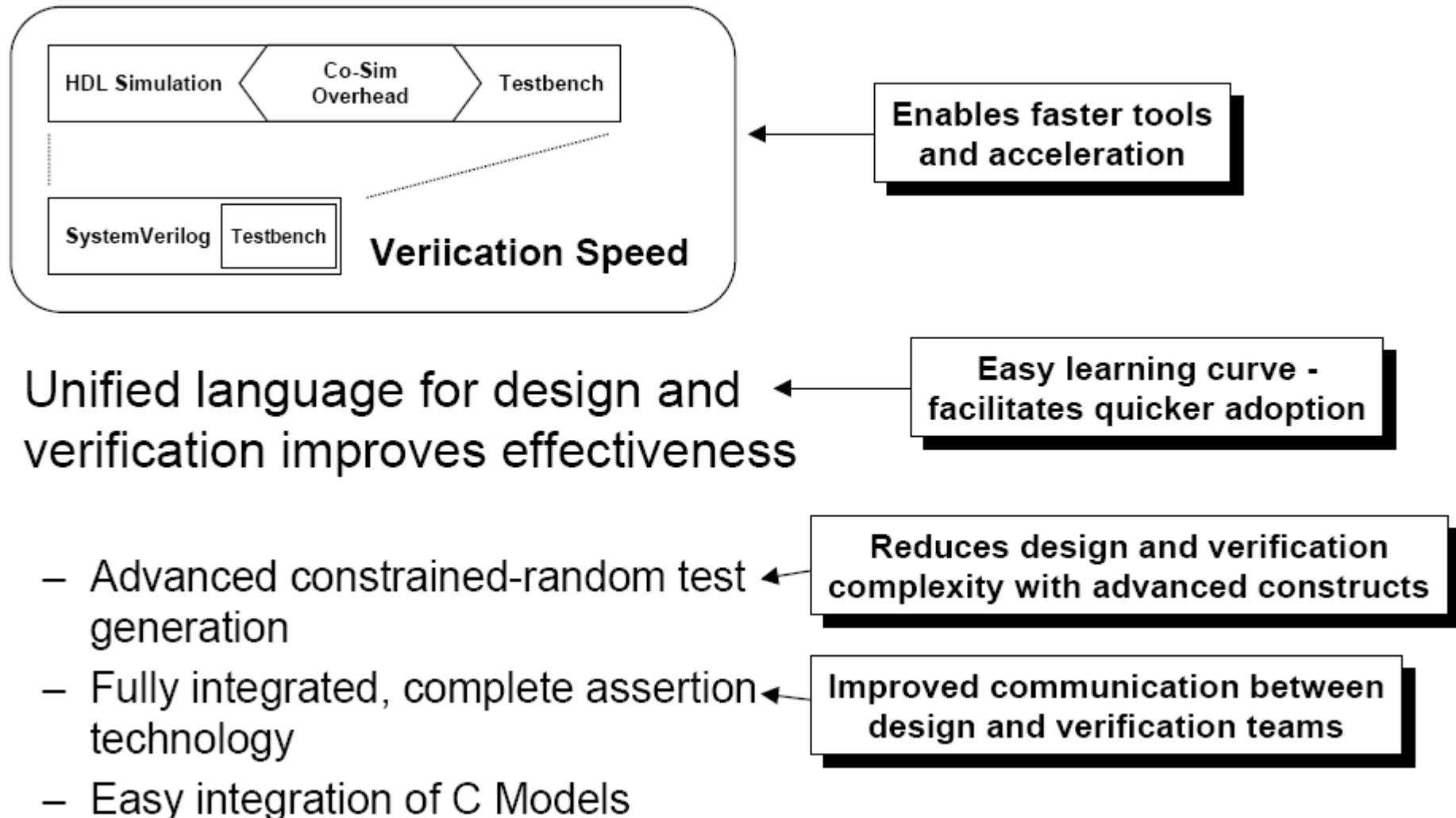


- 2X – 5X less code to capture the same functionality
 - Less code → Fewer bugs
 - Easier to interpret and communicate among teams
- Evolutionary: Reduces learning curve

New constructs to help *eliminate simulation and synthesis mismatches*

Less code & still synthesizable

Single Language for Design and **Verification**: HDVL



SystemVerilog Data Types

- SystemVerilog has:

- 4-state data types

0, 1, X, Z

Uninitialized variables = X
Uninitialized nets = Z
(same as Verilog-2001)

- 2-state data types

0, 1

Uninitialized variables = 0
Uninitialized nets* = 0
(new to SystemVerilog)

New to
SystemVerilog

```
reg      r; // 4-state, Verilog-2001 (sizeable) data type
integer i; // 4-state, Verilog-2001 32-bit signed data type
logic    w; // 4-state, (sizeable) 0, 1, X or Z

bit       b; // 2-state, (sizeable) 1-bit 0 or 1
byte      b8; // 2-state, 8-bit signed integer
shortint  s; // 2-state, 16-bit signed integer
int       i; // 2-state, 32-bit signed integer
longint   l; // 2-state, 64-bit signed integer
```

```
reg [15:0] r16;
logic [15:0] w16;
bit [15:0] b16;
```

reg, logic & bit
can be sized

Other data types have also been added

* - **bit** can be used as either a variable or a net - more later

Almost Universal Data Types: **logic** (or reg)

- **logic** is roughly equivalent to the VHDL `std_ulogic` type
 - Unresolved
 - Either permits only a single driving source -OR- procedural assignments from one or more procedural blocks

– In SystemVerilog: **logic** and **reg** are now the same (like **wire** and **tri** in Verilog)

Illegal to make both continuous assignments and procedural assignments to the same variable

logic is a 4-state type

bit is an equivalent unresolved 2-state type

Error code

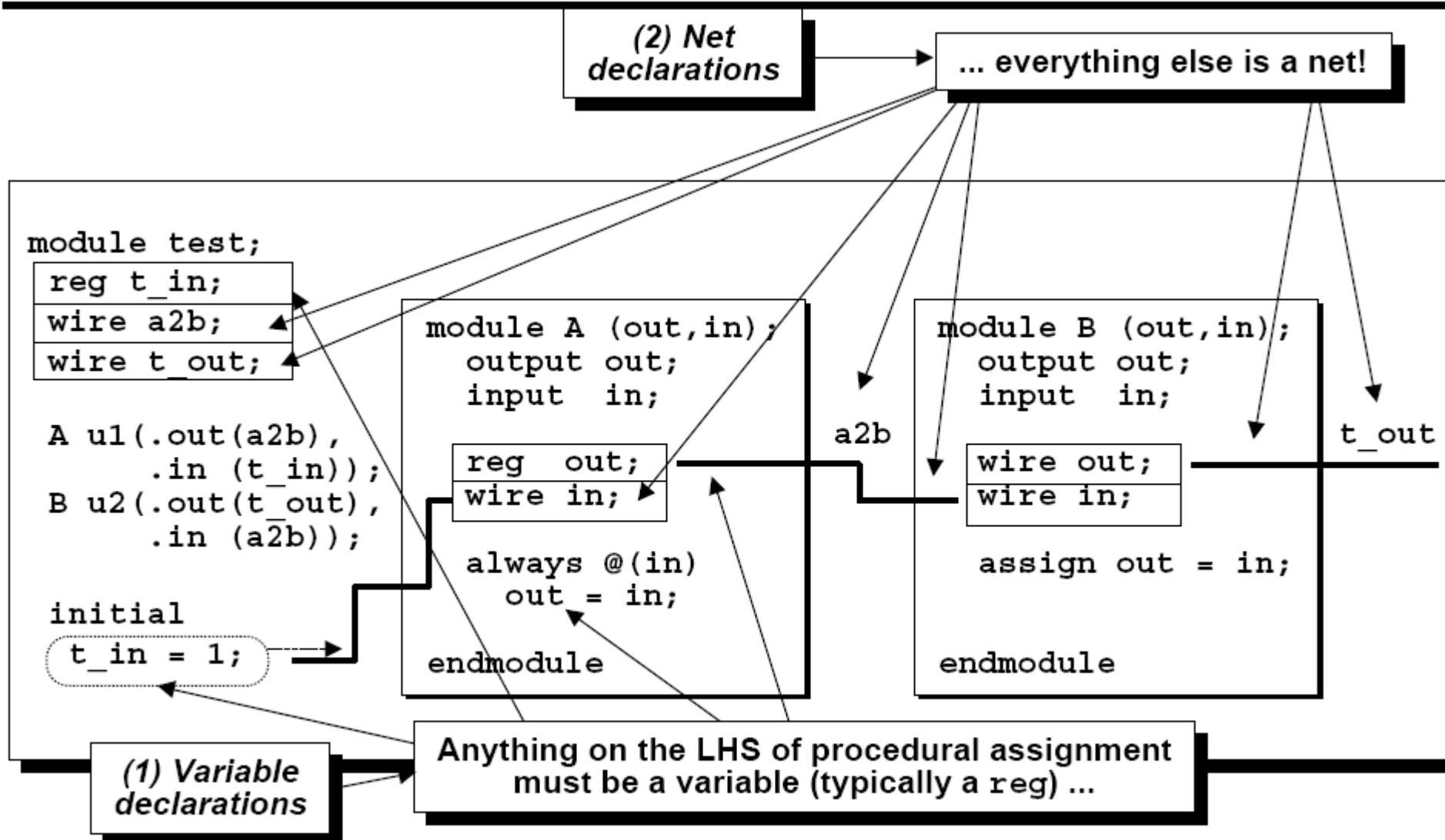
```
assign a = b + 1;
assign a = c + 2;
```

```
always_comb
  a = d + 3;
```

Logic is a better name than reg, so is preferred

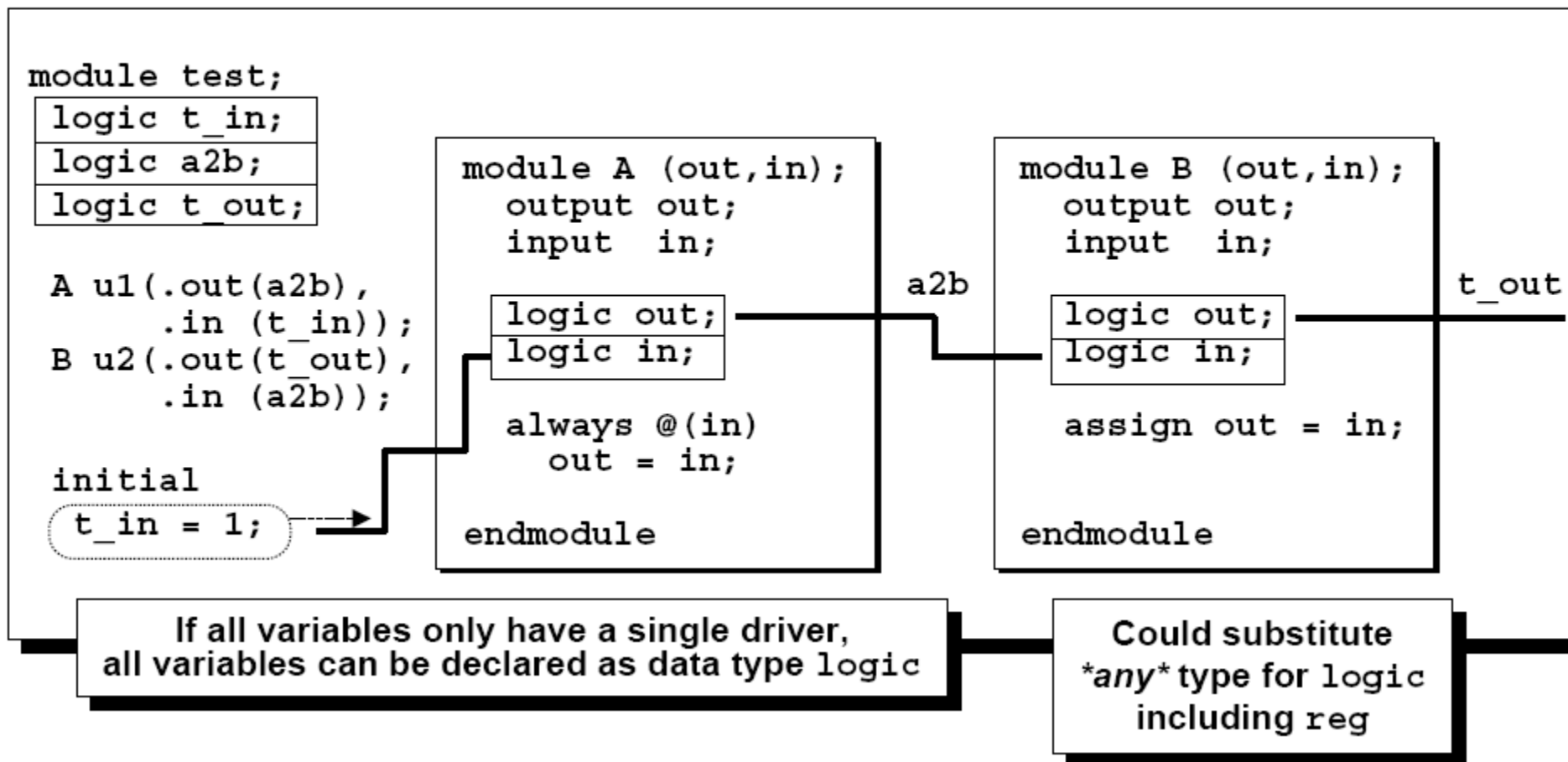
- **wire** net type still required for:
 - Multiply driven buses (such as bus crossbars and onehot muxes)
 - Bi-directional buses (more than one driving source)

Verilog 2001 Data Types



Almost Universal Data Types: logic

- SystemVerilog permits procedural or continuous assignments to variables



User Defined Types: typedef

- Allows the creation of either *user-defined* or *easily-changable* type definitions
 - Good naming convention uses "_t" suffix

```
typedef existing_type mytype_t;
```

defines.vh

```
`ifndef STATE2
  typedef bit    bit_t; // 2-state
`else
  typedef logic bit_t; // 4-state
`endif
```

Make your own types using typedef
Use typedef to get C compatibility

typedef	shortint	short;
typedef	longint	longlong;
typedef	real	double;
typedef	shortreal	float;

Design Strategy: Use All typedef's

- Interesting design strategy to switch easily between 4-state and 2-state simulations
 - Only use typedef-ed types

defines.vh

```
`ifndef STATE2
  typedef bit    bit_t; // 2-state
`else
  typedef logic  bit_t; // 4-state
`endif
```

tb.v

```
module tb;
  bit_t q, d, clk, rst_n;

  dff u1 (.q(q), .d(d), .clk(clk),
         .rst_n(rst_n));

  initial begin
    // stimulus ...
  end
endmodule
```

dff.v

```
module dff (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always @(posedge clk)
    if (!rst_n) q <= 0;
    else      q <= d;
endmodule
```

Default
4-state simulation

Faster
2-state simulation

```
verilog_cmd defines.vh tb.v dff.v
```

```
verilog_cmd defines.vh tb.v dff.v +define+STATE2
```

Enhanced Literal Number Syntax

```
module fsm_sv1b_3;  
  ...  
  
  always @* begin  
    next = 'x; ←  
    case (state)  
      ...  
  endmodule
```

Useful "fill" operations
similar to the VHDL command
(others => ...)

'x is equivalent to Verilog-2001 'bx

'z is equivalent to Verilog-2001 'bz

'1 is equivalent to making an assignment of -1
(2's complement of -1 is all 1's)

'0 is equivalent to making an assignment of 0
(for consistency)

Enumerated Data Types

- Allow to define a datatype whose values have names
 - Useful for state coding, op code, or symbolic data

```
enum {red, green, blue} RGB;
enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, DONE=3'b100} states;
```

Anonymous
2-state
int types

```
enum {bronze=3, silver, gold} medal;
```

silver=4, gold=5

```
enum {a=0, b=7, c, d=8} alphabet;
```

Syntax error
(implicit) c=8
(explicit) d=8

```
enum {bronze=4'h3, silver, gold} medal;
```

silver = 4'h4,
gold=4'h5

```
typedef enum {red, yellow, green, blue, white, black} colors_t;
```

```
colors_t light1, light2;
```

```
initial begin
    light1 = red;
```

```
    if (light1 == red) light1 = green;
```

```
end
```

traffic_light = 0 ("red")

traffic_light = 2 ("green")



Port Instance and Port Connections

- SystemVerilog Enhancements

- Inferred port connections using `.name` & `.*`

- SystemVerilog connections by interface - level I

- SystemVerilog connections by interface - level II

Includes: tasks, functions, assertions, etc.

Great for testbench development - the interface includes the legal interface commands

Great for IP development - the interface reports when it is used wrong

Concise instantiation

Encapsulation of
interface information

Added testbench and
assertion value



.Name Implicit Port with SystemVerilog

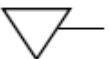
Barrel Shifter*
(0-16)

MUX

ALU (32-bit)

Accumulator*

Shifter (0,1,4)*



```
...
    barrel_shifter barrel_shifter (.bs, .data, .bs_lshft,
                                   .ld_bs, .clk, .rst_n);
mux2          mux                (.y(alu_in),
                                   .i0(multout), .i1(acc),
                                   .sel1(cal_u_muxsel));
alu           alu                (.alu_out, .zero(), .neg(),
                                   .alu_in, .acc, .alu_op);
accumulator   accumulator        (.acc, .alu_out, .ld_acc,
                                   .clk, .rst_n);
shifter       shifter            (.data, .acc, .shift_lshft,
                                   .ld_shift, .en_shift,
                                   .clk, .rst_n);
tribuf        tribuf             (.data, .acc(acc[15:0]),
                                   .en_acc);
endmodule
```

Use .name if port name
and interconnection
wire name are the same

All of the advantages of
named port connections

Less
verbose!



. * Implicit Port with SystemVerilog

```

module calu4 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input          calu_muxsel, en_shft, ld_acc, ld_bs,
  input          ld_multop1, ld_multout, ld_shft, en_acc,
  input          clk, rst_n);

  wire  [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire  [15:0] mop1;

  multop1      multop1      (. *);
  multiplier    multiplier    (. *);
  multoutreg    multoutreg    (. *);
  barrel_shifter barrel_shifter (. *);
  mux2          mux          (.y(alu_in), .i0(multout),
                               .i1(acc), .sel1(calu_muxsel));

  alu           alu           (. *, .zero(), .neg());
  accumulator    accumulator    (. *);
  shifter        shifter        (. *);
  tribuf         tribuf         (. *, .acc(acc[15:0]));
endmodule

```

**This style emphasizes
where port differences
occur**

**Much less
verbose!**

MultOp1 reg*

Multiplier

MultOut reg*

**Barrel Shifter*
(0-16)**

MUX

ALU (32-bit)

Accumulator*

Shifter (0,1,4)*

. * only needs to
specify the difference





Rule for .* and .name Connections

- Rule: mixing .* and .name ports in the same instantiation is prohibited
- Permitted:
 - .name and .name(signal) connections in the same instantiation
 - OR–
 - .* and .name(signal) connections in the same instantiation
- Rules: .name(signal) connections are required for:
 - size-mismatch ←

```
inst u1 (... , .data(data[7:0]), ...);
```
 - name-mismatch ←

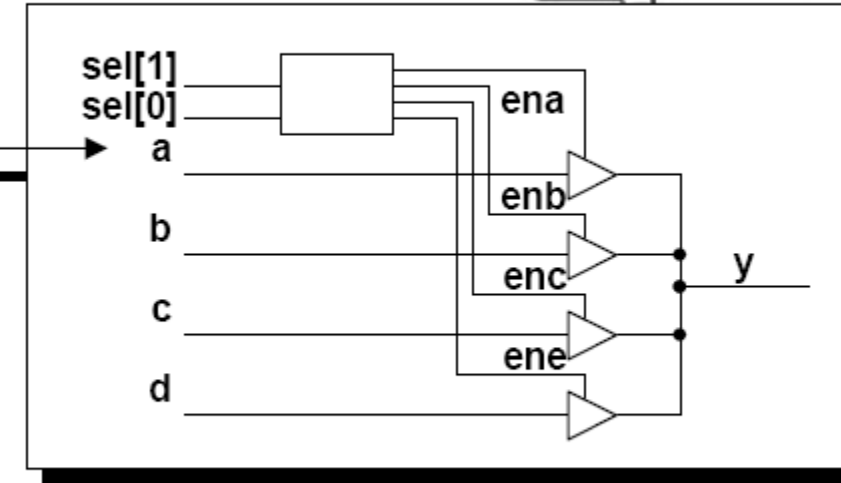
```
inst u2 (... , .data(pdata), ...);
```
 - unconnected ports ←

```
inst u3 (... , .berr( ), ...);
```

NOTE: stronger typing of ports than Verilog-2001 and more concise!

Disadvantage of Implicit * Port Connections

- Easy to connect the wrong ports to the wrong nets for common identifier names
 - Could be difficult to debug



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

  assign y = ena ? a : 8'bz;
endmodule
```

```
module driverb (
  output [7:0] y,
  input [7:0] b,
  input
  ena);

  assign y = ena ? b : 8'bz;
endmodule
```

```
module driverc (
  output [7:0] y,
  input [7:0] c,
  input
  ena);

  assign y = ena ? c : 8'bz;
endmodule
```

```
module driverd (
  output [7:0] y,
  input [7:0] d,
  input
  ena);

  assign y = ena ? d : 8'bz;
endmodule
```

Common enable name



Logic-Specific Processes

- SystemVerilog has three new logic specific processes to show designers intent:

`always_comb`

`always_latch`

`always_ff`

```
always_comb begin
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 ! tmp2;
end
```

```
always_latch
    if (en) q <= d;
```

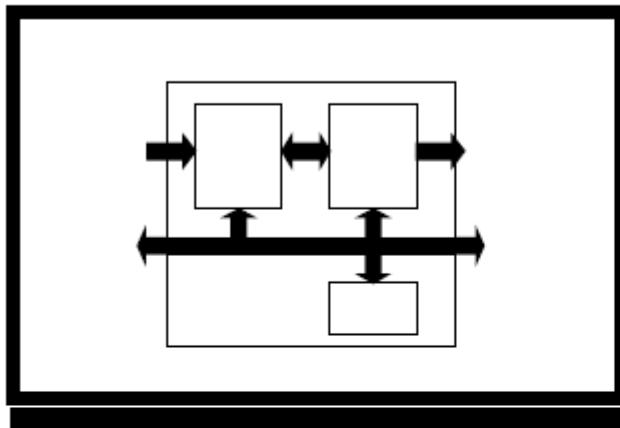
```
always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else      q <= d;
```

Allows simulation tool to perform
some linting functionality

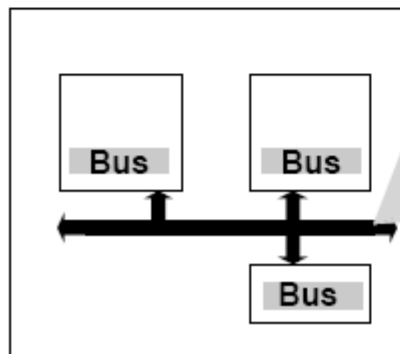


SystemVerilog Interface

Design On A White Board



SystemVerilog Design

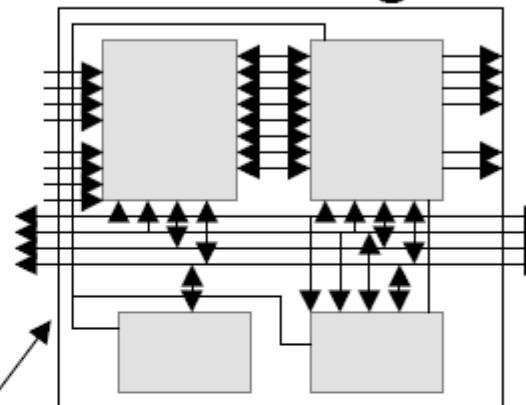


Interface Bus

Signal 1
Signal 2
Read()
Write()
Assert

Verilog-2001 style design

HDL Design



Complex signals

- Bus protocol repeated in blocks
- Hard to add signal through hierarchy

Communication encapsulated in interface

- Reduces errors - easier to modify
- Significant code reduction - saves time
- Enables efficient transaction modeling
- Allows automated block verification

What is an Interface?

- Provides a new hierarchical structure
 - Encapsulates interconnect and communication
 - Separates communication from functionality
 - Eliminates "wiring" errors
 - Enables abstraction in the RTL

```
int i;  
logic [7:0] a;  
  
interface intf;  
    int i;  
    logic [7:0] a;  
endinterface : intf
```

A simple interface
is a bundle of wires

```
int i;  
logic [7:0] a;  
  
typedef struct {  
    int i;  
    logic [7:0] a;  
} s_type;
```

Just like a simple struct
is a bundle of variables

Example without Interface

```

module memMod(input    logic req,
                  bit    clk,
                  logic start,
                  logic[1:0] mode,
                  logic[7:0] addr,
                  inout  logic[7:0] data,
                  output logic gnt,
                  logic rdy);

always @(posedge clk)
    gnt <= req & avail;
endmodule

```

```

module cpuMod(input    bit clk,
                  logic gnt,
                  logic rdy,
                  inout  logic [7:0] data,
                  output logic req,
                  logic start,
                  logic[7:0] addr,
                  logic[1:0] mode);

endmodule

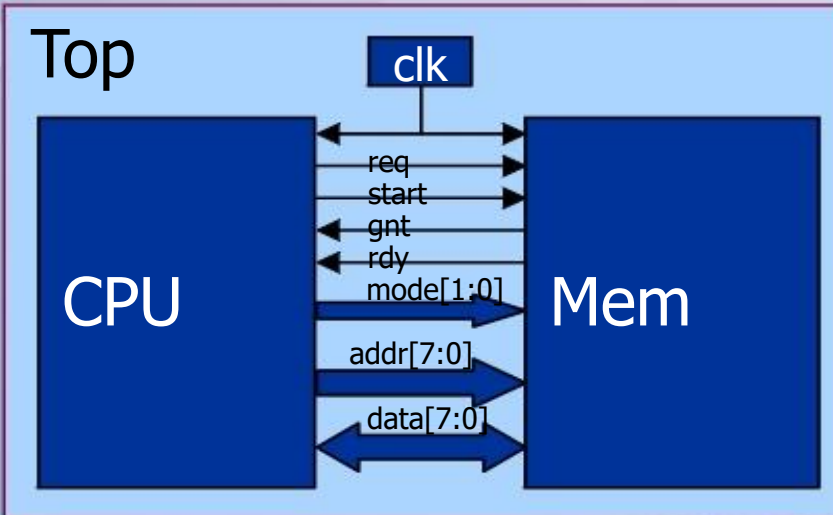
```

```

module top;
    logic req,gnt,start,rdy;
    bit    clk = 0;
    logic [1:0] mode;
    logic [7:0] addr,data;

    memMod mem(req,clk,start,mode,
               addr,data,gnt,rdy);
    cpuMod cpu(clk,gnt,rdy,data,
               req,start,addr,mode);
endmodule

```



Example Using Interfaces

```
interface simple_bus;
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
endinterface: simple_bus
```

Bundle signals
in interface

Use interface
keyword in port list

```
module memMod(interface a,
               input bit clk);
  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule
```

Refer to intf
signals

```
module cpuMod(interface b,
               input bit clk);
endmodule
```

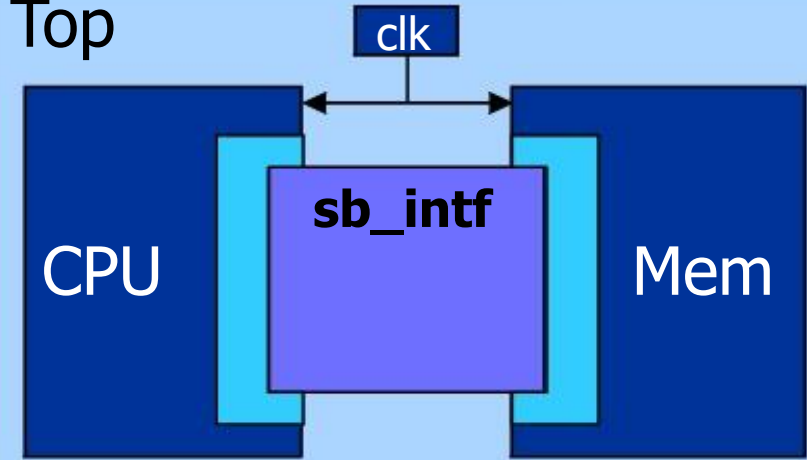
```
module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf),
             .clk(clk));
endmodule
```

interface
instance

Connect
interface

Top



Outline

- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1 (Functional viewpoints)
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法 2 (Language viewpoints)
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- SystemVerilog
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

GOTCHAS

Combinational Logic

- initial value for combinational logic will be ignored for synthesis
- No assign block (legal but not synthesizable)

硬體設計沒有起始值這種東西

```
logic a =0;
```

```
assign a=b+c;
```

WRONG!!!

```
always_comb begin  
    assign a = b&c;  
end
```

How to Properly Initialize a DFF?

- Reset DFF registers
 - You can reset DFF when rst_n triggers
 - **Do not use initial** begin since it is not synthesizable

```
always_ff@(posedge clk or negedge rst_n)
  if(rst_n == 0)
    a <= 0;
  else
    a <= b;
```



```
initial a = 0;
```

```
logic a = 0;
```

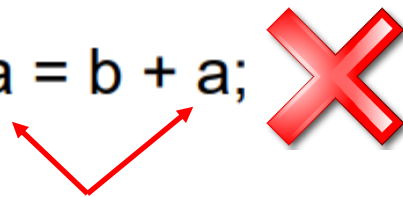
Bad for design, not synthesizable
Only for testbench simulation
Initial values are ignored for synthesis

Combinational Loop

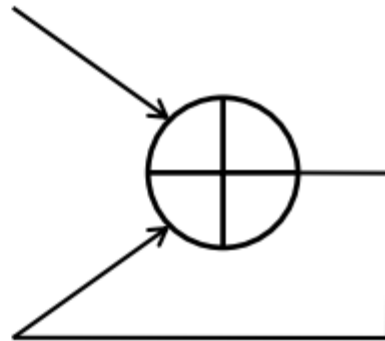
- A variable cannot be assigned by itself in combinational logic
 - Combinational loop (timing loop)

```
wire [3:0] a;  
wire [3:0] b;
```


```
assign a = b + a;
```




Same variable in both sides



```
always_comb  
a = b+a;
```



```
always_ff@(posedge clk)  
a <= b+a;
```



Same variable in both sides
only allowed in seq. logic

WRONG!!! Not synthesizable

Multiple Drivers

- Multiple driver
 - One variable can only be assigned in one always block or in one assign
 - This will cause **race conditions** (actual values depends simulator scheduling)
 - **Solution**: multiple source statements should put in the same always or assign

```
always_comb
  a = e+f;
```

```
always_comb
  a = 2;
```



Same for initial statement
and always_ff

```
assign a=c+d;
assign a=4;
```



```
Initial //at t=0
  a = 1;
```

```
Initial //at t=0
  a = 0;
```

```
always_comb begin
  a = e+f;
  a = 2;
end
```



```
logic a = 1;
```

Same variable used in two loops running simultaneously

```
module twoloops;

integer i;

initial begin // start at time 0
    for (i=0; i <= 7; i = i + 1) begin
        #5 $display("Entered 1st loop at t=%0d, i =
                    %0d", $time, i);
    end
end

initial begin // start at time 0
    for (i=0; i <= 7; i = i + 1) begin
        #2 $display("Entered 2nd loop at t=%0d, i =
                    %0d", $time, i);
    end
end
```

Index variable i in both loops will conflict

- Use **local variable** within for loop

```
module twoloops;
```

```
// integer i; // is now local within the for loops
```

```
initial begin // start at time 0
```

```
    for (int i=0; i <= 7; i = i + 1) begin
```

```
        #5 $display("Entered 1st loop at t=%0d, i =  
%0d", $time, i);
```

```
    end
```

```
end
```

```
initial begin // start at time 0
```

```
    for (int i=0; i <= 7; i = i + 1) begin
```

```
        #2 $display("Entered 2nd loop at t=%0d, i =  
%0d", $time, i);
```

```
    end
```

```
end
```

```
endmodule // twoloops
```

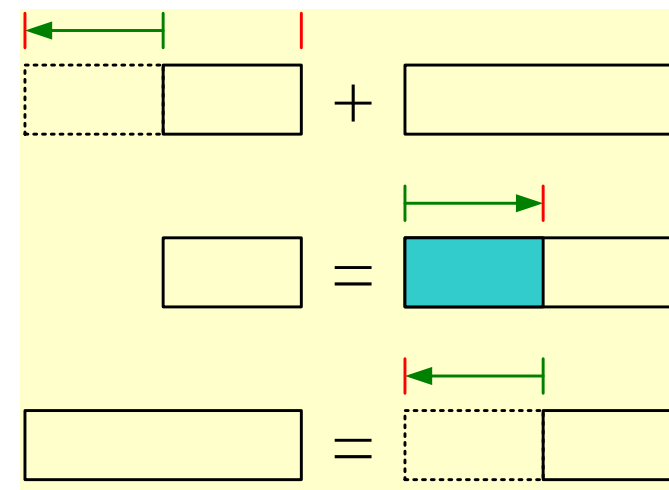

Sizing in Verilog

- Verilog automatically **resizes values** in an expression according to the sizes of variables in the expression.
- Verilog automatically **truncates or extends** the right-hand-side value in an assignment to fit the left-hand-side variable.

```

module sign_size;
reg [3:0] a, b;
reg [15:0] c;
initial
begin
  a = -1; // a is unsigned, so it stores "1111"
  b = 8;
  c = 8; // b = c = 1000
  #10 b = b + a; // result 10111 is truncated. b = 0111
  #10 c = c + a; // c = 00000000000010111
end
endmodule

```



Imcompleted case / if-else => unintentional latch

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
```

```
    reg out;
```

```
    always @( * )
```

```
    begin
```

```
        if ( sel == 2'd0 )
```

```
            out = a;
```

```
        else if ( sel == 2'd1 )
```

```
            out = b
```

```
        else if ( sel == 2'd2 )
```

```
            out = c
```

```
        else if ( sel == 2'd3 )
```

```
            out = d
```

```
        //else
```

```
        //  out = 1'bx;
```

```
    end
```

```
endmodule
```

Nested if-else

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
```

```
    reg out;
```

```
    always @( * )
```

```
    begin
```

```
        case ( sel )
```

```
            2'd0 : out = a;
```

```
            2'd1 : out = b;
```

```
            2'd2 : out = c;
```

```
            2'd3 : out = d;
```

```
            //default : out = 1'bx;
```

```
        endcase
```

```
    end
```

```
endmodule
```

case

How to Find Incomplete Case/if-else

- Code review
- By tools
 - Linting tools: nLint (renamed as Synopsys Spyglass Lint)
 - Synthesis tools: check “latch” keyword in the synthesis log
 - Compiler generates warning messages when inferring latches

SpyGlass(renamed from nLint) rule(1)

- Simulation
 - Combinational Loop
 - Infinite Loop
 - Signal in Sensitivity List Changed in the Block
 - Loss of Significant Bit
- Synthesis
 - Logic Expression Used in Sensitivity
 - Delay in Non-blocking Assignment
 - Blocking/Non-blocking Assignment in Edge-triggered Block
 - Inferred Latch
- ERC
 - Floating Net
 - Partial Input Floating
 - Output Floating
 - Input Floating
- DFT
 - Gated Clock
 - Buffered Clock
 - Reset Driven by Combinational Logic

nLint rule(2)

- Design Style
 - Two-process Style Not Used for FSM
 - Clock Driven by Sequential Logic
 - Clock Signal Used on Both Edges
 - No Set or Reset Signal
 - No Glue Logic Allowed in Top Module
- Language Construct
 - Bit Width Mismatch in Assignment
 - Multi-bit Expression when One Bit Expression is Expected
 - Bit Range Specified for Parameter
 - Bit Width Mismatch Between Module Port and Instance Port
- HDL Translation
 - Verilog/VHDL Reserved Words
 - Include Compiler Directive Used
- Naming Convention
 - Port Name Too Long
 - Clock Name Prefix or Suffix
 - Active Low Signal Name Prefix or Suffix
 - Port Name Does Not Follow the Connected Signal

nLint rule(3)

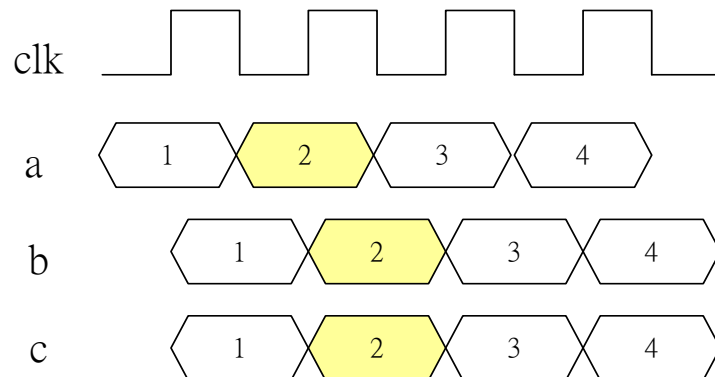
- Coding Style
 - Line Too Long
 - TAB Used in Indentation
 - More than One Statement per Line
 - Unconventional Port Declaration Order
- VITAL Compliant
 - For VHDL only
- Clock
 - Generated Reset
 - Generated Clock
 - Tri-state Buffer in a Clock Path
- Block Interconnect
 - Conflict of Hierarchy Interconnection
 - Block Assembly Error in the Same Hierarchy Level

Blocking vs. non-blocking assignment

- Blocking assignment (for combinational circuit)
 - The assignment will be carried consequently.
- Non-blocking assignment (for sequential circuit)
 - The assignment will be carried in parallel.

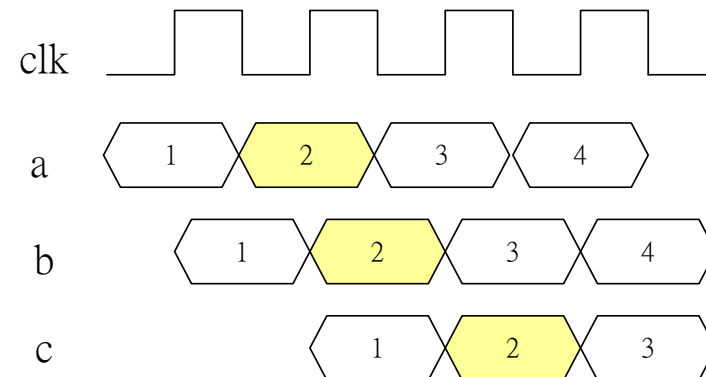
Blocking

```
always@( posedge clk )
begin
  b = a;
  c = b;
end
```



Non-blocking

```
always@( posedge clk )
begin
  b <= a;
  c <= b;
end
```

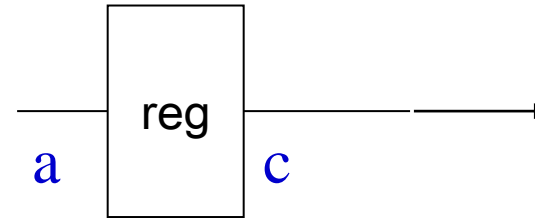


T=0
a=2,
b=1

Blocking vs. non-blocking assignment

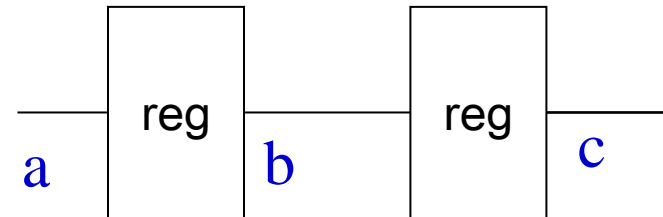
Blocking

```
always@( posedge clk )  
begin  
    b = a;  
    c = b;  
end
```



Non-blocking

```
always@( posedge clk )  
begin  
    b <= a;  
    c <= b;  
end
```



Value Swap

```
initial begin
```

```
    x = 5;
```

```
    y = 3;
```

```
end
```

```
always @(negedge clock) begin
```

```
    x = y;
```

```
    y = x;
```

```
end
```

This will give both x and y the same value.

Race condition: If the circuit was to be built a race condition has been entered which is unstable. The compiler will give a stable output, however this is not the output expected.

The simulator assigns x the value of 3 and then y is then assigned x. As x is now 3, y will not change its value.

```
always @(negedge clock) begin
```

```
    x <= y;
```

```
    y <= x;
```

```
end
```

both the values of x and y are stored first.
Then x is assigned the old value of y (3) and
y is then assigned the old value of x (5)

(Don't) Mix Blocking and NonBlocking

```

always @ (posedge clk) begin
    if(shiftIndex == 0) begin
        if(dataValid == 1) transmitting = 1; //Blocking assign
        else transmitting = 0; //Blocking assign
    end
    //only evaluated on positive clock edge

    if(transmitting == 1) begin
        shiftIndex <= shiftIndex + 1;
        dataOut <= data5b[shiftIndex];

        if(shiftIndex == 4) begin
            complete <= 1;
            shiftIndex <= 0;
        end
    end
    else begin
        complete <= 0;
    end
end
end

```



```

always @* begin
    if(shiftIndex == 0) begin
        if(dataValid == 1) transmitting = 1; //Blocking assign
        else transmitting = 0; //Blocking assign
    end
end

always @ (posedge clk) begin
    if(transmitting == 1) begin
        shiftIndex <= shiftIndex + 1;
        dataOut <= data5b[shiftIndex];

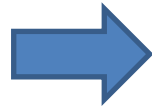
        if(shiftIndex == 4) begin
            complete <= 1;
            shiftIndex <= 0;
        end
    end
    else begin
        complete <= 0;
    end
end
end

```



```
always @(posedge clk or negedge reset) begin
    if (~ reset)
        q = 1'b0;
    else
        q <= d;
end
```

```
always @(posedge clk)
    x = y;
always @(posedge clk)
    y = z;
```



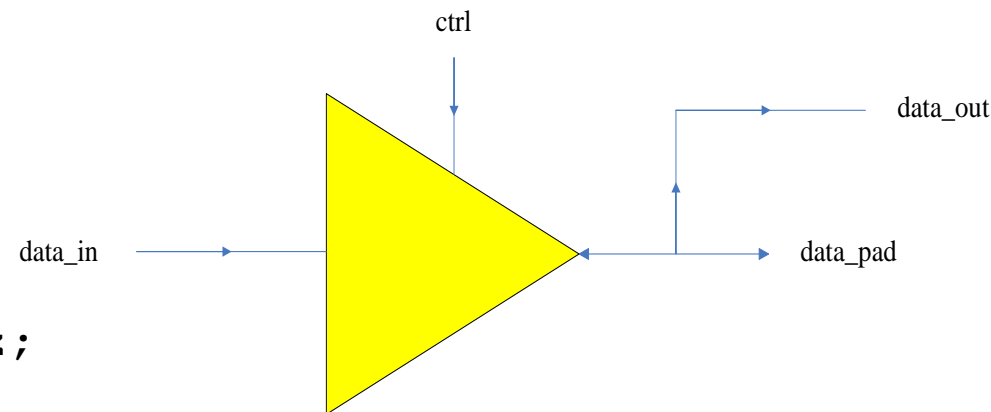
```
always @(posedge clk)
    x <= y;
always @(posedge clk)
    y <= x;
```

Race Condition Using Blocking Assignments

Tri-State Logic

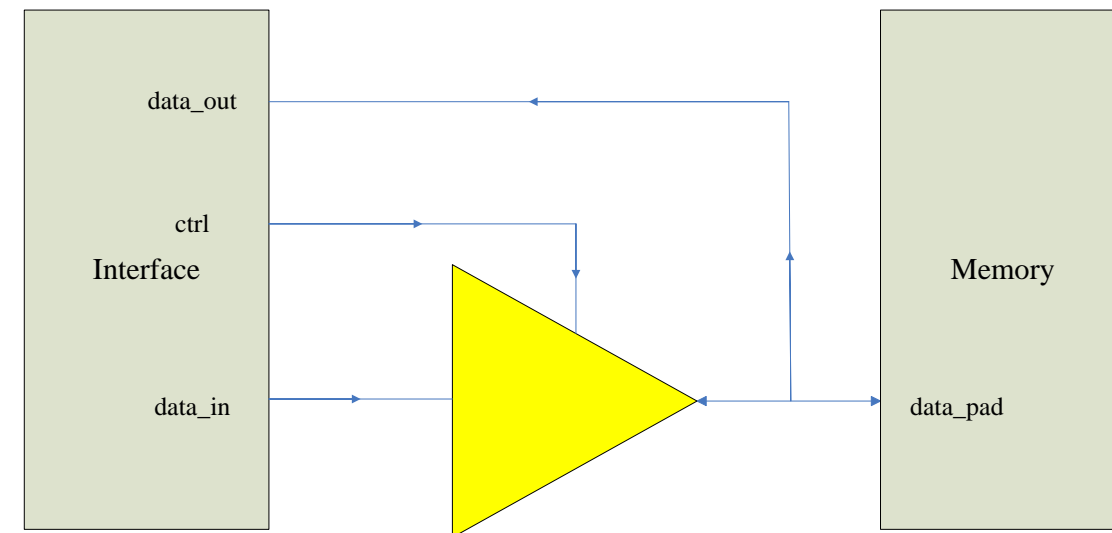
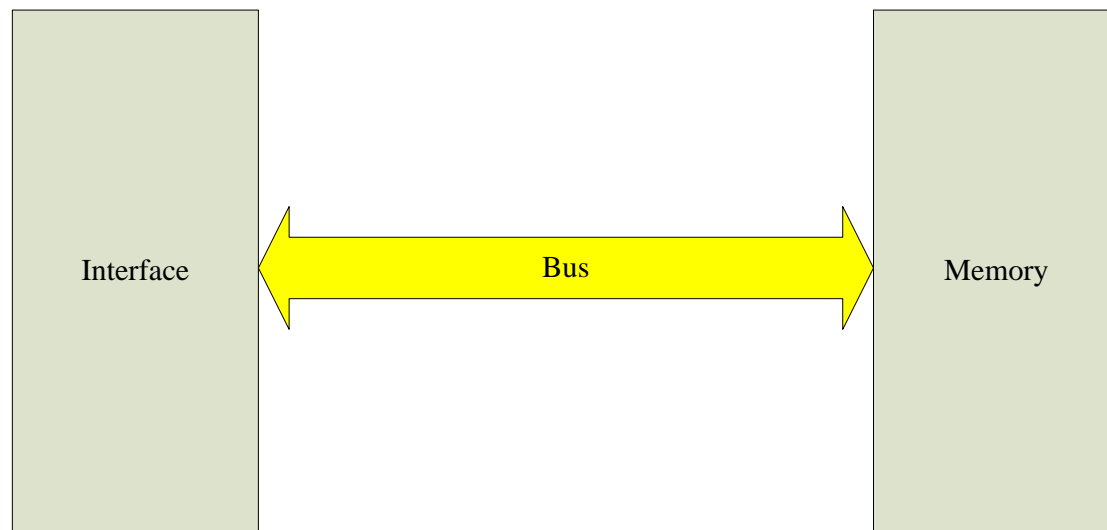
- If possible, use multiplexors to replace tri-state logic
 - Used for bidirectional I/O pad
- How to write tri-state logic?

```
// module declaration
module tri_state_buffer( data_in, data_out, data_pad, ctrl);
// port declaration
input  [15:0] data_in;
input          ctrl;
output [15:0] data_out;
inout  [15:0] data_pad; //bidirectional port
// structure
assign      data_pad = (ctrl)? data_in : 16'hz;
assign      data_out = data_pad;
endmodule
```



Tri-State Logic

- Use in on-chip bus or I/O pad



Quick Glimpse of Synthesizable Subset

- Not all of the Verilog commands can be synthesized into hardware

always	begin end	case endcase	casez endcase
if else	module endmodule	input output	function
reg Wire logic	default	posedge negedge	parameter
or	assign	for	

References

- Verilog HDL – Samir Palnitkar
- The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and Build Gates – Clifford E. Cummings
- Verilog: Frequently Asked Questions: Language, Applications and Extensions - Shivakumar S. Chonnad, Needamangalam B. Balachander

References for SystemVerilog

- **Books**

- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features by Chris Spear (Hardcover - Jul 10, 2006)
- SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling by Stuart Sutherland, Simon Davidmann, Peter Flake, and P. Moorby (Hardcover - Jul 20, 2006)

- **Web site**

- Official site: www.systemverilog.org
- On-line tutorial: <http://www.doulos.com/knowhow/sysverilog/>

- **Course**

- MIT 6.375 Complex Digital Systems

Reference

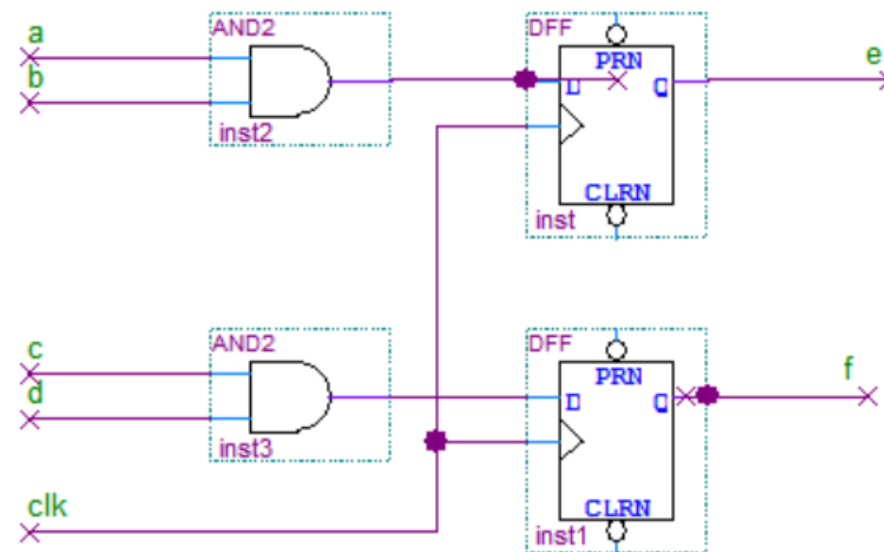
- <https://hom-wang.gitbooks.io/verilog-hdl/content/>
- <http://www.asic-world.com/verilog/index.html>
- <https://www.slideshare.net/itembedded/verilog-14596615>
- http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf
- <http://www-inst.eecs.berkeley.edu/~cs150/fa13/agenda/>
- http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/

Note. 由C語言學習Verilog的思維轉換

- 軟體是循序的，而硬體是並行的

```
1 always@(posedge clk) begin
2   e <= a & b;
3   f <= c & d;
4 end
```

- 硬體要循序，要靠clock和FSM
 - 靠clock並且搭配FSM，當一個state完成後，進入下一個state，這樣就能依照clock的進行，而達成循序的要求
- Verilog程式碼沒有先後之分
 - blocking assignment有先後執行順序，而非blocking assignment同時執行
- 硬體『描述』語言，而非硬體『程式』語言
 - 先寫的不代表先執行，後寫的也不代表後執行，只是代表硬體的架構的描述，也就是說，將原來的電路圖，變成文字描述而已



Verilog vs C

- Verilog 有些語法像C，但不是軟體語言
 - No pointer, no return function, no recursive calls, no memory allocation
- Verilog \Leftrightarrow Hardware module
 - 了解每一行Verilog code 會產生的硬體架構
 - 知道想做的硬體架構應該對應的Verilog code 寫法 (Coding style)
- 何時可以用很像C的高階寫法
 - 在 testbench 時使用

幫幫EDA tools: Helping the Tools*

為什麼要建議某種寫法？

In an ideal world it shouldn't matter how you write the Verilog – optimization in the CAD tools will find the best solution. But the world is not ideal (yet...)

- Tools work best on smaller problems
 - Need to partition real problem into pieces for you and the tools (it's hard to think about 1M gates at one time). Decompose large problems into smaller problems and then connect the solutions
 - Hierarchy in Verilog " partitions in physical layout
- Tools use your code as a starting point.
 - Your structure isn't completely eliminated (this is good...)
 - Little optimization will be done between top-level blocks
- **Structure of the problem** is often important
 - Finding a “good” way to think about the problem is key

請用不讓Tool誤解的寫法



Like optimizing compilers for C, tools are good for local optimizations but don't expect them to rewrite your code and change your algorithm. With practice you'll learn what works and what doesn't...

*adapted from a Stanford EE271 lecture by Mark Horowitz

Simulation and Synthesis

- Not all of the Verilog commands can be synthesized into hardware
- Our primary interest is to build hardware, we will emphasize a synthesizable subset of the language
- Will divide HDL code into synthesizable modules and a test bench (simulation).
 - The **synthesizable** modules describe the hardware.
 - The **test bench** checks whether the output results are correct (only for simulation and cannot be synthesized)

