Logic Synthesis
Course Outline

- Design Compiler Overview
- Some Words about Physical Compiler
- Coding Styles
- Partitioning
- Design Compiler Commands
- Additional Commands for Physical Compiler
- Timing Analysis
- Interaction with Layout Tools
- Speed/Area Optimization
- Logic Structure in High Level Code
- Physical Compiler Overview
Design Compiler Overview

- HDL Code (RTL)
- Cell Libraries (timing and function)
- Constraints (timing, area ...)

Design Compiler

Mapped Netlist
Physical Compiler Overview

- HDL Code (RTL) or mapped Gate Level
- Cell Libraries (timing and function) (xxx.lib -> xxx.db)
- Physical Libraries (cells and technology) (xxx.plib -> xxx.pdb)
- Constraints (timing & physical)
- Floorplan
- Physical Compiler
- Placed Netlist
• synthesis = mapping + optimization

• Design Compiler is constraint-driven
• Designer explores area vs. speed trade-offs simply by varying constraints
Few Synopsys rules of Thumb

- “There is no ‘golden’ script for synthesis”

- “The random setting of optimization switches and constraints to meet your speed goals is NOT a credible methodology”

- “Physics dictates what will fit between two clock edges”
General Synopsys Synthesis Flow

- Analyze and Elaborate RTL code
- Apply Constraints
- Synthesize
- Constraints Met?
  - Yes
    - End
  - No
    - Identify Problem
- Re-code RTL
- Create Custom Wireload
- Floorplan
- Synthesis tricks
General Synopsys Physical Synthesis Flow (mpc)

1. Analyze and Elaborate RTL code
2. Apply Constraints including sime floorplan requirements
3. "Floorplan"
4. Constraints Met?
   - Yes: Create Floorplan → End
   - No: Identify Problem
5. compile_physical or physopt
6. Synthesis tricks
7. No: Re-code RTL
8. Yes: Re-code RTL
General Synopsys Physical Synthesis Flow (floorplan based)

1. mpc run results
2. Update floorplan (can be based on mpc run)
3. Analyze and Elaborate RTL code
4. Apply Constraints including same floorplan requirements
5. compile_physical or physopt
6. Constraints Met?
   - Yes: End
   - No: Identify Problem
7. Re-code RTL
8. Synthesis tricks

Resize up/down, relocate pins etc.
Integration with Structures and other Blocks

- Integration may cause transition problems (violations)
- Top-level compile to fix any problem
- Best way is to model the structures
Coding Style

• Inferring Sequential Devices
• Three-State Inference
• Combinational Logic
• case vs. if synthesis
• Finite State Machines TBD? (Exists)?
Inferring Sequential Devices

- Synopsys can *infer* a broad range of FF’s and latches:

<table>
<thead>
<tr>
<th>Latch</th>
<th>Latch w/ Async</th>
<th>Latch w/Dual Async</th>
<th>Latch w/ Sync</th>
<th>DFF</th>
<th>DFF w/ Async</th>
<th>DFF w/Dual Async</th>
<th>DFF w/ Sync</th>
<th>Muxed DFF</th>
<th>JKFF</th>
<th>MS latch</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

- *An inference report* indicates type, width, and presence of: Bus, Async Reset, Async Set, Sync Reset, Sync Set and Sync Toggle.

**Example:** Inference Report for D Flipflop with Async Reset:

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_reg</td>
<td>Flip-Flop</td>
<td>1</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- For more details refer to Synopsys on-line documentation.
D-Flip-Flops

```Verilog
module dffs (clk, in1, in2, cond, rst_b, out1, out2);
  input  clk, in1, in2, cond, rst_b;
  output out1, out2;
  reg out1, out2;

  always @(posedge clk) begin
    out1 <= in1;
  end

  always @(posedge clk or negedge rst_b) begin
    if (!rst_b)      out2 <= 1'b0;
    else      out2 <= in2;
  end

endmodule
```

*Always use non-blocking assignments for flip-flops.*
**Transparent Latches**

```verilog
module latches (clk, in1, in2, rst_b, out1, out2);
input clk, in1, in2, rst_b;
output out1, out2;
reg out1, out2;

always @(in1 or clk) begin
  if (clk)
    out1 <= in1;
end

always @(in2 or rst_b or clk) begin
  if (!rst_b)
    out2 <= 1'b0;
  else if (clk)
    out2 <= in2;
end
endmodule

Use non-blocking assignments for latches as well
```
**Transparent latches (cont.) - result of synthesis**

Inferred memory devices in process
in routine latches line 7 in file kuku

```
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1_reg</td>
<td>Latch</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

Inferred memory devices in process
in routine latches line 12 in file kuku

```
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>out2_reg</td>
<td>Latch</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```
**Transparent latches (cont.) - result of synthesis**

```
module latches ( clk, in1, in2, rst_b, out1, out2 );
input  clk, in1, in2, rst_b;
output out1, out2;
  itlrpc out2_reg ( .C(clk), .D(in2), .RB(rst_b), .Q(out2) );
  itlpc out1_reg ( .C(clk), .D(in1), .Q(out1) );
endmodule
```

- Note that the instance names of the latches are derived from the HDL reg names.
**Gated Clocks**

- DC doesn’t understand that condition of the gated clock should have setup time with regard to the clock. As a result there is a possibility for false select.
- There is a danger of hold time violation when using gated clocks. Checking hold time violations requires best case libraries.
- Refrain from using gated clocks unless you put them into the clock generator block.
- From below you will see that the gated clocks problem exists for multiple-clock flip-flops and for transparent latches.

**Exception**

- New power compiler and clock tree synthesis flows enable manual instatiation of gated clocks as well as automatic tool inference. User gated clocks in RTL source should be INTENTIONAL and include a latch and and AND function (glithless design). They should be used ONLY where Power Compiler has no ability to detect the shared clocking condition (e.g: stop/idle for global power saving).
module gated_clock (clk1, clk2, in1, in2, cond, rst_b, out1, out2, out3);
input  clk1, clk2, in1, in2, cond, rst_b;
output out1, out2, out3;
reg out1, out2, out3;
wire clk;

/* conditional flip-flop - no problem */
always @(posedge clk1 or negedge rst_b) begin   if (!rst_b)      out1 <= 1'b0;   else if (cond)      out1 <= in1;end

/* conditional latch - refrain from using it */
always @(clk1 or rst_b or in1 or cond) begin   if (!rst_b)      out2 <= 1'b0;   else if (cond & clk1)      out2 <= in1;end

/* multiple clocks flip-flop - refrain from using it */
assign clk = clk1 || clk2;
always @(posedge clk or negedge rst_b) begin   if (!rst_b)      out3 <= 1'b0;   else      out3 <= in2;end
endmodule
**Gated Clocks (cont.) - result of synthesis**

module gated_clock ( clk1, clk2, in1, in2, cond, rst_b, out1, out2, out3 );
input  clk1, clk2, in1, in2, cond, rst_b;
output out1, out2, out3;
wire clk, n56, n164;

mux21b z52 ( .A(out1), .B(in2), .S0(cond), .OUT(n164) );
dffrpz out1_reg ( .C(clk1), .D(n164), .RB(rst_b), .Q(out1) );

iand2b z51 ( .INPUT1(clk1), .INPUT2(cond), .OUTPUT1(n56) );
itlrpc out2_reg ( .C(n56), .D(in1), .RB(rst_b), .Q(out2) );

ior2b z50 ( .INPUT1(clk1), .INPUT2(clk2), .OUTPUT1(clk) );
dffrpc out3_reg ( .C(clk), .D(in2), .RB(rst_b), .Q(out3) );

endmodule

• **Note that gated clocks exist only on out2_reg and out3_reg.**
Gated Clocks (cont.) - how to prevent them

• Don’t use multiple clocked flip-flops
• Conditional latches could be implemented in the following way:

According to evaluation done in Design2 the timing and area of the alternative implementation is only slightly worse than of the original one.
• Explicit (“manual”) clock gating is also possible if multiple latches shared one condition (power saving issue)
Gated Clocks (cont.) - explicit gated clocks

// in the “clock generator”
output clk_gated;
reg cond_latched;
always @(clk or cond)
    if (~clk)
        cond_latched <= cond;
assign clk_gated = (cond_latched | scan_enable) & clk;
.
.
// In the module
input clk_gated;
reg out;

always @(posedge clk_gated or negedge rst_b) begin
    if (!rst_b)
        out <= 1’b0;
    else
        out <= in;
end
.
.

\[\text{cond} \rightarrow \text{TL} \rightarrow \text{clk}\_\text{gated}\]
Three-State Inference

• A three-state driver is inferred when the value z is assigned to a

    module ff_3state (data1, data2, clk, three_state, out1, out2);
    input  data1, data2, clk, three_state;
    output out1;
    output out2;

    reg  out1;
    reg  out2_data;

    always @ (posedge clk)
    begin
        if (three_state)
            out1 = 1’bz;
        else
            out1 = data1;
    end

    always @ (posedge clk)
    out2_data = data2;

    assign out2 = three_state ? 1’bz : out2_data;

endmodule

• Do not use tristate buffers internal to a block.

• DO not define bidirectional ports for a synthesized block.
module ff_3state ( data1, data2, clk, three_state, out1, out2 );
input  data1, data2, clk, three_state;
output out1, out2;

wire n98, n99, n100, n101;
trinvc out1_tri ( .INPUT1(n98), .INPUT2(n99), .OUTPUT1(out2) );
iinve U21 ( .INPUT1(three_state), .OUTPUT1(n99) );
trinvc out1_tri ( .INPUT1(n100), .INPUT2(n101), .OUTPUT1(out1) );
dffpb out1_reg ( .C(clk), .D(data1), .QB(n100) );
dffpb out1_tri_enable_reg ( .C(clk), .D(three_state), .QB(n101) );
dffpb out2_data_reg ( .C(clk), .D(data2), .QB(n98) );
endmodule
**Combinational Logic**

Use combinational logic for:

- Intermediate variables for clarity
- Complex logic that becomes unwieldy in the FF always loop
- Functions that are used more than once with different parameters
- Resource Sharing
wire yy = (A & C | (B != 'GO)) ? D : (D & !A);
wire start_count = start & (count < 17);
reg y;
always @(posedge clk)
  if (start_count)
    y <= yy;

reg yy; // not a real register
always @(A or B or C or D)
begin
  if (A & C | (B != 'GO))
    yy = D;
  else
    yy = D & !A;
end
reg y;
always @(posedge clk)
  if (start & (count < 17))
    y <= yy;
HDL Compiler understands some operators and automatically generates the logic to implement them:

- +/- will use inc, dec, incdec, add, sub, addsub as appropriate
- > >= < <= will generate a comparator

```vhdl
reg [7:0] a, b;
reg [7:0] y;

always @ (posedge clk)
if (a > b)
    y <= a + b;
```

**DesignWare Resources**
Case Synthesis

- case statements infer "tall and skinny" muxes.

```verilog
always @(SEL or A or B or C or D)
begin
    case (SEL)
        2'b00 : OUTC = A;
        2'b01 : OUTC = B;
        2'b10 : OUTC = C;
        default : OUTC = D;
    endcase
end
```

Note: Actual gates synthesized might not be a 4->1 MUX
Case Synthesis (cont.)

Statistics for case statements in always block at line 8 in file kuku

<table>
<thead>
<tr>
<th>Line</th>
<th>full/ parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>auto/auto</td>
</tr>
</tbody>
</table>

Current design is now
{
"try1"
}
Case Synthesis (cont.) - WARNINGS!!!

• If you do not specify all possible branches of the case statement HDL Compiler will synthesize a latch.

• If HDL Compiler can’t statically determine that branches are parallel, it synthesizes hardware that includes a priority encoder.
Case Synthesis - “full case”

• A case statement is called full case if all possible branches are specified.

• Synopsys will synthesize a latch if you don’t specify all possible branches of a case statement. Use // synopsys full_case directive immediately after the case expression if

```verilog
always @(sel or a or b or c)
begin
  case (sel)
    2'b00 : outc <= a;
    2'b01 : outc <= b;
    2'b10 : outc <= c;
    endcase
end
```

And what is wrong here?

A case statement that is Parallel but not Full
Case Synthesis - “full case” (cont.)

Synopsys statistics WITHOUT // synopsys full_case directive

Statistics for case statements in always block at line 8 in file kuku

<table>
<thead>
<tr>
<th>Line</th>
<th>full/ parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>no/auto</td>
</tr>
</tbody>
</table>

Inferred memory devices in process
   in routine try2 line 8 in file kuku

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>outc_reg</td>
<td>Latch</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Current design is now try2
Case Synthesis - “full case” (cont.)

always @(sel or a or b or c)
begin
    case (sel) // synopsys full_case
        2'b00 : outc = a;
        2'b01 : outc = b;
        2'b10 : outc = c;
        Blocking assignment
    endcase
end

Note: This is not a recommended style! Prefer full description of the case.

Synopsys statistics WITH // synopsys full_case directive

Warning: You are using the full_case directive with a case statement in which not all cases are covered. (HDL-370)

Statistics for case statements in always block at line 8 in file kuku

=============================================== |           Line           |  full/ parallel  |  user/auto  |
===============================================
|                                               | 11               |              |            |
===============================================
Current design is now try2
Case Synthesis - “parallel case”

- A case statement is called parallel case if HDL Compiler can determine that no cases overlap.

- Synopsys will synthesize a priority encoder unless // synopsys parallel_case directive is used.

```verilog
always @(w or x)
begin
    case (1'b1)
        w : b = 0;
        x : b = 1;
    endcase
end
```

A case statement that is Not Full or Parallel
Case Synthesis - “parallel case” (cont.)

Synopsys statistics WITHOUT // synopsys parallel_case full_case directive

Statistics for case statements in always block at line 7 in file kuku

<table>
<thead>
<tr>
<th>Line</th>
<th>full/ parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>no/no</td>
</tr>
</tbody>
</table>

Inferred memory devices in process
in routine try3 line 7 in file kuku

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_reg</td>
<td>Latch</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Case Synthesis - “parallel case” (cont.)

Synopsys statistics WITH // synopsys parallel_case full_case directive

Warning: You are using the full_case directive with a case statement in which not all cases are covered. (HDL-370)
Warning: You are using the parallel_case directive with a case statement in which some case-items may overlap. (HDL-371)

Statistics for case statements in always block at line 7 in file kuku

<table>
<thead>
<tr>
<th>Line</th>
<th>full/ parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>user/user</td>
</tr>
</tbody>
</table>

Current design is now try3
**if-then-else Synthesis**

- if-then-else statements *infer* priority-encoded "cascading" MUXs.

```
always @(SEL or A or B or C or D)
begin
    if (SEL == 2'b00)
        OUTC = A;
    else if (SEL == 2'b01)
        OUTC = B;
    else if (SEL == 2'b10)
        OUTC = C;
    else
        OUTC = D;
end
```

Note: Actual gates synthesized might not match those shown above.
**case vs if statements**

- Note that only case statement may be implemented without priority

```
always @(.)
begin
  if (. .)
  ...
  else if (. .)
  ...
  else if (. .)
  ...
  else
  ...
end
```

```
always @(.)
begin
  if (. .)
  ...
  if (. .)
  ...
  else if (. .)
  ...
  else
  ...
end
```

```
always @(.)
begin
  case (. .)
    A: ...
    B: ...
    C: ...
  endcase
end
```
Safe Coding Rules

Sensitivity Lists

Use complete sensitivity lists for combinational always statements. Otherwise, pre-synthesis (high-level) simulation might not match post-synthesis (gate-level) simulation results.

Elaboration will detect incomplete sensitivity lists and generate a warning.

always @(A)
begin
    C = A || B;
end

A
B
C

A
B
C

High-Level Simulation
Synthesized Netlist
Gate-Level Simulation
**Safe Coding Rules (cont.)**

*case and if Statements (part 1)*

- Completely specify all clauses for every *case* and *if* statement.
- Completely specify all outputs for every clause of each *case* or *if* statement.
- Failure to do so will cause extra latches or flip-flops to be synthesized.
- In combinational always blocks - use blocking assignments

```verilog
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
```

What’s wrong with this code?

- S Output
- Missing Clause

The code is non-blocking, which is incorrect for combinational logic.
Safe Coding Rules (cont.)

*case* and *if* Statements (part 2)

- Whenever possible, use a *case* statement with *default* (rather than an *if* statement).

```vhdl
always @(SEL or INNER)
    case (SEL)
    2'b00: Z = 2'b00;
    2'b01: case (INNER)
        2'b00: Z = 2'b11;
        2'b01: Z = 2'b01;
        default: Z = 2'b00;
    endcase
    2'b10: Z = 2'b11;
    default: Z = 2'b10;
endcase
```
**Implication Example**

- Good style takes advantage of if-else priority to synthesize correct

**Bad Style**

```plaintext
case (STATE)
  IDLE:
    if (LATE == 1'b1) 
      ADDR_BUS = ADDR_MAIN;
    else
      ADDR_BUS = ADDR_CNTL;
  INTERRUPT:
    if (LATE == 1'b1) 
      ADDR_BUS = ADDR_MAIN;
    else
      ADDR_BUS = ADDR_INT;
endcase
```

**Good Style**

```plaintext
if (LATE == 1'b1)
  ADDR_BUS = ADDR_MAIN;
else
  case (STATE)
    IDLE:
      ADDR_BUS = ADDR_CNTL;
    INTERRUPT:
      ADDR_BUS = ADDR_INT;
  endcase
```

![Diagram showing LATE and ADDR_MAIN affecting ADDR_BUS](image-url)
Partitioning

What is Partitioning? vs. SOG!

- Partitioning is dividing a design into smaller parts.

- How do you decide on the partitioning of your design?
  - By functionality
  - By designer’s skill
  - For optimal synthesis result
  - All of the above

- Partitioning is done within the HDL description and/or the Design
Partitioning Within the HDL Description

- Module statements create hierarchical design blocks.
- Continuous assignments and *always*@ statements do not create
Partitioning Within Design Compiler

- A designer can re-partition a design after it has been entered into Design Compiler.
**The `group` command**

- Creates a new hierarchical block containing the specified cells.
- The designer provides a “design name” and a “cell name” for the new block.
The `ungroup` command

- Removes levels of hierarchy of the named instances. Very important command.
- `-simple_names` option should be used to remove the extra
Why Partition for Synthesis?

- Produce the best synthesis results.
- Speed up optimization run times.
- Simplify the synthesis process.

Why SOG (Sea Of Gates) for Synthesis?

- Required less resources. (Tool, Designers)
- Less timing constraints to deal with.
- Further placement optimization is allowed.


**Partitioning Rules for Synthesis**

- No hierarchy in combinational paths.
- Register all outputs.
- No glue logic between blocks.
- Separate designs with different goals.
- Maintain a reasonable block size.
- Separate logic, pads, clocks and non-synthesizable structures.
No Hierarchy in Combinational Paths

Bad Example

The path between two REG’s is divided between three different blocks.

- Optimization is limited because hierarchical boundaries prevent sharing of common terms.
No Hierarchy in Combinational Paths (cont.)

Better Example

Related combinational logic is grouped into one block; thus, all related combinational logic is at the same level of hierarchy.
No Hierarchy in Combinational Paths (cont.)

Best Example

Related combinational logic is grouped into the same block that contains the destination register for the combinational logic path.
A NAND gate was added to the TOP level block description, to “bridge” the two instantiated lower-level blocks.
No Glue Logic Between Blocks (cont.)

Good Example

Merge glue logic into the related combinational logic description of the lower-level architectural statements.

- The merged glue logic can now be optimized away.
**Separate Designs with Different Goals**

**Bad Example**

*REG A is in the critical path, but REG C is not.*

- Optimization is limited because the designer cannot isolate parts of a block and optimize them solely for area or for speed.
Separate Designs with Different Goals (cont.)

Use different modules to partition the design into
Maintain a Reasonable Block Size

Bad Example

- If blocks are too small, the designer may be restricting optimization with artificial boundaries.
- If blocks are too big, compile run times are very long; quick iterations