100-1 Under-Graduate Project: RTL Coding Style

Speaker:蘇冠羽

Date: 2011/12/06

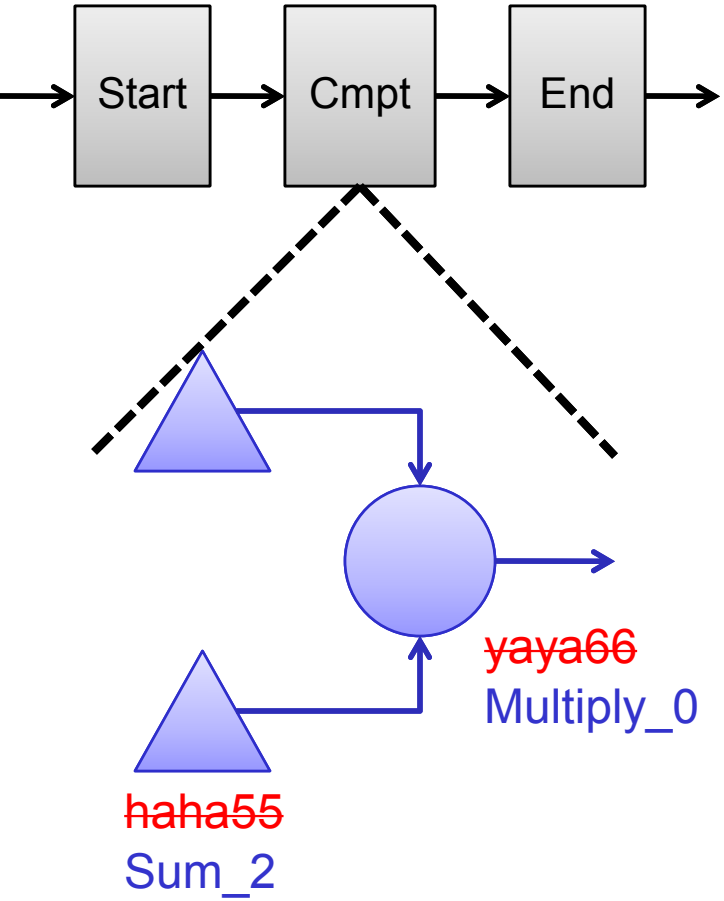
Outline

- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

Pre-RTL Preparation Checklist

- Communicate design issues with your teammates
  - Naming conventions, directory trees and other design organizations
- Have a specification for your design
  - Everyone should have a specification BEFORE they start coding
- Design partition
  - Follow the specification's recommendations for partition
  - Break the design into major functional blocks

RTL Coding Style



- Create a block level drawing of your design before you begin coding.
  - Draw a block diagram of the functions and sub-functions of your design.
- Hierarchy design
- Always think of the poor guy who has to read your RTL code.
  - Easy to understand.
  - Meaningful names.
  - Comments and headers.

Outline

- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

File Headers

- Include informational header at the top of every source file, including scripts.
  - Filename
  - Author information, e.g. name, email…
  - Description of function and list of key features of the module
  - Available parameters
  - Reset scheme and clock domain
  - Date the file was created and modified
  - Critical timing and asynchronous interface

File Header Example: DCT.v

```
1  // ======================================================================
2  // COPYRIGHT (c) 2007 Chin-Yu Chen. ALL RIGHTS RESERVED
3  // ----------------------------------------------------------------------
4  // FILE NAME      : dct.v
5  // TYPE           : top module
6  // DEPARTMENT     : Access Lab, Graduate Institute of Electronics Engineering, National Taiwan University
7  // AUTHOR         : Chin-Yu Chen
8  // AUTHOR'S EMAIL : doow@access.ee.ntu.edu.tw
9  // PURPOSE        : A real-time discrete cosine transform(DCT) implemetation
10 // ----------------------------------------------------------------------
11 // ======================================================================
12 // REVISION HISTORY
13 //
14 //        DATE        VERSION       AUTHOR          COMMENTS
15 //        ----------------------------------------------------------------
16 //        2007/05/30  1.0           Chin-Yu Chen    First creation
17 // ======================================================================
18 // PARAMETERS
19 //
20 //        PARA_NAME       RANGE      COMMENTS          DEFAULT
21 //        ----------------------------------------------------------------
22 //        DATA_WIDTH      [8,16]     Width of data       8
23 // ======================================================================
24 // REUSE ISSUES
25 //
26 // RESET STRATEGY   : asynchronous reset
27 // CLOCK DOMAINS    : posedge trigger clock
28 //======================================================================
```

Identifiers Naming Rule

- Begin with an alpha character (a-z, A-Z) or an underscore (_) and can contain alphanumeric, dollar signs ($) and underscore.
  - Examples of illegal identifiers:
    - 34net
    - a*b_net
    - n@238
- Up to 1023 characters long
- Case sensitive
  - e.g. sel and SEL are different identifiers

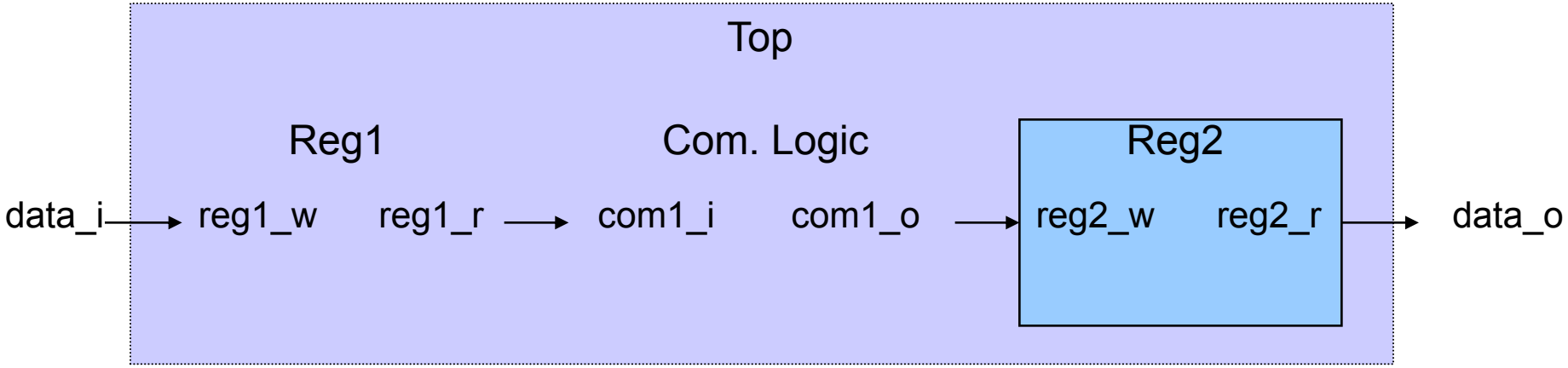
General Naming Conventions(1/3)

- Lowercase letters for all signals, variables, and port names.
  - reg is used in procedural block
- Uppercase letters for constants and user-defined types.
  - e.g. `define MEM_WIDTH 16
- Verilog is case sensitive
- Meaningful names
  - Use *ram_addr* for RAM address bus instead of *ra*

General Naming Conventions(2/3)

- Use *clk* for the clock signal
  - If more than one clock, use *clk* as the prefix for all clock signals (*clk1, clk2, clk_interface*)
- For active low signals, use ***_n**
  - If the reset signal is active low, use *rst_n*
  - *Similarly, for* active high signals, use ***_p**
- For input of a register, use ***_w**
- For output of a register, use ***_r**
- For input signals, use ***_i**
- For output signals, use ***_o**

Example




```
assign reg1_w = data_i;
assign reg2_w = com1_o;
 assign com1_i = reg1_r;

always @ (posedge clk_p) begin
                reg1_r <= reg1_w;
                 reg2_r <= reg2_w;
end

always @(*) begin
                com1_o = com1_i + 4'd5;
end
```

For connection

Logic

General Naming Conventions(3/3)

- Use *[x:0]* (instead of [0:x]) when describing multi-bit buses
  - A somewhat arbitrary suggested “standard”
- Use parameter to improve readability
  - module car (out, in1, in2);

    ……

    parameter S0_STOP = 2’d0, S1_RUN = 2’d1;

    ……

    case (state) S0_STOP:

    …..
- Don’t use HDL reserved words
  - e.g. xor, nand , module

Use comments

- Use comments appropriately to explain
  - Brief, concise, explanatory
  - Avoid “comment clutter”– obvious functionality does not need to be commented
- Single-line comments begin with //
- Multiple-line comments start with /* and end with */
- Use indentation to improve the readability of continued code lines and nested loops
  - e.g.

```
if(a)
    if(b)
        if(c)
```

Module Instantiation

- Always use explicit mapping for ports, use named mapping rather than positional mapping

```
DW_ram_r_w_s_dff
  #((`ram_data_width+`ram_be_data_width),
    (`fifo_depth),1)
    U_int_txf_ram (
      .clk          (refclk),
      .rst_n        (txfifo_ram_reset_n),
      .cs_n         (1'b0),
      .wr_n         (txfifo_wr_en_n),
      .rd_addr      (txfifo_rd_addr),
      .wr_addr      (txfifo_wr_addr),
      .data_in      (txfifo_wr_data),
      .data_out     (txf_ram_data_out)
    );
```

```
module_a (
    clk,
    s1_i,
    s1_o,
    s2_i,
    s2_o
);
```

Use loops and arrays

- Using loop to increase readability
  - Loop is usually used as memory initialization

    for example:

```
module my_module( ... );
...
reg [31:0] reg_file[15:0];
integer tmp;

always @(posedge clk or posedge rst)
If(rst)
for(tmp=0; tmp<16; tmp++)
        reg_file[tmp] <= 32'd0;
...
```

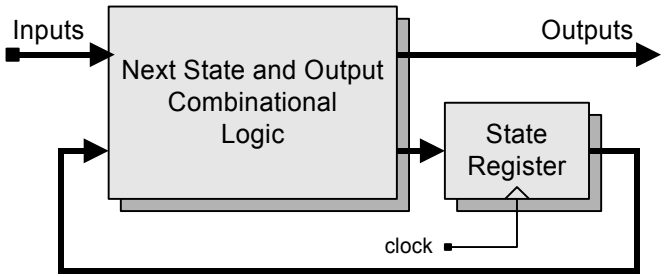
Outline

- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

Finite State Machines

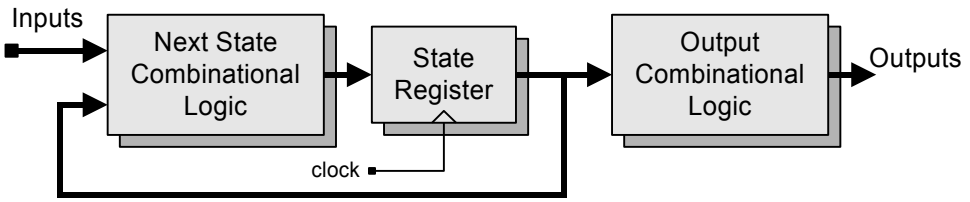
- FSM have widespread application in digital systems.
  - Most frequently used in controller
- Mealy Machine: The next state and the outputs depend on the present state and the inputs.
- Moore Machine: The next state depends on the present state and the inputs, but the output depends on only the present state.

Mealy machine



output = f (In, CS)

Moore machine



output = f (CS)

Modeling FSM in Verilog

- Sequential Circuits
  - Memory elements of States (S)
- Combinational Circuits
  - Next-state Logic (NL)
  - Output Logic (OL)
- Three coding styles
  - (1) Separate S, OL and NL
  - (2) Combines NL+ OL, separate S
  - (3) Combine S + NL, separate OL □ **Not recommended!!**
    Mix the comb. and seq. circuits

Style (1) Separate S, NL, OL

- **S**

```
always @ (posedge clk)
        current_state <= next_state;
```

- **NL**

```
always @ (current_state or In)
  case (current_state)
    S0: case (In)
              In0:  next_state = S1;
              In1:  next_state = S0;
              . . .
         endcase //In
    S1: . . .
    S2: . . .
 endcase //current_state
```

- **OL**

```
// if Moore
always @ (current_state)
  case(current_state)
  S0: z = output_value;
```

```
// if Mealy
always @ (current_state or In)
 case(current_state)
 S0: if(In==0) z = output_value;
       else …
```

Style (2) Combine NL+OL; Separate S

- **S**

```
always @ (posedge clk)
        current_state <= next_state;
```

- **NL+OL**

```
always @ (current_state or In)
  case (current_state)
    S0: beging
          case (In)
            In0:  begin
                    next_state<= S1;
                    Z =values; // Mealy
                  end
            In1:  . . .
          endcase // In
          Z =values; // Moore
        end //S0
    S1: . . .
 endcase // current_state
```

Outline

- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

Reset Signal

- Use the reset signal to initialize registered signals

```
// process with asynchronous reset
always @(posedge clk or posedge rst_a)
  begin : ex5-20_proc
    if  (rst_a == 1'b1)
      begin
        ...   dct_r <= 1'b0;
      end
    else begin
       ---    dct_r <= dct_w;
    end
  end // ex5-20_proc
```

Avoid Latches (1/2)

- Avoid using any latches in your design
- Use a design checking tool (nLint) to check for latches in your design
- Poor Coding Styles
  - Latch inferred because of missing else condition

```
always @(a or b)
  begin
    if (a == 1'b1)
      q <= b;
  end
```

- Latches inferred because of missing assignments and missing condition

```
always @(d)
  begin
    case (d)
      2'b00: z <= 1'b1;
      2'b01: z <= 1'b0;
      2'b10: z <= 1'b1; s <= 1'b1;
    endcase
  end
```

Avoid Latches (2/2)

- Avoid inferred latches
  - Fully assign outputs for all input conditions

Poor coding style:

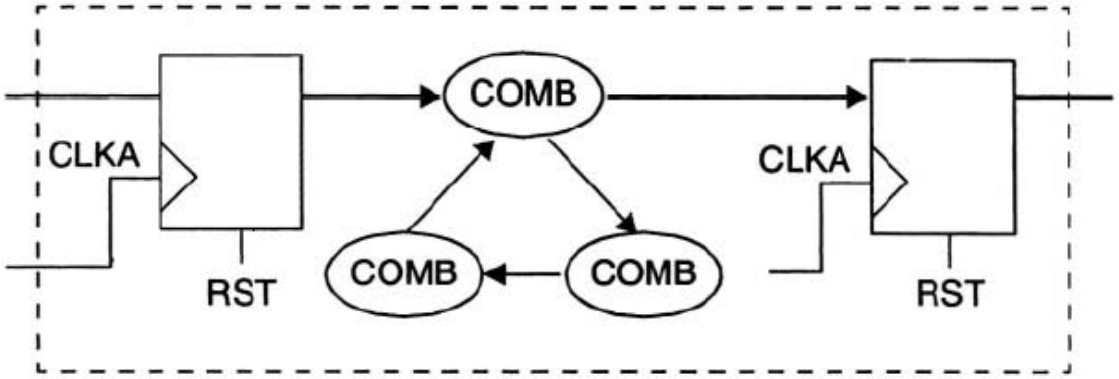
```
always @(g or a or b)
  begin
    if (g == 1'b1)
      q <= 0;
    else if (a == 1'b1)
      q <= b;
  end
```

Recommended coding style:

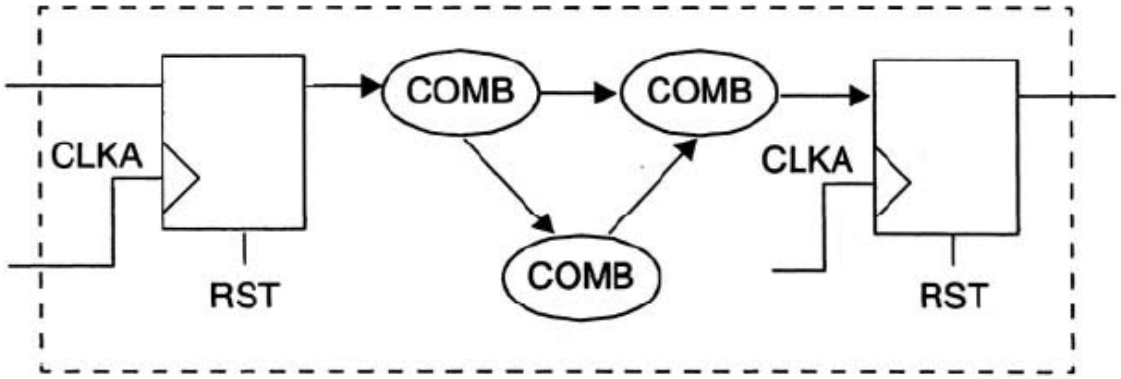
```
always @(g1 or g2 or a or b)
  begin
    q <= 1'b0 ;
    if (g1 == 1'b1)
      q <= a;
    else if (g2 == 1'b1)
      q <= b;
  end
```

Avoid Combinational Feedback

Bad: Combinational processes are looped



Good: Combinational processes are not looped



Sensitivity List (1/3)

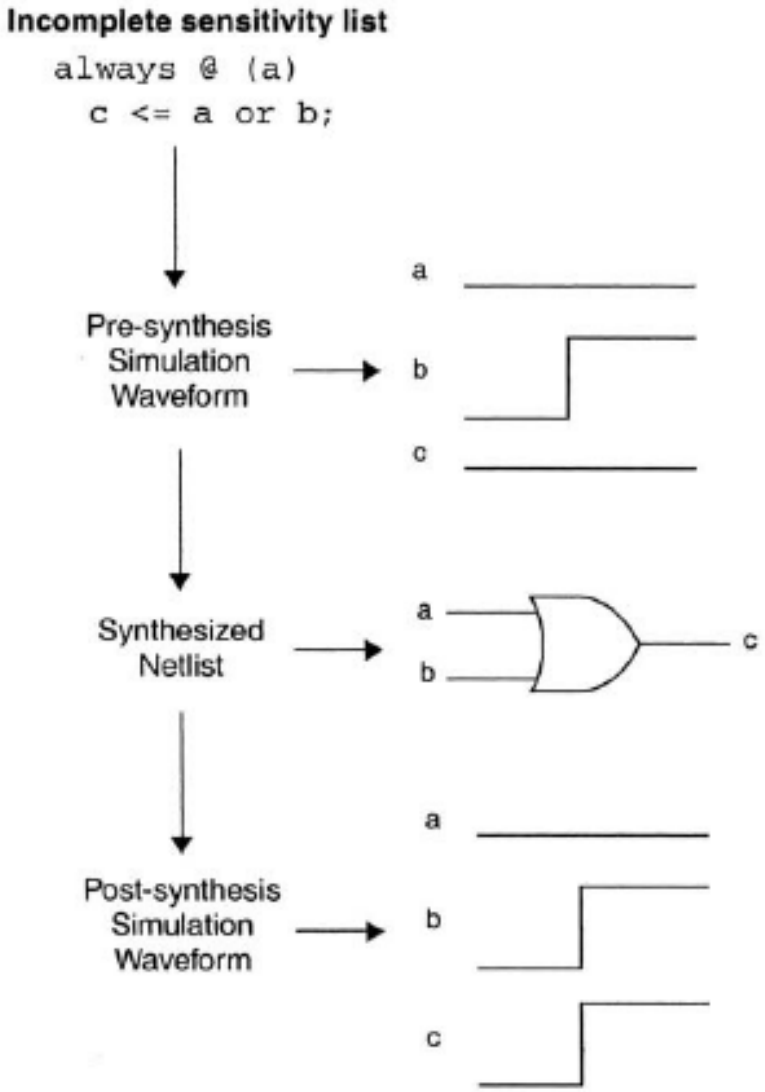
- For combinational blocks, the sensitivity list must include every signal that is read by the process.
  - Signals that appear on the right side of an assign statement
  - Signals that appear in a conditional expression

```
always @(a or inc_dec)
begin : COMBINATIONAL_PROC
  if (inc_dec == 0)
    sum = a + 1;
  else
    sum = a - 1;
end  // COMBINATIONAL_PROC
```

  - For simplicity, Verilog 2001 supports ***always @ (*)***

Sensitivity List (2/3)

- Include a complete sensitivity list in each of *always* blocks
  - If not, the behavior of the pre-synthesis design may differ from that of the post-synthesis netlist.



Sensitivity List (3/3)

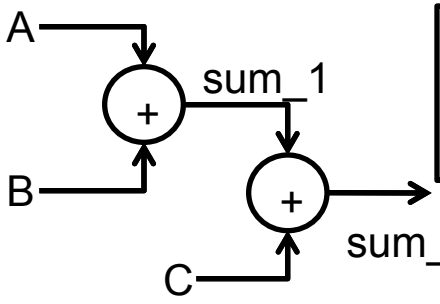
- For sequential blocks
  - The sensitive list must include the clock signal.
  - If an asynchronous reset signal is used, include reset in the sensitivity list.

```
always @(posedge clk)
begin : SEQUENTIAL_PROC
  q <= d;
end  // SEQUENTIAL_PROC
```

- Use only necessary signals in the sensitivity lists
  - Unnecessary signals in the sensitivity list slow down simulation

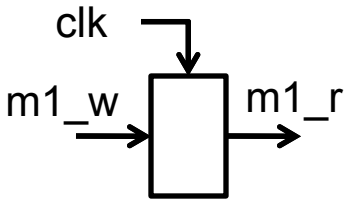
Combinational vs. Sequential Blocks

- **Combinational logic**
  - Use blocking (=) assignments
  - Execute in sequential order



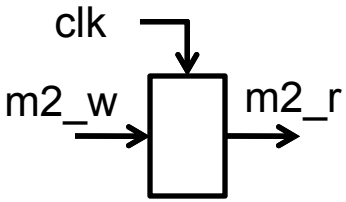

```
always@(*) begin
        sum_1 = A + B;
        sum_2 = sum_1 + C;
end
```

- **Sequential logic**
  - Use nonblocking (<=) assignments
  - Execute concurrently



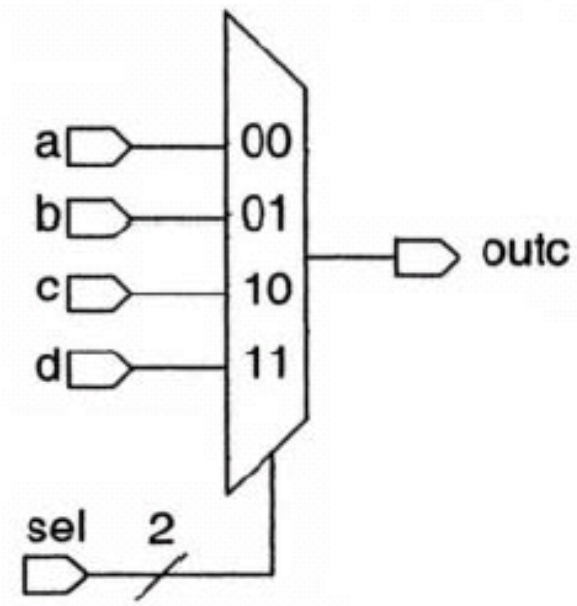

```
always@(posedge clk) begin
    m1_r <= m1_w;
    m2_r <= m2_w;
end
```

- Do not make assignments to the same variable from more than one *always* block.
  Multiple Assignment



case Statement

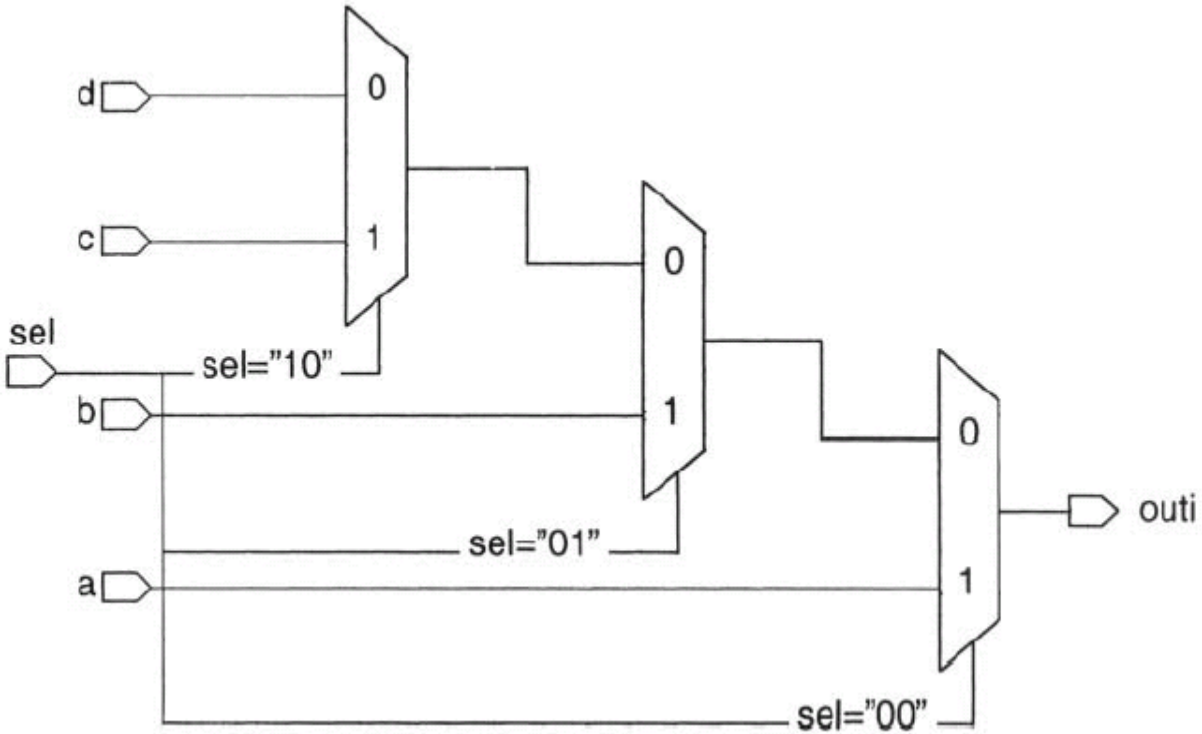
- Fully specified verilog *case* statements result in a single-level multiplexer
  - Partially specified Verilog *case* statements result in latches




```
case (sel)
   2'b00: outc = a;
   2'b01: outc = b;
   2'b10: outc = c;
   default: outc = d;
endcase;
```

if-then-else Statement

- An *if-then-else* statement infers a priority-encoded, cascaded combination of multiplexers.




```
if (sel == 2'b00)
   outi = a;
else if (sel = 2'b01)
   outi = b;
else if (sel = 2'b10)
   outi = c;
else
   outi = d;
```

case vs. *if-then-else* Statements

- *case* statements are preferred if the priority-encoding structure is not required
  - The multiplexer is faster.
- *if-then-else* statement can be useful if you have a late-arriving signal
  - Connect the signal to *a* in last slide
- A conditional assignment may also be used to infer a multiplexer.

```
assign z = (sel_a) ? a : b;
```

Outline

- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

Register All Outputs

- For each subblock of a hierarchical macro design, register all output signals from the subblock.
  - Makes output drive strengths and input delays predictable

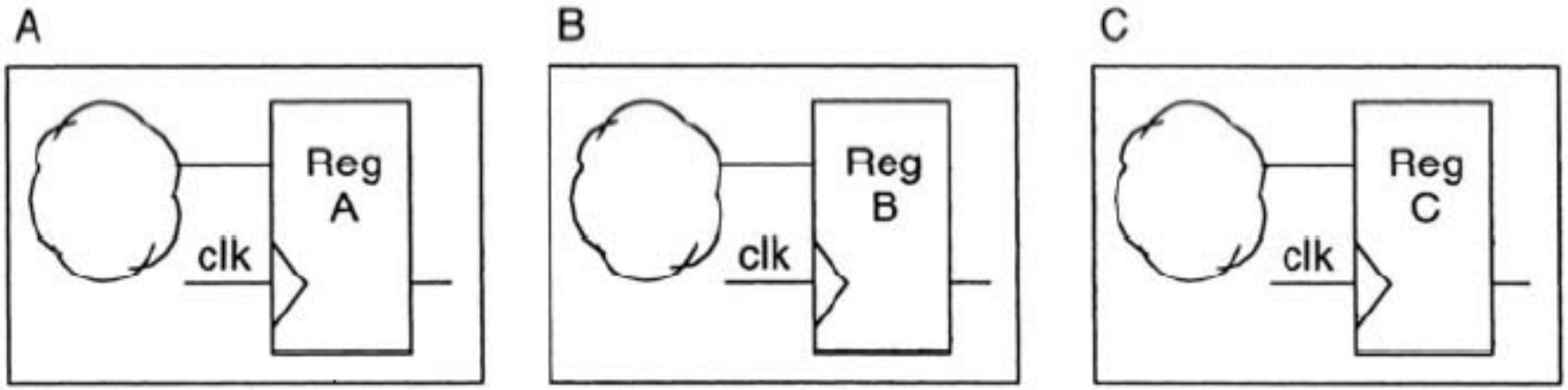
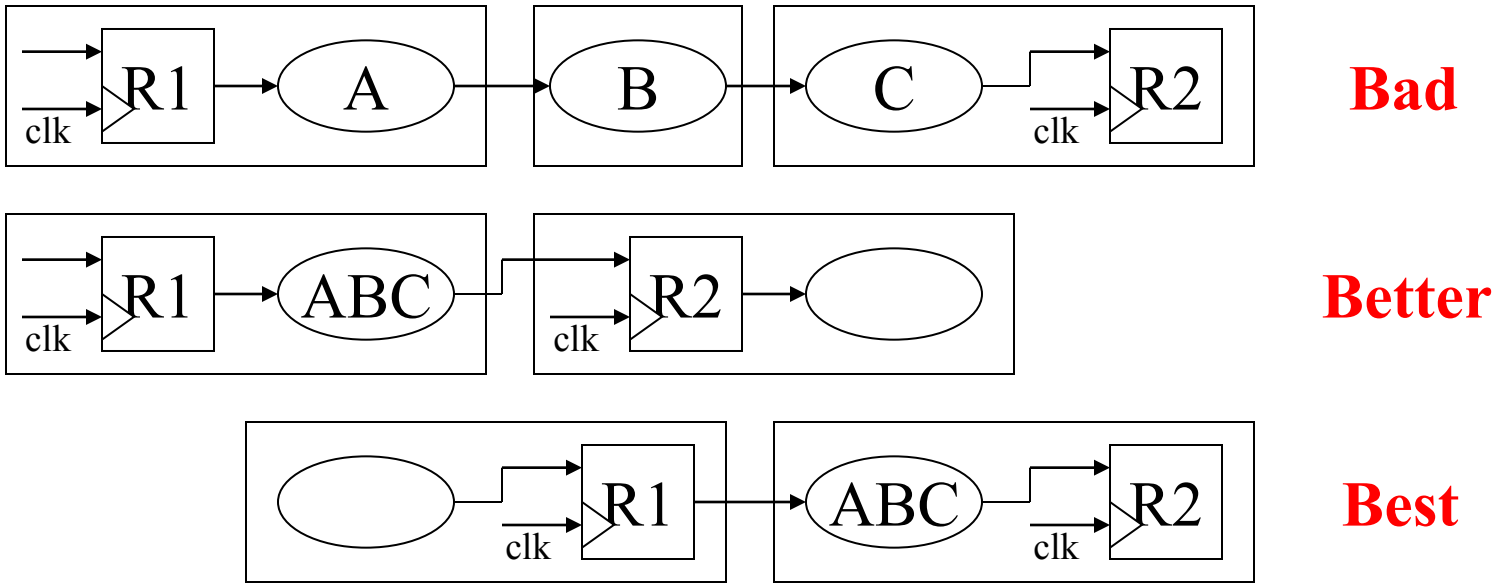


Figure Good example: All output signals are registered

Related Combinational Logic in a Single Module

- Keep related combinational logic together in the same module

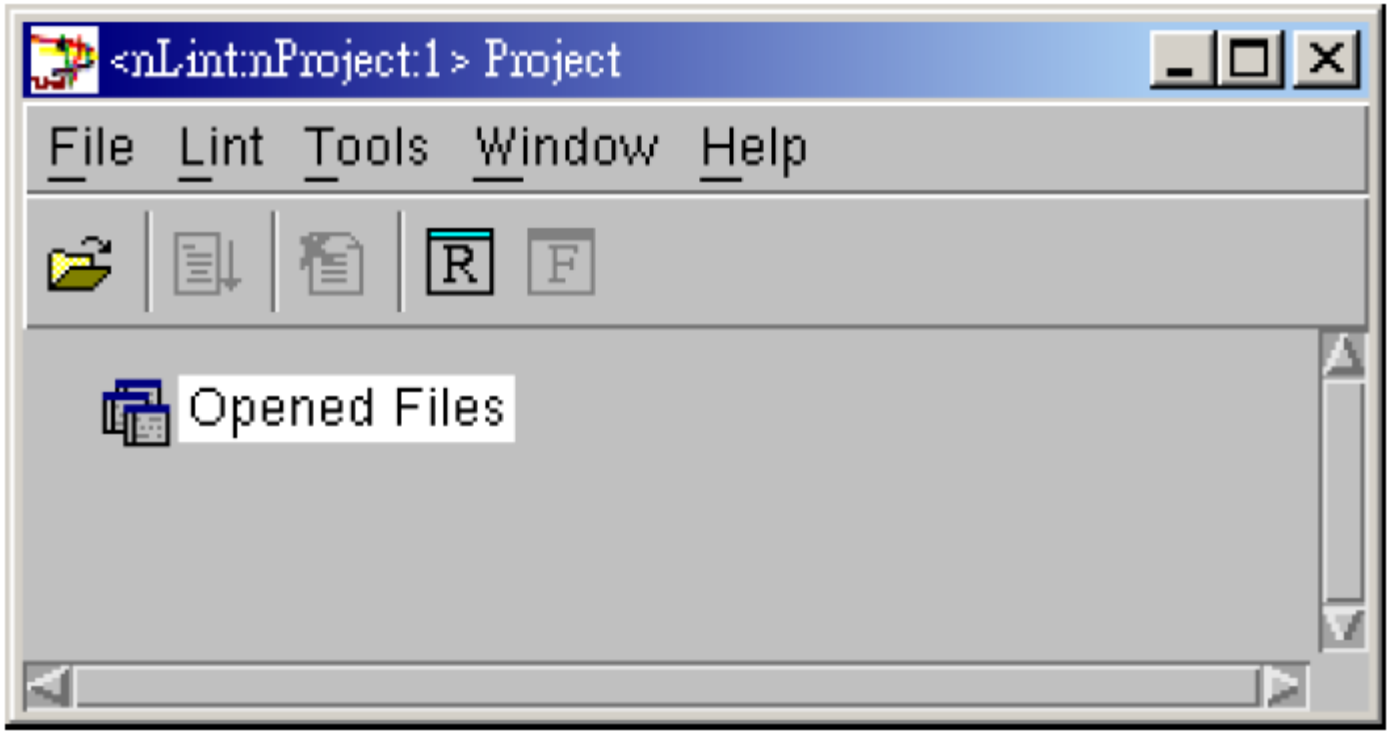


Outline

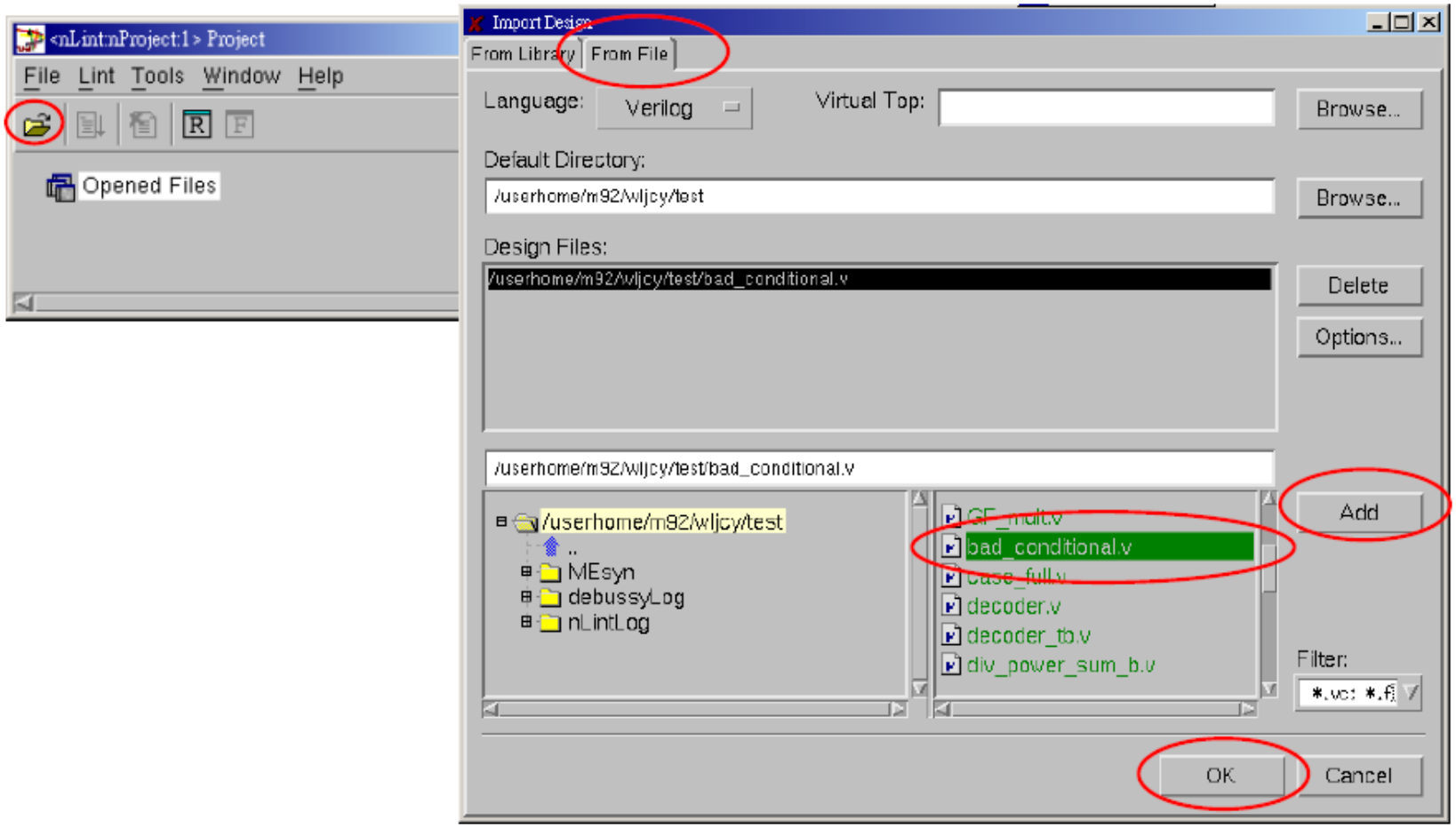
- Principles of RTL Coding Styles
  - Readability
  - Finite state machine (FSM)
  - Coding for synthesis
  - Partitioning for synthesis
- Debugging Tool: nLint

GUI

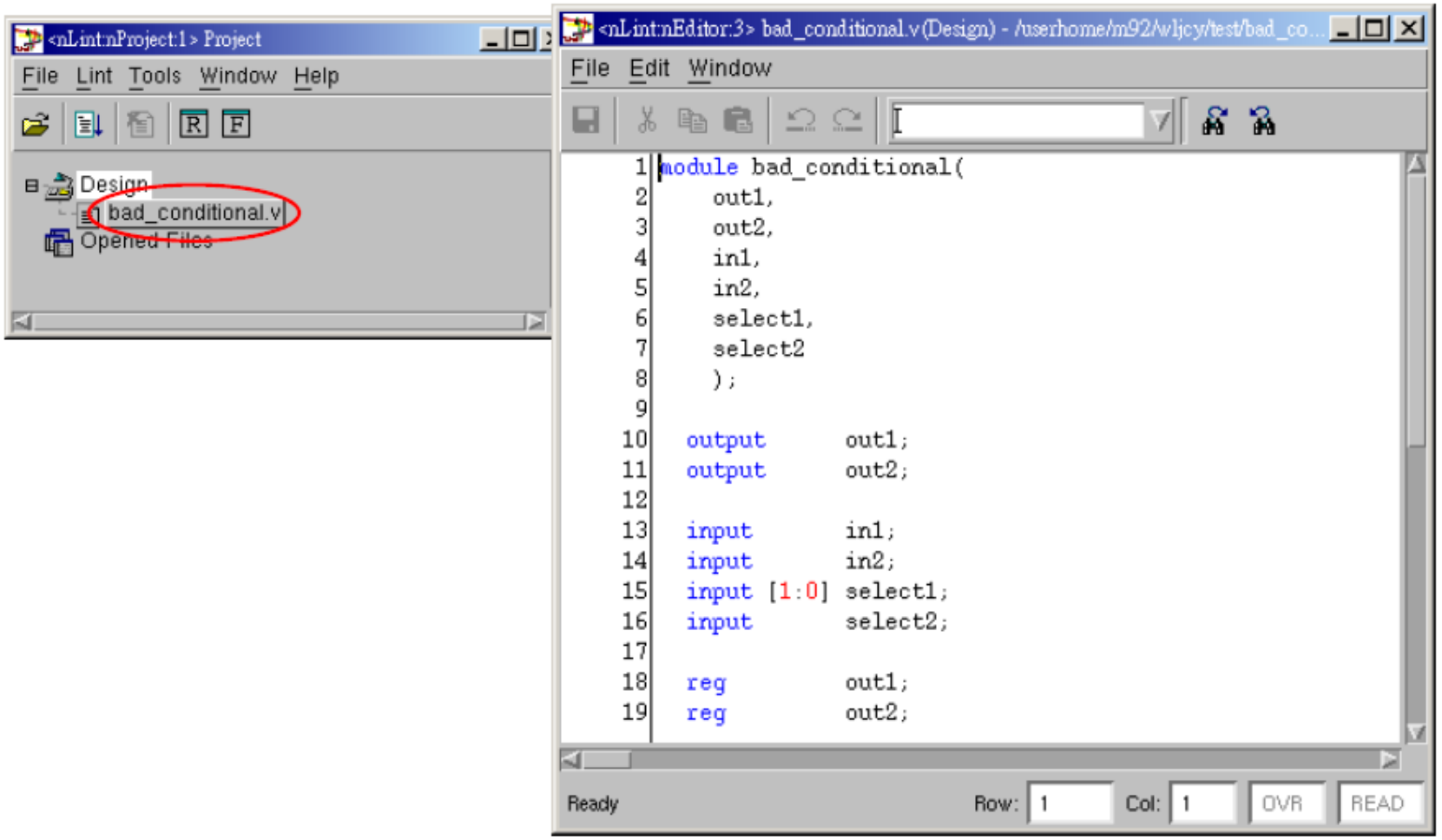
- nLint -gui &



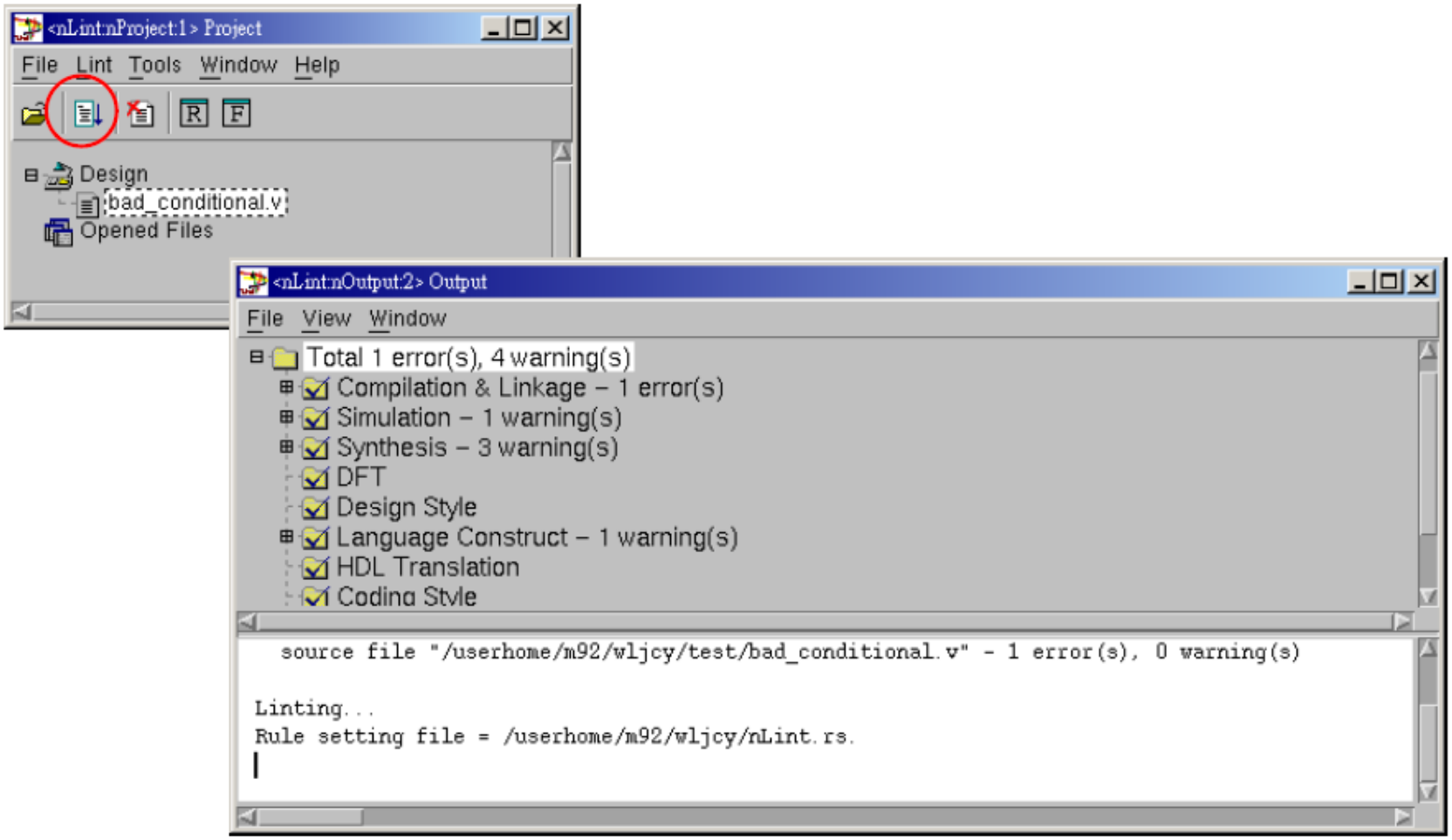
Import Design



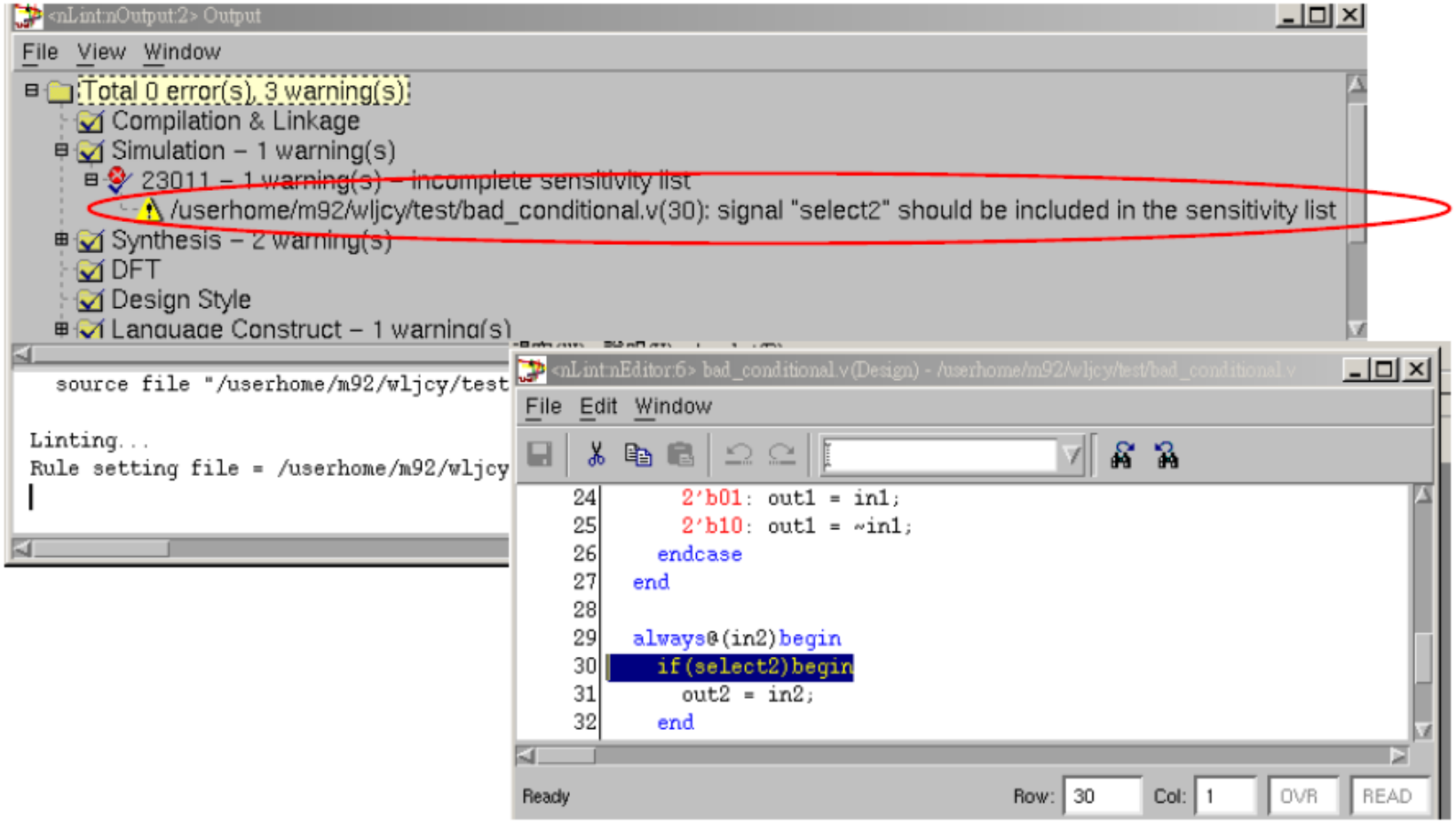
Edit File


<nLint:nProject:1> Project
File Lint Tools Window Help
Design
bad_conditional.v
Opened Files
<nLint:nEditor:3> bad_conditional.v(Design) - /userhome/m92/vljcy/test/bad_co...
File Edit Window
1 module bad_conditional(
2 out1,
3 out2,
4 in1,
5 in2,
6 select1,
7 select2
8);
9
10 output out1;
11 output out2;
12
13 input in1;
14 input in2;
15 input [1:0] select1;
16 input select2;
17
18 reg out1;
19 reg out2;
Ready
Row: 1
Col: 1
OVR
READ

Lint -> Run

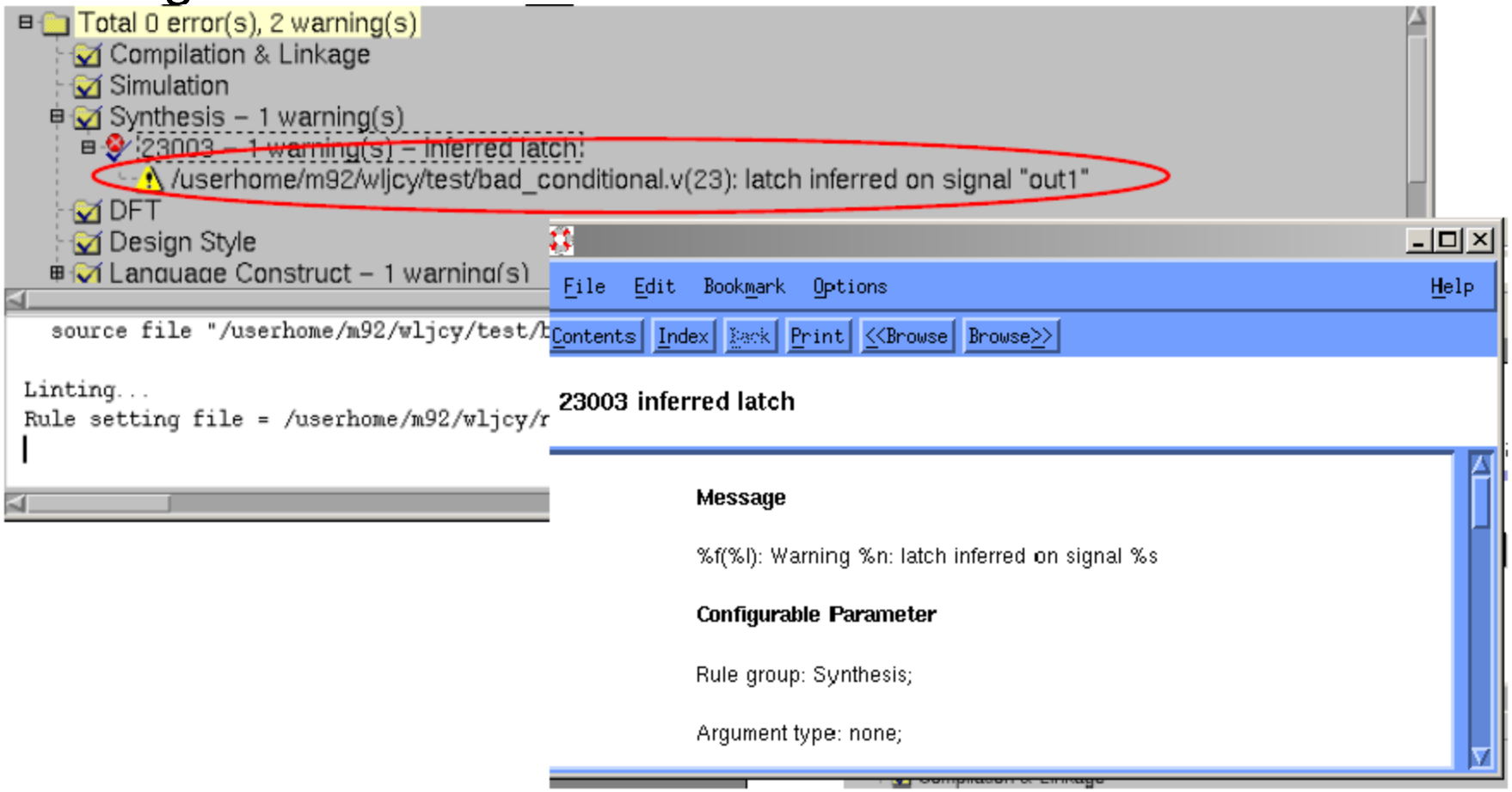

<nLint:nProject:1> Project
File Lint Tools Window Help
Design
bad_conditional.v
Opened Files
<nLint:nOutput:2> Output
File View Window
Total 1 error(s), 4 warning(s)
Compilation & Linkage – 1 error(s)
Simulation – 1 warning(s)
Synthesis – 3 warning(s)
DFT
Design Style
Language Construct – 1 warning(s)
HDL Translation
Coding Style
source file "/userhome/m92/wljcy/test/bad_conditional.v" - 1 error(s), 0 warning(s)
Linting...
Rule setting file = /userhome/m92/wljcy/nLint.rs.

Fix Warning 1

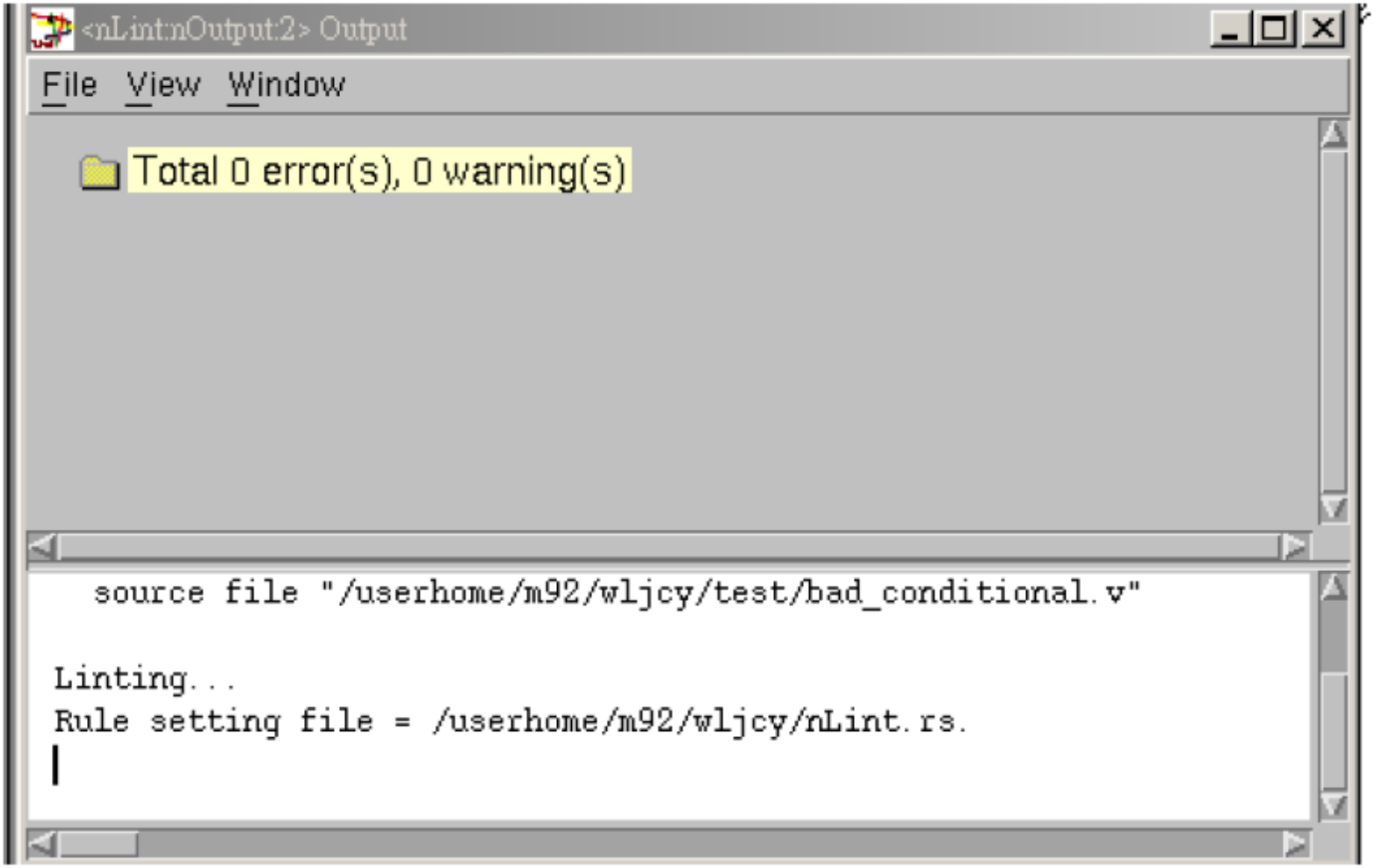


Search Rule

- Right click -> Search Rule



No Error & Warning



Check for Synthesizable (1/2)

- *SpringSoft nLint*
  - Check for correct mapping of your design
  - Not so power in detecting latches

- *Synopsys Design Compiler*
  - Synthesis Tool
  - The embedded *Presto Compiler* can list your flip-flops and latches in details

```
> dv -no_gui
> read_verilog yourdesign.v
```

Check for Synthesizable (2/2)

```
Inferred memory devices in process
        in routine cache line 281 in file
                '/home/m97/gieks/cache/cache.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     block6_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block7_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block0_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|      state_reg      | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
|     block1_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|  mem_fetching_reg   | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|     block3_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block5_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block2_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block4_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
Current design is now '/home/m97/gieks/cache/cache.db:cache'
Loaded 1 design.
Current design is 'cache'.
cache
design_vision>
```

Checking latches
using Design Compiler