Write Efficient HDL Code

- Use parentheses control complex structure of a design.
- Use operator bit-width efficiently.
- Propagate constant value.
Use Parentheses Properly

- Out = a + b + c + d + e + f;
Use Parentheses Properly (cont.)

- Out = (a+b) + (c+d) + (e+f);
Use Operator Bit-Width Efficiently

module test(a,b,out);
    input [7:0] a,b;
    output [8:0] out;
    assign out=add_lt_10(a,b);

function [8:0] add_lt_10;
    input [7:0] a,b;
    reg [7:0] temp;
    begin
        if (b<10) temp=b;
        else temp=10;
        add_lt_10=a+temp[3:0];  //use [3:0] for temp
    end
endfunction

endmodule
Propagate Constant Value

```verilog
parameter size = 8;
wire [3:0] a, b, c, d, e;
assign c = size + 2;  // constant
assign d = a + 1;     // incremener
assign e = a + b;     // adder
```
Synopsys HDL Compiler Directive

- What we have mentioned
  - //synopsys full_case
  - //synopsys parallel_case
  - //synopsys translate_on & //synopsys translate_off control the HDL Compiler translation of Verilog code off & on
- Your dc_shell script should only contain commands that set constraints & attributes

```verilog
module trivial(a,b,f);
  input a,b;
  output f;
  assign f = a & b;

  //synopsys translate_off
  initial $monitor(a,b,f);
  //synopsys translate_on
endmodule
```
Coding Skill-Data-Path Duplication

No_dulpicated

module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1,PTR2;
input [15:0] ADDRESS, B;
input CONTROL;    // CONTROL is late arriving
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wile [15:0] ADDR;
assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
assign OFFSET = BASE - PTR; // Could be any function f(BASE,PTR)
assign ADDR = ADDRESS - {8'h00, OFFSET};
assign COUNT = ADDR + B;
endmodule

Dulpicated

module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] OFFSET1,OFFSET2;
wile [15:0] ADDR1,ADDR2,COUNT1,COUNT2;
assign OFFSET1 = BASE - PTR1; // Could be f(BASE,PTR)
assign OFFSET2 = BASE - PTR2; // Could be f(BASE,PTR)
assign ADDR1 = ADDRESS - {8'h00, OFFSET1};
assign ADDR2 = ADDRESS - {8'h00, OFFSET2};
assign COUNT1 = ADDR1 + B;
assign COUNT2 = ADDR2 + B;
assign COUNT = (CONTROL == 1'b1) ? COUNT1 : COUNT2;
endmodule
Coding Skill-Data-Path Duplication

- We assume that signal “CONTROL” is the latest arrival pin.
- By this skill, we will reduce latency but we must pay for it, area!
Coding Skill -- operator in if

- We assume that signal “A” is latest arrival signal

Before_improved

```verilog
module cond_op(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
// A is latest arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
    if (A + B < 24)
        Z <= C;
    else
        Z <= D;
end
endmodule
```

Improved

```verilog
module cond_op_improved (A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
// A is latest arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
    if (A < 24 - B)
        Z <= C;
    else
        Z <= D;
end
endmodule
```
Coding Skill -- operator in if

- In this example, not only latency reduced, but also area reduced.

Before_improved

Improved
Learning Objectives

- Describe procedural continuous assignment statements `assign`, `deassign`, `force`, and `release`.
- Understand how to override parameters by using the `defparam` statement at the time of module instantiation.
- Explain conditional compilation and execution of parts of the Verilog.
- Identify system tasks for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.
Procedural Continuous Assignments

- Procedural Assignments assign a value to a register.
  - The value stays in the register until another procedural assignment puts another value in that register.

- Procedural Continuous Assignments behave differently.
  - They are procedural statements which allow value of expressions to be driven continuously onto register or net for limited periods of time.
  - They override existing assignments to a register or net,
  - They provide an useful extension to regular procedural assignment statement.
assign & deassign

- First type of procedural continuous assignment.
- The LHS of procedural continuous assignments can only be a register or a concatenation of registers.
- It cannot be a part or bit select of a net or an array of registers.
- Procedural continuous assignments override the effect of regular procedural assignments.
- Procedural continuous assignments are normally used for controlled periods of time.
module edge_dff(q, qbar, d, clk, reset);

output q, qbar;
input  d, clk, reset;
reg q, qbar;

always @(negedge clk)
begin
  q = d;
  qbar = ~d;
end

always @(reset)
if (reset)
begin
  assign q = 1'b0;
  assign qbar = 1'b1;
end
else
begin
  deassign q;
  deassign qbar;
end

endmodule
force & release

- Second form of the procedural continuous assignments.
- They can be used to override assignments on both register and nets.
- Typically, used in the interactive debugging process, where certain registers or nets are forced to a value and the effect on other registers and nets is noted.
- They not be used inside design blocks.
- They should appear only in stimulus or as debug statements.
Example

module stimulus;
...
// instantiate the d-flip-flop
edge_dff dff(Q, Qbar, D, CLK, RESET);
...
initial
begin
  // these statements force value of 1
  // on dff.q between time 50 and 100,
  // regardless of the actual output of
  // the edge_dff
  #50 force dff.q = 1'b1;
  #50 release dff.q;
end
...
endmodule

module top;
...
assign out = a & b & c;
...
initial
begin
  #50 force out = a | b & c;
  #50 release;
end
...
endmodule
Overriding Parameters

- Parameters can be defined in a module definition.
- However, during compilation of Verilog modules, parameter values can be altered separately for each module instance.
- this allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values.
- Two ways to override parameter values:
  - defparam statement
  - module instance parameter value assignment.
Example

```verilog
// Define a module hello_world
module hello_world;
parameter id_num = 0; // define a module identification number = 0
initial
    $display("Displaying hello_world id number = %d", id_num);
endmodule

// Define top-level module
module top;
// Change parameter values in the instantiated modules
// Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

// Instantiate two hello_world modules
hello_world w1( );
hello_world w2( );
endmodule
```

Displaying hello_world id number = 1
Displaying hello_world id number = 2
Condition Compilation and Execution

- A portion of Verilog might be suitable for one environment and not for the other.
- The designer does not wish to create two versions of Verilog design for the two environments.
- Instead, the designer can specify that the particular portion of the code be compiled only if certain flag is set. This is called conditional compilation.
- A designer might also want to execute certain parts of the Verilog design only when a flag is set at runtime. This is called conditional execution.
Conditional Compilation

- Conditional compilation can be accomplished by using compiler directives `ifdef, `else, and `endif.

```vhdl
// Conditional Compilation
// Example 1
`ifdef TEST // compile module test only if
module test; // text marco TEXT is defined
...
endmodule
`else // compile the stimulus as default
module stimulus;
...
endmodule
`endif // completion
```

```vhdl
// Conditional Compilation
// Example 2
module top;
...
bus_master b1 ( );
`ifdef ADD_B2
    bus_master b2 ( );
`endif
    `endif
endmodule
```
Conditional Execution

- Conditional execution flags allow the designer to control statement execution flow at run time.
- All statements are compiled but executed conditionally.
- Conditional execution flags can be used only for behavioral statements.
- The system task keyword $test$plusargs is used for conditional execution.

```verilog
// Conditional Execution
module test;
...
initial begin
  if ($test$plusargs("DISPLAY_VAR"))
    $display("Display = %b ", {a, b, c}); // display only if flag is set
  else
    $display("No Display"); // otherwise no display
end
endmodule
```
Files Open/Write/Close

- File Output
  - Opening a file
    - integer filepointer;
    - initial filepointer = $fopen("file.out");
  - Writing to file
    - fdisplay(filepointer, "Display");
    - fmonitor(filepointer, "P1=%b", P1);
  - Closing files
    - $fclose(filepointer);
Useful System Tasks

- **Strobing**
  - The `$strobe` task is very similar to the `$display` task except for a slight difference.
  - The `$strobe` task provides a synchronization mechanism to ensure that data is displayed only after all other assignment statements.

- **Random Number Generation**
  - `$random`; or `$random(seed)``
  - returns a 32-bit random number.
Useful System Tasks

- Initializing memory form file

```verilog
reg [7:0] memory[0:7];
initial begin
    $readmemb("init.dat", memory);
    // display contents of initialized memory
    for (i=0; i<8; i=i+1)
        $display("Memory[%0d] = %b", i, memory[i]);
end

Memory[0] = xxxxxxxx
Memory[1] = xxxxxxxx
Memory[2] = 11111111
Memory[3] = 01010101
Memory[4] = 00000000
Memory[5] = 10101010
Memory[6] = 1111zzzz
Memory[7] = 00001111
```

uninitialized locations default to x

```text
init.dat

@002
11111111 01010101
00000000 10101010

@006
1111zzzz 00001111
```
Digital System Design

Coding Style

Chihhao Chao (趙之昊)
2009.3.25
Principles of RTL Coding Styles

- Readability
- Simplicity
- Locality
- Portability
- Reusability
- Reconfigurable / Parameterized
Naming Conventions

- Lowercase letters for signal names
- Uppercase letters for constants
- Case-insensitive naming
- Use $clk$ for clocks, $rst$ for resets
- Suffixes
  - $_n$ for active-low
  - $_a$ for async
  - $_z$ for tri-state
- Identical names for connected signals and ports
- Do not use HDL reserved words
- Consistency within group, division and corporation
File Header

- Should be included for all source files
- Contents
  - Author information
  - Revision history
  - Purpose description
  - Available parameters
  - Reset scheme and clock domain
  - Critical timing and asynchronous interface
  - Test structures
- A corporation-wide standard template
Example: ALU.v

1 // ******************************************************
2 // ACCESS Laboratory, Graduate Institute of Electronics Engineering, NTU
3 // -----------------------------------------------------------
4 // FILE NAME : ALU.v
5 // AUTHOR : Jih-Chiang Yeo
6 // DATE : 2004-03-01
7 // VERSION : 1.0
8 // PURPOSE : A ALU with Six Kind of Instructions
9 // -----------------------------------------------------------
10 // ABSTRACT
11 //
12 // This is a design of ALU with six kind of instructions. We define the -
13 // instruction set as follows:
14 //
15 // instruction[2:0] operation
16 // =======================================
17 // 000 two's complement addition
18 // 001 two's complement subtraction
19 // 010 bit-wise NOT
20 // 011 bit-wise AND
21 // 100 bit-wise OR
22 // 101 bit-wise XOR
23 // others no operation
24 //
25 // The default wordlength is eight bits, and the wordlength can be redef-
26 // ined by parameter DATA_WIDTH.
27 // -----------------------------------------------------------
Example: ALU.v

```vhdl
27 // REVISION HISTORY
28 //
29 //
30 // VERSION  DATE          AUTHOR               DESCRIPTION
31 // --------------------------------------------------------------
32 // 1.0       2004-02-26   Jih-Chiang Yeo       Original
33 // 1.1       2004-03-01   Jih-Chiang Yeo       Add some comments
34 //
35 // PARAMETERS
36 //
37 // PARA_NAME  RANGE     DESCRIPTION               DEFAULT
38 // -----------------------------------------------
39 // DATA_WIDTH [4,32]     width of data            8
40 //
41 // REUSE ISSUES
42 //
43 // RESET STRATEGY : asynchronous reset
44 // CLOCK DOMAINS : posedge trigger clock
45 // OTHER   ;
46 //**************************************************************************
```
Ports

- **Ordering**
  - One port per line with appropriate comments
  - Outputs first then inputs
  - Clocks, resets, enables, other controls, address bus then data bus

- **Instantiating Mapping**
  - Using *named mapping* instead of *positional mapping*
Example: ALU.v

```vhdl
46 // *************************************************************
47 module ALU(
48   // OUTPUTS
49   alu_out,  // The output of ALU
50   // INPUTS
51   input_1,  // one of the inputs of ALU
52   input_2,  // the other input of ALU
53   // CONTROLS
54   clk,      // posedge trigger clock
55   rst_a,    // asynchronous reset
56   instruction // instruction selection
57 );
58
59 parameter DATA_WIDTH = 8;
60
62 output [DATA_WIDTH-1:0] alu_out;
63 input  [DATA_WIDTH-1:0] input_1, input_2;
64 input   clk, reset;
65 input   [3:0]            instruction;
```
Pre-RTL Preparation Checklist

- Communicate design issues with your team
  - Naming conventions, revision control, directory trees and other design organizations.

- Have a specification for your design?
  - Take it for granted that everyone has a specification BEFORE they start coding.

- Design partition
  - Follow the specification’s recommendations for partition.
  - Break the design into major functional blocks.
Pre-RTL Preparation Checklist

- Work from the outside: I/O interfaces
  - Make sure the function and timing of each interface is clear.

- How are the buses defined?
  - Try to use unidirectional buses wherever possible.

- Are there any compatibility requirements?
  - Get the tester’s specification and understand how the chip is supposed to behave.

- What other IP are you using?
  - Start with the interface to each IP block.
**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.
- Always think of the poor guy who has to read your RTL code.
  - Correlate “top to bottom” in the RTL description with ”left to right” in the block diagram.
  - Comments and headers.
- Hierarchical design
Comments and Formats

- Appropriate comments
  - Process (always block), function, …

- Comment end statements

- One statement per line

- Coding in a tabular manner

- Line length restriction
  - A fixed number between 72-78

- Indentation
  - 2 or 4
  - do not use tab
Coding Practices

- Little-endian for multi-bit bus
  - [31:0] instead [0:31]
- Operand sizes should match
  - \texttt{reg [32:0] a; a = 33'h1_ffff_ffff; a = 1; // a is 33'h1_0000_0001}
- Use parentheses () in complex statements
- Do not assign signals don’t-care values
  - Avoid don’t-care propagation
- Reset all storage elements
  - Avoid don’t-care propagation
Coding Practices

- Use high level constructs (case, if, always@) as much as possible.
  - DC takes Boolean expressions and gate level instantiations and replaces them with a sum-of-products “pla”-like internal description that is fed to the gate level mapping optimization.
- Don’t instantiate gates unless you have to; make the code technology independent.
  - Put a wrapper around the gate level design.
- Use for-loops only for bit-wise operations that can only be described one bit at a time.
Combinational vs. Sequential Blocks

- Use separate always@ processes for sequential logic and combinational logic.
  - There is a sequential optimization process in DC.

- Combinational block
  - Use blocking (=) assignments
  - Minimize signals required in sensitivity list
  - Assignment should be applied in topological order of a circuit graph

- Sequential block
  - Use non-blocking (<=) assignments
  - Avoid race problems in simulation
Function, Conditional Statements

- Use function to model combinational logic when possible, instead of repeating the same sections of code.
- Know whether you have prioritized or parallel condition.
  - Prioritized: if-else, parallel: case
- Completely specify all branches of all conditional statements.
  - If you completely specify the case statement (or use default), DC will recognize the case is fully specified and parallel.
Coding for FSM

- Partition FSM and non-FSM logic
- Prefer Moore (PO is PI-independent) to Mealy (PO is PI-dependent)
- Prefer Moore with state-outputs as POs
- 3-always paradigm
  - One for sequential logic (Current State, CS, edge triggered)
  - One for next-state logic (NL, pure combinational block)
  - One for PO logic (OL, if required)
- Use parameters to define the state names
- Assign a default (reset) state
Portability

- Do not use hard-coded numbers
- Avoid embedded synthesis scripts
- Use technology-independent libraries
- Avoid instantiating gates
Clocks and Resets

- Simple clocking is easier to understand, analyze, and maintain
  - The preferred clocking structure is a single global clock and positive edge-triggered flops.
- Avoid using both edges of the clock
  - Duty-cycle sensitive
  - Difficult DFT process
- Don’t buffer clock and reset networks
  - Should be handled during physical synthesis later
- Avoid manual gated clock
  - Clock gating circuits tend to be technology specific and timing dependent.
Clocks and Resets

- Avoid internally generated clocks and resets
  - Limited testability
  - Result in low fault/test coverage in DFT/ATPG process

![Diagram of D flip-flop with clock signal]
Clocks and Resets

- Gated clock / internally generated clock design
  - If your design requires a gated clock, model it using synchronous load registers.

![Clock Generation Diagram]

- TOP
  - submodule1
  - submodule2
  - submodule3

- Clocks and Resets

- Clocks and Resets
Low Power

- Clock gating
  - 50%~70% power consumed in clock network reported
  - Gating the clock to an entire block
  - Gating the clock to a register

```verilog
always @(posedge clk)
  if (en)
    q <= q_nxt;
```

Assign clk1 = clk & en;
always @(posedge clk1)
  q <= q_nxt;

Synchronicity

- Infer technology-independent registers
  - (positive) edge-triggered registers
- Avoid latches intentionally
  - Except for small memory and FIFO
  - For low-power
- Avoid latches unintentionally
  - Avoid incomplete assignment in case statement
  - Use default assignments
  - Avoid incomplete if-then-else chain
- Avoid combinational feedback loops
  - STA and ATPG problem
Coding for Synthesis

- No # delay statements
- Avoid *full_case* and *parallel_case*
  - Evil twin
  - Pre-synthesis and post-synthesis simulation mismatch
- Explicitly declare wires
- Avoid glue logic at the top-level
- Avoid expression in port connections
Partitioning

- Register all outputs
  - Make output drive strengths and input delay predictable
  - Ease time budgeting and constraints
Partitioning

- Keep related logic together
  - Improve synthesis quality
- Partition logic with different design goals
- Avoid asynchronous logic
  - Technology dependent
  - More difficult to ensure correct functionality and timing
  - As small as possible and isolation
- Keep sharable resources in the same block
Partitioning

- Avoid timing exception
  - Point-to-point, false path, multi-cycle path
- Keep sharable resources in the same block
Design Rule Checker

- nLint is a design rule checker that can help hardware designers to create syntax and semantics correct HDL code.
- nLint reads in HDL source code, analyzes it, and outputs \textbf{warnings and errors}.
  - Including position and message.
- nLint checks against approximately 200 rules, which are \textbf{RMM} compliant.
Example: bad_conditional.v

```verilog
always @(in1 or select1) begin
    case (select1)
        2'b00: out1 = 1'b0;
        2'b01: out1 = in1;
        2'b10: out1 = ~in1;
    endcase
end

always @(in2) begin
    if (select2) begin
        out2 = in2;
    end else begin
        out2 = ~in2
    end
end
```

1. Incomplete conditional assignment
2. Incomplete sensitivity list
3. Error!! need “;”
Use command

- nLint bad_conditional.v

```
white:/userhome/m92/wljcy/test% nLint bad_conditional.v
logDir = /userhome/m92/wljcy/test/nLintLog

nLint - A HDL Design Rule Checker, Release 1.1v8 (SOLARIS) 11/27/2001
Copyright (C) 1996 - 2001 by Novas Software, Inc.
Analyzing...
  source file "bad_conditional.v" - 1 error(s), 0 warning(s)
Linting...
Rule setting file = /userhome/m92/wljcy/nLint.rs.
*** Working Directory [/userhome/m92/wljcy/test/] ***

bad_conditional.v(22): Warning 23007: incompletely specified case statement detected (Language Construct) (Verilog)
bad_conditional.v(23): Warning 23003: latch inferred on signal "out1" (Synthesis) (Verilog)
bad_conditional.v(30): Warning 23011: signal "select2" should be included in the _sensitivity list (Simulation,Synthesis) (Verilog)
bad_conditional.v(32): Error 11000: syntax error -> "end" () (Unknown)
bad_conditional.v(34): Warning 23003: latch inferred on signal "out2" (Synthesis) (Verilog)

Lint processing done.
```
No Error & Warning

```
white:/userhome/m92/wljcy/test%
nLint bad_conditional.v
logDir = /userhome/m92/wljcy/test/nLintLog

nLint - A HDL Design Rule Checker, Release 1.1v8 (SOLARIS) 11/27/2001
Copyright (C) 1996 - 2001 by Novas Software, Inc.
Analyzing...
    source file "bad_conditional.v"
Linting...
Rule setting file = /userhome/m92/wljcy/nLint.rs.
Lint processing done.
```
GUI

- nLint –gui &
Import Design
Edit File

```vhdl
module bad_conditional(
    out1,
    out2,
    in1,
    in2,
    select1,
    select2
);

output out1;
output out2;
input in1;
input in2;
input [1:0] select1;
input select2;
reg out1;
reg out2;
```

Modeling Techniques and Coding Style 2009.3.25   Chihhao Chao

pp. 59
File Organizer
Rule Organizer

data lost from operand "%s" due to width mismatch or shift operation
Lint -> Run

source file "/userhome/m92/wljcy/test/badconditional.v" - 1 error(s), 0 warning(s)

Linting...
Rule setting file = /userhome/m92/wljcy/nLint.rs.
Fix error
Fix Warning 1

File: \texttt{/userhome/m92/wljcy/test/bad\_conditional.v(30)}: signal "select2" should be included in the sensitivity list.

```vhdl
always@ (in2) begin
  if (select2) begin
    out2 = in2;
  else
    out1 = in1;
  endcase
end
```
Fix Warning 2

```vhdl
always @(in1 or select1) begin
  case (select1)
    2'b00: out1 = 1'b0;
    2'b01: out1 = in1;
    2'b10: out1 = ~in1;
  endcase
end
```

Warning:
```
/home/m92/wlcy/test/bad_conditional.v(23): latch inferred on signal "out1"
```
Search Rule

- Right click -> Search Rule
Show Source On nTrace

- Tools -> Preferences
No Error & Warning

```
source file "~/userhome/m92/wljcy/test/bad_conditional.v"
Linting...
Rule setting file = ~/userhome/m92/wljcy/nLint.rs.
```