Introduction to HDL Synthesis

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:34534
Email: jimmy@ee.ncu.edu.tw

Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level optimization
  - Multi-level optimization
- Modeling for synthesis
Levels of Design

**STRUCTURAL DOMAIN**
- Processors, Memories, Buses
- Registers, ALUs, MUXs
- Logic gates, Flip-flops
- Transistors

**BEHAVIORAL DOMAIN**
- Flowcharts, Algorithms
- Logic synthesis
- Boolean expression
- Switch function

**PHYSICAL DOMAIN**
- Transistor layout
- Cells
- Chips, Modules
- Boards, MCMs

**Synthesis**
- Transform HDL descriptions into logic gate networks (structural domain) in a particular library
- Advantages
  - Reduce time to generate netlists
  - Easier to retarget designs from one technology to another
  - Reduce debugging effort
- Requirement
  - Robust HDL synthesizers
HDL Synthesis

Synthesis = Domain Translation + Optimization

Domain translation

Behavioral domain

Optimization

(area, timing, power...)

RTL synthesis

Structural domain

Domain Translation

Consistent with data manipulation functions

Consistent with special semantics

Input HDL Description

3-address Code

Combinational Circuit Generation

Initial Structural Netlist

Special Element Inferences

Optimization

(area, timing ...)

7-5

7-6
Two-Level Logic Optimization

- Two-level logic representations
  - Sum-of-product form
  - Product-of-sum form
- Two-level logic optimization
  - Key technique in logic optimization
  - Many efficient algorithms to find a near minimal representation in a practical amount of time
  - In commercial use for several years
  - Minimization criteria: **number of product terms**
- Example: \( F = XYZ + X\overline{Y}Z + X\overline{Y}Z + \overline{X}YZ + XY\overline{Z} \)
  \[ F = \overline{X}Y + YZ \]

---

Multi-Level Logic Optimization

- Transforms a combinational circuit to meet performance or area constraints
  - Two-level minimization
  - Common factors or kernel extraction
  - Common expression resubsitution
- In commercial use for several years
- Example:
  - Example:
    - \( f_1 = abcd + abce + \overline{abc}d + \overline{abc}d + \bar{a}c + cdf + \overline{abc}de + \overline{abc}def \)
    - \( f_1 = \overline{c} \overline{(a + x)} + a\overline{c}x \)
    - \( f_2 = bdg + bdg + bdg + bdg + bdg \)
    - \( x = d(b + f) + \overline{d} \overline{(b + e)} \)
    - \( f_2 = gx \)
Outline

- Synthesis overview
- **RTL synthesis**
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level optimization
  - Multi-level optimization
- Modeling for synthesis

Typical Domain Translation Flow

- Translate original HDL code into 3-address format
- Conduct special element inferences before combinational circuit generation
- Conduct special element inferences process by process (local view)
Combinational Circuit Generation

- Functional unit allocation
  - Straightforward mapping with 3-address code
- Interconnection binding
  - Using control/data flow analysis

Functional Unit Allocation

- 3-address code
  - $x = y \text{ op } z$ in general form
  - Function unit op with inputs $y$ and $z$ and output $x$
Interconnection Binding

- Need the dependency information among functional units
  - Using control/data flow analysis
  - A traditional technique used in compiler design for a variety of code optimizations
  - Statically analyze and compute the set of assignments reaching a particular point in a program

Control/Data Flow Analysis

- Terminology
  - A definition of a variable x
    - An assignment assigns a value to the variable x
    - d1 can reach d4 but cannot reach d3
    - d1 is killed by d2 before reaching d3
  - A definition can only be affected by those definitions being able to reach it
  - Use a set of data flow equations to compute which assignments can reach a target assignment

/*d1*/  x = a;
if(s) begin
/*d2*/   x = b;
/*d3*/   y = x + a;
end
/*d4*/  y = x;

/*d1*/ x=a;
/*d4*/ y=x;
/*d2*/ x=b;
/*d3*/ y=x+a;
**Combinational Circuit Generation**

```
always @(x or a or b or c or d or s)
begin
   // d1
   x = a + b;
   // d2
   if (s) x = c - d;
   // d3
   else x = x;
   // d4
   y = x;
end

Input HDL
```

```
always @(x or a or b or c or d or s)
begin
   // d1
   x = a + b;
   // d2
   if (s) x = c - d;
   // d3
   else x = x;
   // d4
   y = x;
end
```

**Functional unit allocation**

```
computed by control/data flow analysis
```

**Interconnection binding**

```
In[d1] = {d4, d5}
In[d2] = {d1, d5}
In[d3] = {d1, d5}
In[d4] = {d2, d3, d5}
In[d5] = {d4, d5}
```

```
always @(x or a or b or c or d or s)
begin
   // d1
   x = a + b;
   // d2
   if (s) x = c - d;
   // d3
   else x = x;
   // d4
   y = x;
end
```

**Special Element Inferences**

- Given a HDL code at RTL, three special elements need to be inferred to keep the special semantics
  - Latch (D-type) inference
  - Flip-Flop (D-type) inference
  - Tri-state buffer inference
- Some simple rules are used in typical approaches

```
reg Q;
always@((D or en)
if(en) Q = D;
Latch inferred!!

reg Q;
always@(@posedge clk)
Q = D;
Flip-flop inferred!!

reg Q;
always@((D or en)
if(en) Q = D;
else Q = 1’bz;
Tri-state buffer inferred!!
```
## Preliminaries

- Sequential section
  - Edge triggered always statement
- Combinational section
  - All signals whose values are used in the always statement are included in the sensitivity list

```verbatim
reg Q;
always@(posedge clk)
  Q = D;
```

**Sequential section**
Conduct flip-flop inference

```verbatim
reg Q;
always@(in or en)
  if(en) Q = in;
```

**Combinational section**
Conduct latch inference

---

## Rules of Latch Inference

- Condition 1: There is no branch associated with the output of a conditional assignment for a value of the selector
  - Output depends on its previous value implicitly

always@(s or a)
if(s) Q = a;

Q = a;

always Q = Q;

Q depends on its previous value at this branch
Rules of Latch Inference

- **Condition 2**: The output value of a conditional assignment depends on its previous value explicitly

\[ \text{always@}(s \text{ or } z \text{ or } y \text{ or } a) \]
\[ \begin{array}{l}
\text{begin} \\
\text{z} = y; \\
\text{if}(s) \ y = a; \\
\text{else} \ y = z; \\
\text{end}
\end{array} \]

\( y \) depends on its previous value at this branch via the assignment \( z = y; \)

Typical Latch Inference

- Conditional assignments are not completely specified
  - Check if the *else-clause* exists
- Outputs conditionally assigned in an if-statement are not assigned before entering or after leaving the if-statement

\[ \text{always@}(D \text{ or } S) \]
\[ \text{if}(S) \ Q = D; \]
\[ \text{Infer latch for Q} \]

\[ \text{always@}(S \text{ or } A \text{ or } B) \]
\[ \begin{array}{l}
\text{begin} \\
Q = A; \\
\text{if}(S) \ Q = B; \\
\text{end}
\end{array} \]
\[ \text{Do not infer latch for Q} \]
Typical Coding Style Limitation

always@(a or en)
if(en) x=a;

always@(a or en)
if(en) x=a;
else x=x;

Latch description

Wrong circuit!

Typical Coding Style Limitation

- Process by process
- No consideration on the dependencies across processes
- No warrantee on the consistency of memory semantics

Module EXP(in, s1, s2, o1, o2, o3);
input in, s1, s2;
output o1, o2, o3;
reg o1, o2, o3;
always@((in or s1 or o2)
"d1" if(s1) o1=in;
"d2" else o1=o2;
always@( s1 or s2 or o1) begin
"d3" o3=s1&s2;
"d4" if(s2) o2=o1;
end
endmodule

Infer a latch for o2

Asynchronous feedback loop

o1 depends on its value via o2 at d4
Terminology

- Clocked statement: edge-triggered always statement
  - Simple clocked statement
    e.g., `always @ (posedge clock)`
  - Complex clocked statement
    e.g., `always @ (posedge clock or posedge reset)`
- **Flip-flop inference** must be conducted only when synthesizing the **clocked statements**

Infer FF for Simple Clocked Statements

- Infer a flip-flop for each variable being assigned in the simple clocked statement

```verbatim
input a, b, s, clk;
output y, w;
reg x, w, y, z;
always @ (posedge clk)
begin
  /* d1 */ x = a;
  /* d2 */ if (s) y = x;
  /* d3 */ else   y = z;
  /* d4 */ z = b;
  /* d5 */ w = 1'b1;
end
```

Diagram:

- Used after defined connected to input of x
- Used before defined connected to output of z
Infer FF for Simple Clocked Statements

- Two post-processes
  - Propagating constants
  - Removing the flip-flops without fanouts

Infer FF for Complex Clocked Statements

- Require the following syntactic template
  - An if-statement immediately follows the always statement
  - Each variable in the event list except the clock signal must be a selective signal of the if-statements
  - Assignments in the blocks B1 and B2 must be constant assignments (e.g., x=1, etc.)

\[
\text{always } @ (\text{posedge } \text{clock or posedge } \text{reset or negedge } \text{set})
\]

\[
\text{if(reset) begin B1 end} \\
\text{else if !set begin B2 end} \\
\text{else begin B3 end}
\]
Typical Coding Style Limitation

```verbatim
always @ (posedge clk or posedge R)
if(R) Q = 0;
else Q = D;
```

Typical Tri-State Buffer Inference

- If a data object Q is assigned a high impedance value ‘Z’ in a multi-way branch statement (if, case, ?:)
  - Associated Q with a tri-state buffer
- If Q associated with a tri-state buffer has also a memory attribute (latch, flip-flop)
  - Have **Hi-Z propagation problem**
    - Real hardware cannot propagate Hi-Z value
    - Require two memory elements for the control and the data inputs of tri-state buffer
Typical Tri-state Buffer Inference

- It may suffer from mismatches between synthesis and simulation
  - Process by process
  - May incur the Hi-Z propagation problem

```verilog
reg QA, QB;
always @ (En or D)
  if(En) QA = D;
  else QA = 1'bz;
always @ (posedge clk)
  QB = QA;  // assignment can pass Hi-Z to QB
```

Comments on Special Element Inference

- Typical synthesizers
  - Use ad hoc methods to solve latch inference, flip-flop inference and tri-state buffer inference
  - Incur extra limitations on coding style
  - Do not consider the dependencies across processes
  - Suffer from synthesis/simulation mismatches

- A lot of efforts can be done to enhance the synthesis capabilities
  - It may require more computation time
  - Users’ acceptance is another problem
Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level optimization
  - Multi-level optimization
- Modeling for synthesis

Two-Level Logic Optimization

**Basic idea**: Boolean law $x + x' = 1$ allows for grouping $x1x2+x1 \cdot x'2 = x1$

Approaches to simplify logic functions:
- Karnaugh maps [Kar53]
- Quine-McCluskey [McC56]
3-Variable Karnaugh Maps

Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Implicant

Prime implicant:
- $a'b'c'$
- $ac'$
- $a'cd'$

Adjacency to 1's:
- $a'b'c'd'$
- $ab'c'$
- $abc'$
Minimum Form

- A sum-of-products expression containing a non-prime implicant cannot be minimum
  - Could be simplified by combining the nonprime term with additional minterm
- To find the minimum sum-of-products
  - Find a minimum number of prime implicants which cover all of the 1’s
  - Not every prime implicant is needed
  - If prime implicants are selected in the wrong order, a nonminimum solution may result

Multi-Level Logic

- Multi-level logic:
  - A set of logic equations with no cyclic dependencies
- Example: $Z = (AB + C)(D + E + FG) + H$
  - 4-level, 6 gates, 13 gate inputs
Multi-Level v.s. Two-Level

Two-level:
- Often used in control logic design
  \[ f_1 = x_1x_2 + x_1x_3 + x_1x_4 \]
  \[ f_2 = x_1'x_2 + x_1'x_3 + x_1x_4 \]
- Only \( x_1x_4 \) shared
- Sharing restricted to common cube

Multi-level:
- Datapath or control logic design
- Can share \( x_2 + x_3 \) between the two expressions
- Can use complex gates
  \[ g_1 = x_2 + x_3 \]
  \[ g_2 = x_2x_4 \]
  \[ f_1 = x_1y_1 + y_2 \]
  \[ f_2 = x_1'y_1 + y_2 \]
  \((y_i)\ is\ the\ output\ of\ gate\ g_i)\]
Multi-Level Logic Optimization

- Technology independent
  - Decomposition/Restructuring
    - Algebraic
    - Functional
  - Node Optimization
    - Two-level logic optimization techniques are used

Basic Operations

1. decomposition
   (single function)
   \[ f = abc + abd + (ac)'d' + b'c'd' \]
   \[ f = xy + (xy)' \]
   \[ x = ab \]
   \[ y = c + d \]

2. extraction
   (multiple functions)
   \[ f = (az + bz')cd + e \]
   \[ g = (az + bz')e' \]
   \[ h = cde \]
   \[ f = xy + e \]
   \[ g = xe' \]
   \[ h = ye \]
   \[ x = az + bz' \]
   \[ y = cd \]
Basic Operations

3. factoring
   (series-parallel decomposition)
   \[ f = ac + ad + bc + bd + e \]
   \[ f = (a + b)(c + d) + e \]

4. substitution
   (with complement)
   \[ g = a + b \]
   \[ f = a + bc + b'c' \]
   \[ f = g(a + c) + g'c' \]

5. elimination
   \[ f = ga + g'b \]
   \[ g = c + d \]
   \[ f = ac + ad + bc'd' \]
   \[ g = c + d \]

“Division” plays a key role!!

Division

- Division: \( p \) is a Boolean divisor of \( f \) if \( q \neq \phi \) and \( r \) exist such that \( f = pq + r \)
  - \( p \) is said to be a factor of \( f \) if in addition \( r = \phi : \)
    \[ f = pq \]
  - \( q \) is called the quotient
  - \( r \) is called the remainder
  - \( q \) and \( r \) are not unique

- Weak division: the unique algebraic division such that \( r \) has as few cubes as possible
  - The quotient \( q \) resulting from weak division is denoted by \( f \div p \) (it is unique)
Weak Division Algorithm

\text{Weak\_div}(f, p):
\begin{align*}
U &= \text{Set } \{u_j\} \text{ of cubes in } f \text{ with literals not in } p \text{ deleted} \\
V &= \text{Set } \{v_j\} \text{ of cubes in } f \text{ with literals in } p \text{ deleted} \\
/* \text{ note that } u_jv_j \text{ is the } j\text{-th cube of } f */ \\
V^i &= \{v_j \in V : u_j = p_j\} \\
q &= \bigcap V^i \\
r &= f - pq \\
\text{return}(q, r)
\end{align*}

Example
\begin{align*}
f &= \text{acg + adg + ae + bc + bd + be + a'b} \\
p &= \text{ag + b} \\
U &= \text{ag + ag + a + b + b + b + b} \\
V &= \text{c + d + e + c + d + e + a'} \\
V^a &= \text{c + d} \\
V^b &= \text{c + d + e + a'} \\
q &= c + d = f/p
\end{align*}
Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level optimization
  - Multi-level optimization
- Modeling for synthesis

Non-Synthesizable Verilog Constructs

- initial
- Loops
  - repeat
  - forever
  - while
- Data types
  - event
  - real
  - time
- UDPs
- fork …join blocks
- procedural assignments
  - assign and deassign
  - force and release
  - disable
- Some operators
  - / and %
  - === and !==

Not commonly supported by synthesis tools !!
Typical HDL Synthesizer Directives

- Directives: special case of regular comments
  - Ignored by the simulator
  - Affect the synthesis results
- Turn translator off by using
  // synopsys translate_off
  /* synopsys translate_off */
- Turn translator on by using
  // synopsys translate_on
  /* synopsys translate_on */

case

- The first case-item that evaluates to true determines the path
- Full case
  - HDL compiler directive
  - Use " synopsys full_case "
- Parallel case
  - if HDL Compiler can statically determine or
  - Use " synopsys parallel_case "
case Examples

- full and parallel
  
  ```
  input [1:0] a;
  always @ (a or w or x or y or z)
  begin
    case (a)
      2'b11: b = w;
      2'b10: b = x;
      2'b01: b = y;
      2'b00: b = z;
    endcase
  end
  ```

- parallel but not full
  
  ```
  input [1:0] a;
  always @ (a or w or z)
  begin
    case (a)
      2'b11: b = w;
      2'b00: b = z;
    endcase
  end
  ```

case Examples (cont’d)

- Not full nor parallel
  
  ```
  always @ (w or x) begin
    case (2'b11)
      w:
        b = 10;
      x:
        b = 01;
    endcase
  end
  ```
parallel_case

- //synopsys parallel_case
  - Affect the way logic is generated for the case statement
  - Generate mux logic instead of priority encoder
  - Used when only one case-item is executed

```verilog
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
                state3 = 4'b0100, state4 = 4'b1000;
case (1) //synopsys parallel_case
  current_state[0] : next_state = state2;
case (1) //synopsys parallel_case
  current_state[1] : next_state = state3;
case (1) //synopsys parallel_case
case (1) //synopsys parallel_case
  current_state[3] : next_state = state1;
endcase
```

full_case

- //synopsys full_case
  - Asserts all possible clauses of a case statement have been covered
  - Can avoid the need for default logic
  - Avoid latch inference

```verilog
reg[1:0] in, out;
reg[3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
                state3 = 4'b0100, state4 = 4'b1000;
case (in) //synopsys full_case
  0: out = 2;
  1: out = 3;
  2: out = 0;
endcase
```
full_case (cont’d)

```markdown
reg [1:0] state, next_state;
//synopsys state vector state
always @ (state or in) begin
  case (state) //synopsys full_case
    0: begin
      out = 3;
      next_state = 1;
      end
    1: begin
      out = 2;
      next_state = 2;
      end
    2: begin
      out = 1;
      next_state = 3;
      end
    3: begin
      out = 0;
      if (in)
        next_state = 0;
      else
        next_state = 3;
      endcase
    end
  always @ (posedge clock)
    state = next_state;
end
```

Directives for Register Inference

- `async_set_reset : use //synopsys async_set_reset`

```markdown
module async_set_reset (reset, set, d, gate, y, t);
input reset, set, gate, d;
output y, t;
//synopsys async_set_reset “reset, set”
reg y, t;
always @ (reset or set)
  begin : direct_set_reset
    if(reset)
      y = 1'b0;  //asynchronous reset
    else if(set)
      y = 1'b1;  //asynchronous set
  end
  always @ (gate or reset) // for set (gate or reset)
    if (reset)  //for set : if (set)
      t = 1'b0;  // for set : t=1'b1
    else if (gate)
      t = d;
  endmodule
```
Directives for Register Inference

- `async_set_reset`

```
module sync_set_reset ( clk, reset, set, d1, d2, y, t );
input  clk, reset, set, d1, d2;
output y, t;
// synopsys sync_set_reset "reset, set"
reg y, t;
always @ (posedge clk)
begin : synchronous_reset
  if (reset)
    y=1'b0;    // synchronous reset
  else
    y=d1;
end
always @ (posedge clk)
begin : synchronous_set
  if (set)
    t=1'b1;      // synchronous set
  else
    t=d2;
end
endmodule
```
Directives for Register Inference

- **One_hot**
  - No more than one signal is active high
  - use //synopsys one_hot "object_name"

```verilog
class one_hot_example
  (reset,set,reset2,set2,y,t);
  input reset,set,reset2,set2;
  output y,t;
  //synopsys async_set_reset "reset,set"
  //synopsys async_set_reset "reset2,set2"
  //synopsys one_hot "reset,set"
  reg y,t;

  always @ (reset or set)
  begin : direct_set_reset
       if (reset)
           y=1’b0; // asynchronous reset by “reset”
       else if(set)
           y=1’b1; // asynchronous set by “set”
  end

  always @ (reset2 or set2)
  begin : direct_set_reset_too
       if (reset2)
           t=1’b0; // asynchronous reset by “reset2”
       else if(set2)
           t=1’b1; // asynchronous set by “~reset2 set2”
  end

  // synopsys translate_off
  always @(reset or set)
  if (reset & set)
     $write("ONE_HOT violation for ‘reset’, ‘set’.");
  //synopsys translate_on
endmodule
```

Directives for Register Inference

- **One_cold**
  - No more than one signal is active low
  - use //synopsys one_cold "object_name"

```verilog
class one_cold (reset,set,reset2,set2,y,t);
  input reset,set,reset2,set2;
  output y,t;
  //synopsys async_set_reset "reset,set"
  //synopsys async_set_reset "reset2,set2"
  //synopsys one_cold "reset,set"
  reg y,t;

  always @ (reset or set)
  begin : direct_set_reset
       if (~reset)
           y=1’b0; // asynchronous reset by “~reset”
       else if(~set)
           y=1’b1; // asynchronous set by “~set”
  end

  always @ (reset2 or set2)
  begin : direct_set_reset_too
       if (~reset2)
           t=1’b0; // asynchronous reset by “~reset2”
       else if(~set2)
           t=1’b1; // asynchronous set by “reset2 ~set2”
  end

  // synopsys translate_off
  always @(reset or set)
  if (~reset &  ~set)
     $write("ONE_COLD violation for ‘reset’, ‘set’.");
  //synopsys translate_on
endmodule
```
**Multiplexer Inference**

```verilog
module mux8to1(DIN,SEL,DOUT);
  input [7:0] DIN;
  input [2:0] SEL;
  output DOUT;
  reg DOUT;
//synopsys infer_mux “mux_blk”
  always @(SEL or DIN)
  begin: mux_blk
    case (SEL)
      3'b000: DOUT <= DIN[0];
      3'b001: DOUT <= DIN[1];
      3'b010: DOUT <= DIN[2];
      3'b011: DOUT <= DIN[3];
      3'b100: DOUT <= DIN[4];
      3'b101: DOUT <= DIN[5];
      3'b110: DOUT <= DIN[6];
      3'b111: DOUT <= DIN[7];
    endcase
  end
endmodule
```

**Resource Sharing**

- Assigning similar operations to a common netlist cells
  - Can reduce hardware
  - May degrade performance

- Possible sharing
  * + -
  > >= < <=
  - sharing within the same always block
Control Flow Conflict

- Two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations.

```
always begin
  Z1 = A + B;
  if(COND_1)
    Z2 = C + D;
  else begin
    Z2 = E + F;
    if(COND_2)
      Z3 = G + H;
    else
      Z3 = I + J;
  end
  if( ! COND_1)
    Z4 = K + L;
  else
    Z4 = M + N;
end
```

Data Flow Conflict

- No sharing if causes a combinational loop.

```
always @ (A or B or C or D or E or F or Z or ADD_B)
begin
  if (ADD_B) begin
    Temp_1 = A + B;
    Z = Temp_1 + C;
  end
  else begin
    Temp_2 = D + E;
    Z = Temp_2 + F;
  end
end
```
Resource Sharing Methods

- **Automatic**
  - HDL compiler identifies the possibility
  - Logic optimizer minimizes the area while meeting the timing constraints
    
    ```verilog
    always @ (A or B or C or ADD_B)
    begin
      if (ADD_B)
        Z = B + A;
      else
        Z = A + C;
    end
    ```
    - 2 adders with a output mux (sharing)
    - 1 adder with 1 input mux (no sharing)

Resource Sharing Methods (cont'd)

- **Manual**
  - Synthesizer provides some directives
  - Force the sharing of specified operations
  - Prevent the sharing of specified operations
  - Force specified operations to use a particular type of resource
  - Only in a named combinational block

```verilog
module TWO_ADDS_6 (A, B, C, Z, ADD_B);
  input [3:0] A,B,C;
  input ADD_B;
  output [3:0] Z;
  reg[3:0] Z;
  always @ (A or B or C or ADD_B) begin : b1
    /*synopsys resource r0; ops = "A1 A2" */
    if (ADD_B)
      Z= A + B; //synopsys label A1
    else
      Z= A + C; //synopsys label A2
  end
endmodule
```

- Declare an identifier for an individual resource.
  - resource r0;
- Place labels on the operations.
  - if (ADD_B)
    - Z= A + B; //synopsys label A1
    - else
    - Z= A + C; //synopsys label A2
  - endif;
- Use the ops directive to bind the labeled operations to the resource they share.
  - ops = "A1 A2";
Explicit Resource Sharing

- Original
  
  if (F)
  A=B+C;
  else
  A=B+20;

- Modified
  
  if(F)
  T=C;
  else
  T=20;
  A=B+T;

Using Parenthesis

A=B+C+D+F

A=(B+C)+(D+F)