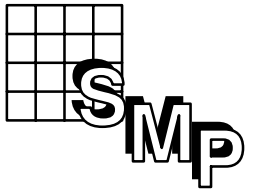

Synopsys Methodology Program

Writing Successful RTL Descriptions in Verilog

Mike Parkin, Sun Microsystems, Inc.

This paper discusses a few techniques for writing successful RTL descriptions in Verilog®. While Verilog is the language used in the examples, some of the ideas are also applicable to other HDL languages, such as VHDL.

The use of structure to control the synthesis process is the main ingredient for success when writing RTL descriptions. Structure is defined as the way in which parts are arranged or put together to form a whole, and it is created through the use of modules and cell instantiation. The designer should code the RTL to reflect the desired hardware structure. Why use Verilog HDL instead of schematic capture? The main reason is to make the logic designer more productive. Quality is also improved because more time can be spent on logic verification rather than on the detailed gate-level implementation. And a more concise and readable design makes it easier for the design to be reused by others.

Optimized functions such as decoders, adders, and multi-bit multiplexers become building blocks for success. These functions can be hand-crafted by the designer and then reused in many places, or they can be referenced or instantiated from the DesignWare™ library supplied by Synopsys.

In order to insure the gate-level design behaves the same as the RTL version, it is important to understand common pitfalls in coding Verilog that can cause an RTL/gate-level simulation discrepancy.

Improving the Quality of Results

The quality of the synthesized design can be improved using the following techniques:

- Module partitioning
- Adding structure
- Horizontal partitioning
- Adding hierarchy (vertical partitioning)
- Performing operations in parallel
- Use multiplexers for logic implementation

Module Partitioning

Where possible, register module outputs and keep the critical path in one block.

Keep as much of the critical path in one module or block as possible. This enables Design Compiler™ to optimize the critical path while it is compiling a single module or block without having to iterate between several different modules. Placing the registers on module outputs also simplifies the compilation process because timing budgets for registered module outputs are not needed. Registering module inputs does not yield much improvement since the input arrival times can be computed and output using the characterize and write_script commands. The size of a module should be based on a logical partitioning as opposed to an arbitrary gate count. Instantiating a set of pre-compiled basic building blocks can reduce the complexity of the design and the associated compile effort even for larger modules. In this case, a large percentage of the gate count ends up in the instantiated or inferred modules. The last point is to keep most of the logic in the leaf modules. This simplifies the compilation process because the top-level modules will need little or no compilation and constraints can more easily be propagated down the hierarchy.

The following design examples were compiled using Synopsys Design Compiler version 2.2b, the LSI LCA200k library, and the “B3X3” wireload model with WCCOM operating conditions. The constraints were:

```
set_load   5 * load_of (IV/A) all_outputs()
set_drive  drive_of (IV/Z) all_inputs()
```

Adding Structure

One goal of logic synthesis is to produce an optimal netlist that is independent of the original structure. Until this goal is achieved, controlling the structure of a logic description is one of the best ways to ensure an optimal implementation. Controlling the structure by using separate assignment statements and through the use of parentheses really has very little effect on

the generated logic. The only case where parentheses have a significant effect is when resources are used. Resources and function invocations are assigned and preserved when the HDL code is read and have a significant impact on the generated logic.

Note: In v3.0 there is tree-height minimization of expressions. The following behavioral code specifies a 32-bit arithmetic shift right operation:

```
assign shifta = {{31{a[31]}}, a} >> shiftCnt;
```

An iteration value of 31 is the largest value that is required. For smaller shift amounts the extra bits will be truncated when assigned to the variable shifta, which is 32 bits wide. This expression produces a design of 742 gates that is almost 50% slower than the structural logic design. The shift right arithmetic function can also be described without using any extra bits:

```
// 16.63 ns, 1431 gates
assign shifta = (a[31] ? ((a >> shiftCnt) |
 (((32'b1 << shiftCnt) - 1) << (32-shiftCnt))) : a >> shiftCnt);
```

This arithmetic shift operation shifts the 32 bits right by the shift count, and replaces the vacated positions with bit 31 of variable a. The expression `(a >> shiftCnt)` shifts "a" by the shift count but doesn't replace the vacated positions. The expression `((32'b1 << shiftCnt) - 1)` produces a string of 1s equal in length to the value of the shift count, which is equal to the number of vacated bit positions. This string of 1s needs to occupy the vacated bit positions starting with bit 31. The expression `(32-shiftCnt)` is the number of bit positions that the string of 1s needs to be left shifted. The final result is the logical OR of the shifted string of 1s and the logical right shift value `(a>>shiftCnt)`. While this expression is equivalent, it is much too complex to be practical. When synthesized and optimized, this specification produces a design with 1,431 gates that is three times slower and over twice the area of the structural logic design (see Figure 1):

```
// structured shift right arithmetic design
// 6.87 ns, 613 gates, fastest, optimal version
module shiftRightAr(a, shiftCnt, z);
input [31:0] a;
input [4:0] shiftCnt;
output [31:0] z;
wire [31:0] d0, d1, d2, d3, notA;
assign notA = ~a;

mux2to1L32 m21_0 (notA,{notA[31], notA[31:1]}, shiftCnt[0], d0);
mux2to1L32 m21_1 (d0,{{ 2{a[31]}}, d0[31:2]}, shiftCnt[1], d1);
mux2to1L32 m21_2 (d1,{{ 4{notA[31]}}, d1[31:4]}, shiftCnt[2],
d2);
```

```
mux2to1L32 m21_3 (d2,{{ 8{a[31]}}, d2[31:8]}, shiftCnt[3], d3);
mux2to1L32 m21_4 (d3,{{16{notA[31]}}, d3[31:16]},shiftCnt[4],
z);
endmodule

module mux2to1L32 (a, b, s, z);
input [31:0] a, b;
input s;
output [31:0] z;
     assign z = ~(s ? b : a);
endmodule
```

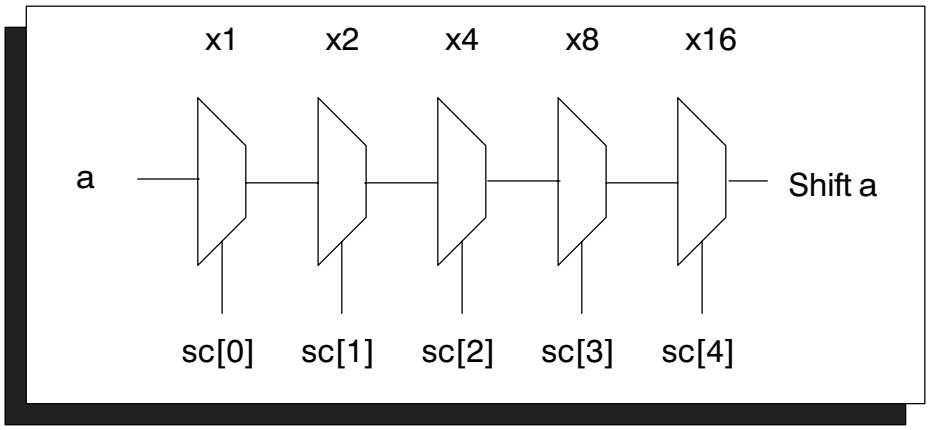

Figure 1. Structured Shift Right Arithmetic Design

The structural logic design produces a design of 613 gates that is three times faster. While each of the 32-bit multiplexers is instantiated, the module mux2to1L32 is defined without using gate-level instantiation. The shifter may also be specified without the mux instantiations:

```
// 8.00 ns, 518 gates, with no structuring
// 8.55 ns, 598 gates, with structuring
// smallest design but 20% slower than optimal
module shiftRightAr(a, shiftCnt, z);
input [31:0] a;
input [4:0] shiftCnt;
output [31:0] z;
wire [31:0] d0, d1, d2, d3, notA;
assign notA = ~a;

assign
     d0 = (shiftCnt[0] ? {a[31], a} >> 1 : a),
```

```
        d1 = ~(shiftCnt[1] ? {{2{a[31]}}, d0} >> 2 : d0),
        d2 = ~(shiftCnt[2] ? {{4{notA[31]}}, d1} >> 4 : d1),
        d3 = ~(shiftCnt[3] ? {{8{a[31]}}, d2} >> 8 : d2),
        z = ~(shiftCnt[4] ? {{16{notA[31]}}, d3} >> 16 : d3);
endmodule
```

Now the logic synthesis tool is free to optimize various pieces of each multiplexer. This specification, when compiled without structuring, produces a design that is only 518 gates, but is also 20% slower. With the default structuring enabled the resultant design is actually bigger and slower, a case where no structuring is a big win. Furthermore, due to the optimizations, the symmetry is lost, making default structuring unattractive for use in a datapath design.

Horizontal Partitioning

A combinational circuit can be expressed as a sum of products that can be implemented by a circuit with two levels of logic. However this may result in gates with a maximum fan-in equal to the number of inputs. By building up the logic in levels, a circuit with a a regular structure can be designed. In addition, the circuit can broken into horizontal slices to minimize the maximum fan-in of a logic gate. A carry lookahead adder is a classic example. A 32-bit adder is broken into eight 4-bit blocks. Each adder block generates a `group propagate` and `group generate` signal. A carry-lookahead block takes as input the `group generate` and `propagate` signals and generates a carry-in to each block and a carry-out for the entire adder.

A 32-bit priority encoder is another example where bit-slicing can yield significant results. The first priority encoder, `priorityEncode32b`, compiles to produce a design of 205 gates. The critical path consists of seven levels of logic. The second module, `priorityEncode32`, is restructured using four 8-bit blocks (see Figure 2). The restructured priority encoder compiles to 182 gates and four levels of logic. The worst-case delay is reduced by 26%, while the gate count is reduced by 23. This restructuring reduces the scope and complexity of the problem from 32 bits to 8 bits, which allows the HDL compiler to produce a more optimal design.

```
// 7.14 ns, 205 gates (original design)
module priorityEncode32b (bitVector, adr, valid);
input [31:0] bitVector;

output [4:0] adr;
output valid;
```

```
function [4:0] encode;
input [31:0] in;
integer i;
begin: _encode
     encode = 5'd0;
     for ( i=31; i>=0; i=i-1 )
     if ( !in[i] ) begin
          encode = i;
          disable _encode;
     end
end
endfunction

assign adr = encode(bitVector);
assign valid = ~(&bitVector);
endmodule

// 5.31 ns, 182 gates (smaller/faster design)
module priorityEncode32 (bitVector, adr, valid);
input [31:0] bitVector;
output [4:0] adr;
output valid;
// synopsys dc_script_begin,
// dont_touch -cell {pe0 pe1 pe2 pe3}
// synopsys dc_script_end

wire [3:0] anyValid;
wire [2:0] adr0, adr1, adr2, adr3, adr4, afAdr;
wire [2:0] msgAdr, lsgAdr;
wire msaf = |anyValid[3:2];

// partition into 8-bit groups for optimal speed/gate-count
priorityEncode8 pe0 (bitVector[7:0], adr0, anyValid[0]);
priorityEncode8 pe1 (bitVector[15:8], adr1, anyValid[1]);
priorityEncode8 pe2 (bitVector[23:16], adr2, anyValid[2]);
priorityEncode8 pe3 (bitVector[31:24], adr3, anyValid[3]);

// select most significant group using valid bits
assign msgAdr = anyValid[3] ? adr3 : adr2;
assign lsgAdr = anyValid[1] ? adr1 : adr0;
assign afAdr = msaf ? msgAdr : lsgAdr;
assign adr = {msaf, msaf ? anyValid[3] : anyValid[1], afAdr};
assign valid = |anyValid;

endmodule

module priorityEncode8 (in, out, anyValid);
```

```
input [7:0] in;
output [2:0] out;
output anyValid;

function [2:0] encode;
input [7:0] in;
integer i;
begin : _encode
      encode = 3'd0;
      for ( i=7; i>=0; i=i-1 )
      if ( !in[i] ) begin
            encode = i;
            disable _encode;
      end
end
endfunction

assign out = encode(in);
assign anyValid = !(&in);
endmodule
```

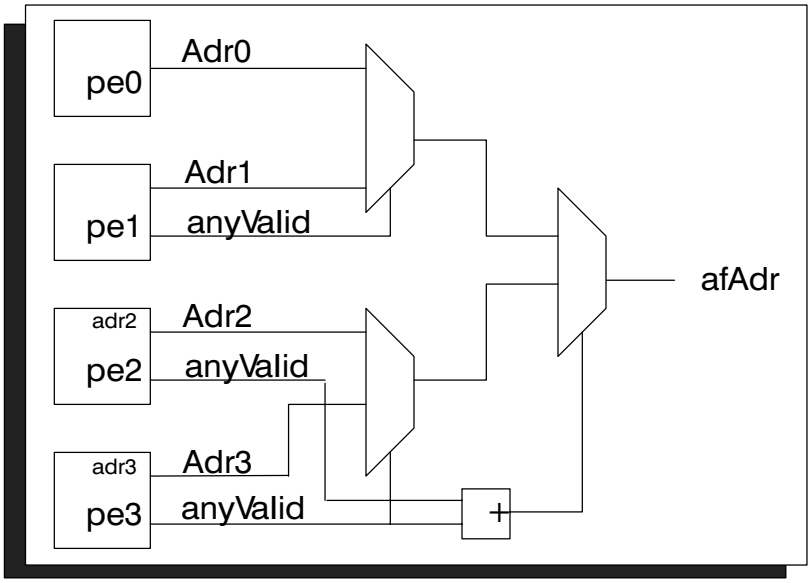

Figure 2. 32-Bit Fast Priority Encoder

Adding Hierarchy

Collapsing the hierarchy results in more efficient synthesis in some cases. In other cases, adding hierarchy can improve the design. This was shown in the case of the shift right arithmetic function described earlier. In the case of the priority encoder, additional hierarchy achieved significant improvements in both speed and area. Another case where additional

hierarchy achieves significant results is in the balanced tree decoder described in the following example. Adding the hierarchy in these cases helps to define the final implementation and preserves the structure that yields an optimal design. A 32-bit decoder with negative asserting output can be coded as:

```
// decoder using variable array index
module decoder32V1(adr, decode);
input [4:0] adr;
output [31:0] decode;
reg [31:0] decode; // note: pseudo_reg

always @(adr) begin
      decode = 32'hffffffff;
      decode[adr] = 1'b0;
end
endmodule
```

This design turns out to be the least-efficient implementation of several alternative designs. It compiles to 125 gates.

A more concise representation of a decoder is given as

```
// decoder using shift operator
module decoder32V2(adr, decode);
input [4:0] adr;
output [31:0] decode;
assign decode = ~(1'b1 << adr);
endmodule
```

This produces a slightly faster design of 94 gates. More dramatic results can be obtained by using a balanced tree decoder. By adding a second level of hierarchy, a balanced tree decoder can be specified (see Figure 3).

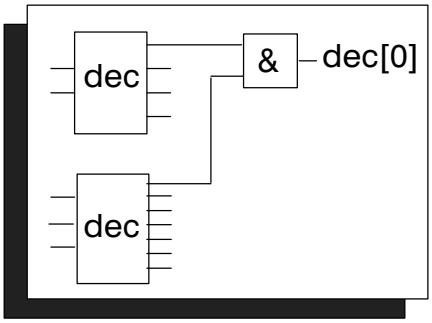

Figure 3. Balanced Tree Decoder

```
// balanced tree decoder: smaller and faster
//
module decoder32BT (adr, dec);
input [4:0] adr;
output [31:0] dec;

wire [3:0] da = 1'b1 << adr[1:0]; // 2 to 4 decoder
wire [7:0] db = 1'b1 << adr[4:2]; // 3 to 8 decoder

decode32L2 d32l2 (da, db, dec);
endmodule

module decode32L2(da, db, dec);
input [3:0] da;
input [7:0] db;
output [31:0] dec;

wire [31:0] dbVec =
     {{4{db[7]}}, {4{db[6]}}, {4{db[5]}}, {4{db[4]}},
     {4{db[3]}}, {4{db[2]}}, {4{db[1]}}, {4{db[0]}}};
wire [31:0] daVec = {8{da}};

assign dec = ~(dbVec & daVec);
endmodule
```

This design compiles to 68 gates, which is about 50% smaller than the `decoder32V1` module, and it is the fastest of the three modules.

Performing Operations in Parallel

This is the classic technique of using more resources to achieve a decrease in speed. In this example, an array of four 6-bit counters are compared. The output is the index of the smallest counter. Various search strategies are used to find the smallest element: linear search, binary search, and parallel search.The first implementation uses a `task` and a `for` loop to compare all the values. This results in a serial compare. First `cntr[0]` is compared to `cntr[1]`, the smallest `cntr` is selected and then compared to `cntr[2]`. The smallest result from the second comparator is selected and compared to `cntr[3]`. This process involves three comparators and two multiplexers in series for a total delay of 22.41 ns and 18 levels of logic. The total area is 527 gates.

```
// Linear Search - 22.41 ns, 527 gates
module arrayCmpV1(clk, reset, inc, index, min);
input clk, reset, inc;
input [1:0] index;
```

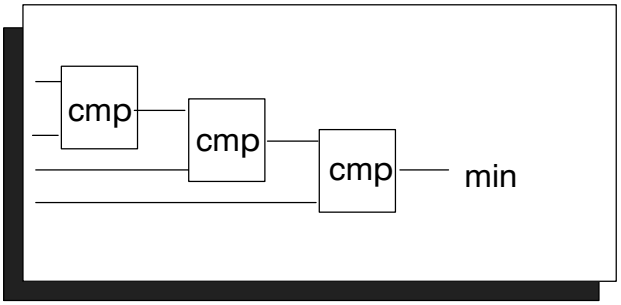

Figure 4. Linear Search

```
output [1:0] min;

reg [5:0] cntr[0:4];
reg [1:0] min;  // pseudo register
integer i;

// compare each array element to mincount
task sel;
     output [1:0] sel;
     reg [5:0] mincount;
     begin : _sc
          mincount = cntr[0];
          sel = 2'd0;
          for ( i = 1; i <= 3; i=i+1 )
               if ( cntr[i] < mincount ) begin
                    mincount = cntr[i];
                         sel = i;
               end
     end
endtask

always @(cntr[0] or cntr[1] or cntr[2] or cntr[3])
 sel(min);

always @(posedge clk)
     if (reset)
          for( i=0; i<=3; i=i+1 )
               cntr[i] <= 6'd0;
     else if (inc)
          cntr[index] <= cntr[index] + 1'b1;
endmodule
```

A second version of this design needs two comparators in series and takes 14.9 ns with eleven levels of logic and a total area of 512 gates. This design is both smaller and faster than the first version.

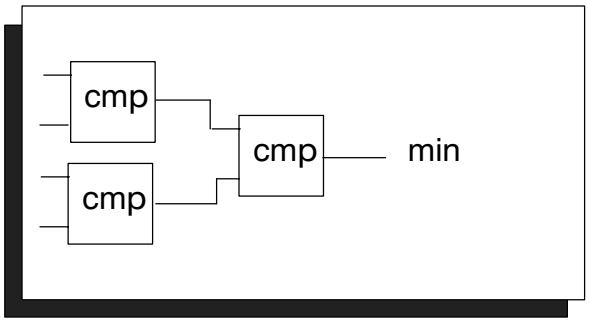

Figure 5. Binary Search

```
// Binary Search - 14.9 ns, 512 gates (smallest area)
module arrayCmpV2(clk, reset, inc, index, min);
input clk, reset, inc;
input [1:0] index;

output [1:0] min;

reg [5:0] cntr[0:4];
integer i;

// binary tree comparison
wire c3lt2 = cntr[3] < cntr[2];
wire c1lt0 = cntr[1] < cntr[0];

wire [5:0] cntr32 = c3lt2 ? cntr[3] : cntr[2];
wire [5:0] cntr10 = c1lt0 ? cntr[1] : cntr[0];

wire c32lt10 = cntr32 < cntr10;

// select the smallest value
assign min = {c32lt10, c32lt10 ? c3lt2: c1lt0};

always @(posedge clk)
     if (reset)
          for( i=0; i<=3; i=i+1 )
               cntr[i] <= 6'd0;
     else if (inc)
          cntr[index] <= cntr[index] + 1;
endmodule
```

A third implementation performs all the comparisons in parallel. The same path now takes 11.4 ns with eight levels of logic and has a total area of 612 gates. This version is about 20% faster than the second version, with a 20% increase in area.

Exploit parallelism to improve speed.

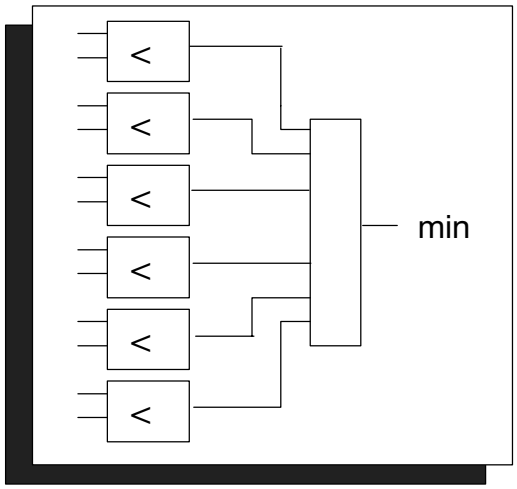

Figure 6. Parallel Search

```
// Parallel Search - 11.4 ns, 612 gates (fastest design)
module arrayCmpV3(clk, reset, inc, index, min);
input clk, reset, inc;
input [1:0] index;

output [1:0] min;

reg [5:0] cntr[0:4];
integer i;

// compare all counters to each other
wire l32 = cntr[3] < cntr[2];
wire l31 = cntr[3] < cntr[1];
wire l30 = cntr[3] < cntr[0];
wire l21 = cntr[2] < cntr[1];
wire l20 = cntr[2] < cntr[0];
wire l10 = cntr[1] < cntr[0];

// select the smallest value
assign min = {l31&l30 | l21&l20, l32&l30 | l10&~l21};

always @(posedge clk)
```

```
        if (reset)
             for( i=0; i<=3; i=i+1 )
                  cntr[i] <= 6'd0;
        else if (inc)
             cntr[index] <= cntr[index] + 1;
endmodule
```

Use Multiplexers for Logic Implementation

When using CMOS technology, there is a significant speed advantage in using pass-gate multiplexers. Where possible, rewrite the code to take advantage of the muxes. The following example is for the condition code logic of a SPARC® processor.

```
// Using and-or logic - 5.50 ns, 52 gates
module condcodeV1 (cc, bc, valid);
input [3:0] cc;
input [3:0] bc;
output valid;

wire n, z, v, c;
wire [15:0] bcdec;

assign {n,z,v,c} = cc;
assign bcdec = 1'b1 << bc;
assign valid = |(bcdec & {!v, !n, !c, !(c | z), !(n ^ v), (!(z |
(n ^ v))),
 !z, 1'b1, v, n, c, c | z, n ^ v, z | (n ^ v), z,
1'b0});
endmodule
```

The following code takes advantage of some symmetry in the specification and uses the branch condition as the index for an 8-to-1 multiplexer. This turns out to be the most optimal design in terms of both area and speed.

```
// Mux version - 5.17 ns, 30 gates
module condcodeV2 (cc, bc, valid);
input [3:0] cc;
input [3:0] bc;
output valid;

wire n, z, v, c;
wire [7:0] ccdec;

assign {n, z, v, c} = cc;
```

```
assign ccdec = {v, n, c, c | z, n ^ v, z | (n ^ v), z, 1'b0};
assign valid = bc[3] ^ ccdec[bc[2:0]];
endmodule
```

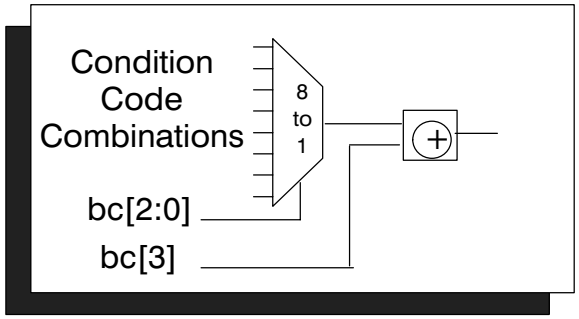

Figure 7. Condition Code Logic Using Mux

Vendor-Independent HDL

Technology-transparent design makes it easy to target a design to a specific vendor. This provides flexibility in choosing a primary vendor and in selecting an alternative source, if one is desired. Since it is unlikely that two vendors will have exactly the same macro library or process, the design will have to be resynthesized, instead of just translated using the new library. The main way to achieve a vendor-independent design is to avoid instantiating vendor macrocells in the design. Another technique is to create a small set of user defined macrocells and define these in terms of the vendor library. One set of these user-defined macros need to be created for each vendor. Using module definitions for simple components often achieves the same result without instantiating vendor cells:

```
module mux4to1x2 (in0, in1, in2, in3, sel, out);
input [1:0] in0, in1, in2, in3;
input [1:0] sel;
output [1:0] out;

mux4to1 m4_0 (in0[0], in1[0], in2[0], in3[0], sel, out[0]);
mux4to1 m4_1 (in0[1], in1[1], in2[1], in3[1], sel, out[1]);
endmodule

// Module mux4to1 takes the place of vendor cell
module mux4to1 (in0, in1, in2, in3, sel, out);
input in0, in1, in2, in3;
input [1:0] sel;
output out;
wire [3:0] vec = {in3, in2, in1, in0};
```

```
assign out = vec[sel];
endmodule
```

In Design Compiler v3.0 you may now use vendor-independent cells from the GTECH (for generic technology) library. There is a 4-to-1 mux in the library, called the GTECH_MUX4, that can replace the mux4to1 in the previous example. The command `set_map_only` can be used to instruct the compiler to select and preserve an equivalent cell from the technology library. This is the only way to insure the equivalent cell will be selected from the target library. Currently there is no Verilog simulation library for the GTECH components but a library can easily be created by changing the names in an existing library.

```
// GTECH version of mux4to1x2
// use of GTECH components makes this vendor independent
// set_map_only command locks in equivalent cells from target
lib

module mux4to1x2 (in0, in1, in2, in3, sel, out);
input [1:0] in0, in1, in2, in3;
input [1:0] sel;
output [1:0] out;

GTECH_MUX4 m4_0 (in0[0], in1[0], in2[0], in3[0],
     sel[0], sel[1], out[0]);
GTECH_MUX4 m4_1 (in3[1], in2[1], in1[1], in0[1],
     sel[0], sel[1], out[1]);
endmodule
```

Using Verilog Constructs

Don’t-Care Conditions

A design must be flattened or compiled using Boolean optimization in order to use dont_cares.

Don’t-care conditions for Synopsys can be specified by assigning an output to ‘bx. Dont_cares are usually found inside a case statement where some input combinations should never occur, or where outputs are dont_cares. While the meaning of the don’t-care specification is consistent between the simulation and synthesis tools, it is treated differently. When an output is assigned a value of ‘bx in Verilog, the output becomes unknown. Synopsys treats the assignment of ‘bx as a don’t-care specification. It will use this don’t-care specification to minimize the synthesized logic. The gate-level design will produce an output whose value,

while not undefined, is dependent on the particular logic generated. If the default clause in a case statement specifies don't-care conditions that should never occur, the Verilog model will produce an unknown output when the default statement is executed. If this output is used, unknowns will propagate to other parts of the design and the error will be detected. If the default clause of a case statement specifies a known value and there are no overlapping case items (when parallel case is used), then the RTL version will match the synthesized gate-level version. The proper use of dont_cares should not produce any significant simulation discrepency. In Synopsys v3.0 or earlier, in order to utilize the don't-care conditions in the optimization phase, the design must either be flattened, compiled using Boolean optimization, or the state table must be extracted before compilation.

Put don't care assignments in a lower-level block so flattening can work.

For this reason the designer may want to partition the design such that don't-care assignments are in a lower-level block, so flattening can work. This is necessary for the FSM extract command.

Procedural Assignment

Use the non-blocking construct for procedural assignments to state regs.

With the blocking assignment operator, statements are executed sequentially. In the PC chain example below, all the PC registers would get the value of fetchAdr.

```
always @(posedge clk) if (~hold) begin // pc chain
     fetchPC = fetchAdr;
     decPC = fetchPC;
     execPC = decPC;
     writePC = execPC;
end
```

This problem can be avoided by using fork and join in addition to an intra assignment delay. When using Synopsys, the fork and join must be bypassed using compiler directives.

```
always @(posedge clk) if (~hold) // pc chain
/* synopsys translate_off */ fork /* synopsys translate_on */
     fetchPC = #d2q fetchAdr;
     decPC = #d2q fetchPC;
     execPC = #d2q decPC;
     writePC = #d2q execPC;
/* synopsys translate_off */ join /* synopsys translate_on */
```

The non-blocking procedural assignment "<=" allows multiple procedural assignments to be specified within the same `always` block. Furthermore, it will simulate without the need for an intra-assignment delay.

```
reg [31:0] fetchPC, decPC, execPC, writePC;
always @(posedge clk) if (~hold) begin // pc chain
     fetchPC <= fetchAdr;
     decPC <= fetchPC;
     execPC <= decPC
     writePC <= execPC;
end
```

The pc-chain can also be expressed as a single shift-register-like statement:

```
always @(posedge clk) if (~hold) // pc chain
{fetchPC,decPC,execPC,writePC}<={fetchAdr,fetchPC,decPC,execPC};
```

Don't mix blocking and non-blocking assignments in the same block.

Mixing blocking and non-blocking assignments in the same block is not allowed and will result in a Synopsys Verilog HDL Compiler™ read error.

Using Functions with Component Implication

Function invocations can be used instead of module instantiations. This is really a question of style, not a recommendation. The function invocation can also be mapped to a specific implementation using the map_to_module compiler directive. The compiler directives `map_to_module` and `return_port_name` map a function or a task to a module. For the purposes of simulation, the contents of the function or task are used, but for synthesis the module is used in place of the function or task. This allows component instantiation to be used in a design for optimization purposes without altering the behavioral model. When multiple outputs are required, use a task instead of a function. When using component implication, the RTL-level model and the gate-level model may be different. Therefore the design can not be fully verified until simulation is run on the gate-level design. For this reason, component implication should be used with caution or not at all. The following code illustrates the use of component implication:

```
function mux8; // 8to1 mux
// synopsys map_to_module mux8to1
// synopsys return_port_name out
```

```
input [7:0] vec;
input [2:0] sel;
      mux8 = vec[sel];
endfunction

function mux32; // 32to1 mux
// synopsys map_to_module mux32to1
// synopsys return_port_name out
input [31:0] vec;
input [4:0] sel;
      mux32 = vec[sel];
endfunction

wire valid = ~(a[5] ? mux8(v8, a[4:2]) : mux32(v32, a[4:0]));
```

Register Inference

Using the Synopsys Verilog HDL compiler, flip-flops can be inferred by declaring a variable of type `reg` and by using this variable inside an `always` block that uses the `posedge` or `negedge clk` construct. The following example will generate an 8-bit register that recirculates the data when the `loadEnable` signal is not active:

```
reg [7:0] Q;
always @(posedge clk)
      if (loadEnable)
            Q <= D;
```

All variables that are assigned in the body of an `always` block must be declared to be of type `reg` even though a flip-flop will not be inferred. These "`reg`s" are really just wires, and for the sake of clarity should be commented as such. Please note that this is a Verilog language feature and not a Synopsys limitation.

Register inference allows designs to be technology-independent. Much of the design functionality can be specified using the more easily understood procedural assignment. There is no need to specify default assignments to prevent latch inference. If a design is free of component instantiations, then a gate-level simulation library is not required. Therefore, the design can be targeted to any vendor library without the need to recode macro libraries. The instance names of `reg`s are "`name_reg`", which allows easy identification for layout. Scan-string stitching can key off the inferred register instance names. The design of state machines is made easier because state assignments can be made using parameters and the enum compiler directive. The state machine compiler can easily make trade-offs between fully decoded

one-hot state assignments and fully encoded states. Refer to the *Synopsys HDL Compiler for Verilog Reference Manual* for more details about using the state machine compiler.

The Reset Problem

> *For sync resets, set the v3.0 variable* `compile_preserve_sync_resets = true.`

The use of a synchronous reset results in the creation of discrete logic to perform the reset function, which under some circumstances will not initialize the flip-flop during logic simulation. This problem can be solved in one of two ways. The first and preferred solution is to set the v3.0 compile variable: `compile_preserve_sync_resets = true`. This places the reset logic next to the flip-flop and in some cases a flip-flop with a synchronous reset is generated. The other alternative is to use an asynchronous reset.

```
reg [7:0] Q;

// synchronous reset
always @(posedge clk)
     if (reset)
          Q <= 0;
     else if (loadEnable)
          Q <= D;

// asynchronous reset
always @(posedge clk or posedge reset)
     if (reset)
          Q <= 0;
     else if (loadEnable)
          Q <= D;
```

Latch Inference

> *Within an always block, fully specify assignments or use default conditions.*

A variable of type `reg` which is assigned a value inside an `always` block without the `posedge` or `negedge` clock construct generates a latch if the variable is not assigned a value under all conditions. If there is a set of input conditions for which the variable is not assigned, the variable will retain its current value. The example below is for a round-robin priority selector. `grant` designates the device that currently has the bus. `request` designates those devices that are requesting use of the bus. `grant_next` is the next device to be granted the bus. Priority is always given to the device one greater than the current one. Using

a default for the case statement will not prevent the creation of a latch. If no requests are active, then `grant_next` is not assigned so a latch will be created. The solution is to specify the default conditions at the beginning of the `always` block before the case statement.

```
module  rrpriocase(request,grant,grant_next);
input [3:0] request, grant;
output [3:0] grant_next;
always @(request or grant) begin
     grant_next = grant; // put default here to avoid latch in-
ference
     case(1) // synopsys parallel_case full_case
          grant[0]:
               if (request[1]) grant_next = 4'b0010;
               else if (request[2]) grant_next = 4'b0100;
               else if (request[3]) grant_next = 4'b1000;
               else if (request[0]) grant_next = 4'b0001;
          grant[1]:
               if (request[2]) grant_next = 4'b0100;
               else if (request[3]) grant_next = 4'b1000;
               else if (request[0]) grant_next = 4'b0001;
               else if (request[1]) grant_next = 4'b0010;
          grant[2]:
               if (request[3]) grant_next = 4'b1000;
               else if (request[0]) grant_next = 4'b0001;
               else if (request[1]) grant_next = 4'b0010;
               else if (request[2]) grant_next = 4'b0100;
          grant[3]:
               if (request[0]) grant_next = 4'b0001;
               else if (request[1]) grant_next = 4'b0010;
               else if (request[2]) grant_next = 4'b0100;
               else if (request[3]) grant_next = 4'b1000;
     endcase
end
endmodule
```

Another solution is to add the following else clause at the end of each case item:

```
else grant_next = grant;
```

Using Arrays of Multiplexers

Instantiate multiplexers that output vectors.

When using v3.0 or earlier it is a good idea to instantiate multiplexers that output vectors. (For time-critical signals it may also be necessary to instantiate multiplexers that only output

a single bit.) This involves creating a library of multiplexer modules that can be instantiated in the design. In the following example, there is a single input vector to simplify parameter passing through multiple levels of hierarchy. Use parameterized modules to build different versions of a design.

```
// may want to pass an array as a parameter
reg [1:0] in [0:3];

// convert array to vector
wire [2*4:1] inVec = {in[0], in[1], in[2], in[3]};

// multiplexer instantiation:
mux4to1x4(inVec, sel, out);

// multiplexer module
module mux4to1x4 (inVec, sel, out);
input [4*4:1] inVec;
input [1:0] sel;
output [3:0] out;

wire [3:0] in0, in1, in2, in3;
assign {in3, in2, in1, in0} = inVec;

// synopsys dc_script_begin
// dont_touch {m4_0 m4_1 m4_2 m4_3}
// synopsys dc_script_end

// note: instance names must be unique
mux4to1 m4_0 ({in3[0], in2[0], in1[0], in0[0]}, sel, out[0]);
mux4to1 m4_1 ({in3[1], in2[1], in1[1], in0[1]}, sel, out[1]);
mux4to1 m4_2 ({in3[2], in2[2], in1[2], in0[2]}, sel, out[2]);
mux4to1 m4_3 ({in3[3], in2[3], in1[3], in0[3]}, sel, out[3]);
endmodule

// Module mux4to1 should map to vendor MUX4 cell
module mux4to1 (vec, sel, out);
input [3:0] vec;
input [1:0] sel;
output out;

assign out = vec[sel];
endmodule
```

Using Arrays

The Synopsys Verilog HDL Compiler supports memory arrays. For a group of registers that are accessed using an index, the memory array construct provides a more concise specification. The Verilog HDL models memories as an array of register variables. Each register in the array is addressed by a single array index.

The following example declares a memory array called `cntr` that is used to implement a bank of eight counters. A counter index variable can directly select the register to be incremented:

```
reg [5:0] cntr[0:7];

always @(posedge clk or posedge reset)
     if (reset)
          for (i=0; i<8; i=i+1)
               cntr[i] <= 6'd0;
     else if (inc)
          cntr[index] <= cntr[index] + 1'b1;
```

Without the use of arrays, this description requires eight incrementors instead of one and many more lines of code.

```
reg [5:0] cntr7, cntr6, cntr5, cntr4;
reg [5:0] cntr3, cntr2, cntr1, cntr0;
always @(posedge clk or posedge reset)
     if (reset)
        {cntr7, cntr6, cntr5, cntr4} <= {6'd0, 6'd0, 6'd0, 6'd0};
        {cntr3, cntr2, cntr1, cntr0} <= {6'd0, 6'd0, 6'd0,
6'd0};
     else if (inc)
          case(index) // parallel_case full_case
               3'd0: cntr0 <= cntr0 + 1;
               3'd1: cntr1 <= cntr1 + 1;
               3'd2: cntr2 <= cntr2 + 1;
               3'd3: cntr3 <= cntr3 + 1;
               3'd4: cntr4 <= cntr4 + 1;
               3'd5: cntr5 <= cntr5 + 1;
               3'd6: cntr6 <= cntr6 + 1;
               3'd7: cntr7 <= cntr7 + 1;
          endcase
```

The eight incrementors can be reduced to one by rewriting the increment section of the code and adding even more lines of code:

```
reg [5:0] result; // pseudo reg
always @(index or cntr7 or cntr6 or cntr5 or cntr0
     or cntr3 or cntr2 or cntr1 or cntr0)
          case(index) // parallel_case
               3'd0: result = cntr0;
               3'd1: result = cntr1;
               3'd2: result = cntr2;
               3'd3: result = cntr3;
               3'd4: result = cntr4;
               3'd5: result = cntr5;
               3'd6: result = cntr6;
               3'd7: result = cntr7;
               default: result = 'bx;
          endcase

wire [5:0] inc = result + 1'b1;

always @(posedge clk or posedge reset)
     if (reset)
        {cntr7, cntr6, cntr5, cntr4} <= {6'd0, 6'd0, 6'd0,
6'd0};
        {cntr3, cntr2, cntr1, cntr0} <= {6'd0, 6'd0, 6'd0,
6'd0};
     else if (inc)
          case(index) // parallel_case full_case
               3'd0: cntr0 <= inc;
               3'd1: cntr1 <= inc;
               3'd2: cntr2 <= inc;
               3'd3: cntr3 <= inc;
               3'd4: cntr4 <= inc;
               3'd5: cntr5 <= inc;
               3'd6: cntr6 <= inc;
               3'd7: cntr7 <= inc;
          endcase
```

While arrays can be used to specify the function more concisely, the resultant logic is not very optimal in terms of area and speed. Instead of using the index operator [], the index can be generated using a decoder. This decoded index can then be used for both fetching the selected counter and assigning a new value to it. This technique of using a decoded value as the select control produces a more optimal design. The use of a for-loop in the example below is just a more succinct way of writing the resultant description.

```
wire [7:0] incDec = inc << index;

always @(posedge clk or posedge reset)
     if (reset)
```

```
      for( i=0; i<=7; i=i+1 )
           cntr[i] <= 6'd0;
else for( i=0; i<=7; i=i+1 )
      if (incDec[i])
           cntr[i] <= cntr[i] + 1'b1;
```

A decoder with enable can be inferred through the use of a function call:

```
wire [7:0] incDec = decode8en(index, inc);
```

Register File Example

The following example declares a memory array called `rf` which consists of thirty-two 32-bit registers.

```
reg [31:0] rf[0:31];
```

While a multi-ported register file would most likely be implemented as a megacell, it can also be generated by the design compiler using a memory array. The following example contains the specification for a 32-location register file that would be contained in a typical 32-bit microprocessor.

```
module regfile(clk, weA, weB, dinA, dinB, destA, destB,
      srcA, srcB, doutA, doutB);

input clk, weA, weB;
input [31:0] dinA, dinB;
input [4:0] destA, destB;
input [4:0] srcA, srcB;

output [31:0] doutA, doutB;

reg [31:0] rf [0:31];

assign doutA = srcA==0 ? 0 : rf[srcA];
assign doutB = srcB==0 ? 0 : rf[srcB];

always @ (posedge clk) begin
 if ( weA )
       rf[destA] <= dinA;
 if ( weB )
       rf[destB] <= dinB;
end
endmodule
```

The logic generated for this example is not very optimal in terms of speed or area. A more optimal implementation of a register file can be generated by using a decoded version of the

source and destination addresses. A selector operator is used to multiplex between dinB and dinA. Priority is automatically given to `destB` without the use of additional logic. For this version, the generated logic for each element is identical to the specification in the for-loop, with the "?" operator mapping to a 2-to-1 multiplexer. This version is more efficient because there are fewer indexing operations and because the decode logic is explicitly specified.

```
module regfile(clk, weA, weB, dinA, dinB, destA, destB,
     srcA, srcB, doutA, doutB);

input clk, weA, weB;
input [31:0] dinA, dinB;
input [4:0] destA, destB;
input [4:0] srcA, srcB;

output [31:0] doutA, doutB;

reg [31:0] rf [0:31];
integer i;

assign doutA = srcA==0 ? 0 : rf[srcA];
assign doutB = srcB==0 ? 0 : rf[srcB];

wire [31:0] weDecA = (weA << destA); // additional detail
wire [31:0] weDecB = (weB << destB); // additional detail
wire [31:0] weDec = weDecA | weDecB; // additional detail

always @ (posedge clk) begin
 for ( i=0; i<=31; i=i+1 ) // for-loop replaces random ac-
cess
     rf[i] <= weDec[i] ? (weDecB[i] ? dinB : dinA) : rf[i];
end
endmodule
```

Array of Counters

The following example describes an array of counters. On each clock, one counter can be conditionally incremented and one counter conditionally decremented. If an attempt is made to increment and decrement the same counter in the same clock, the old value should be preserved. The signal inhibit is used to achieve this function.

```
reg [5:0] cntr[0:7]; // 2-D array declaration
wire inhibit = incEnable & decEnable & (incAdr == decAdr);
wire inc = incEnable & ~inhibit;
wire dec = decEnable & ~inhibit;
```

```
always @(posedge clk) begin
     cntr[incAdr] <= inc ? cntr[incAdr]+1 : cntr[incAdr];
     cntr[decAdr] <= dec ? cntr[decAdr]-1 : cntr[decAdr];
end
```

When inc&~dec&(incAdr==decAdr), the example above will not simulate correctly. The second statement will take priority over the first statement since it is the last one in the block and it will overwrite the incremented value assigned in the first statement. Furthermore the selected `cntr` value to increment is computed twice instead of once. Here is the corrected example:

```
always @(posedge clk) begin
     if (inc)
          cntr[incAdr] <= cntr[incAdr]+ 1'b1;
     if (dec)
          cntr[decAdr] <= cntr[decAdr]- 1'b1;
end
```

The quality of the synthesized circuit can still be dramatically improved by using a decoded version of the address.

```
reg [5:0] cntr[0:7];
wire [7:0] inc = incEnable << incAdr;
wire [7:0] dec = decEnable << decAdr;

always @(posedge clk) begin
     for (i = 0; i <= 7; i = i + 1)
          cntr[i] <= inc[i] ^ dec[i] ?
               (inc[i] ? cntr[i] + 1 : cntr[i] - 1) : cntr[i];
end
```

This code still has a problem in that eight incrementors and eight decrementors will be created. The following example fixes this last problem.

```
wire [5:0] cntrPlus1 = cntr[incAdr] + 1'b1;
wire [5:0] cntrMinus1 = cntr[decAdr] - 1'b1;

always @(posedge clk) begin
     for (i = 0; i <= 7; i = i + 1)
          cntr[i] <= inc[i] ^ dec[i] ?
               (inc[i] ? cntrPlus1 : cntrMinus1) : cntr[i];
end
```

Multiple Assignments to the Same Variable

Avoid multiple assignments to the same variable except for arrays and vectors.

Assignments to the same variable in separate statements should be avoided except when used with 2-D arrays or vectors, where different elements can be updated at the same time without contention. When the same element is being written, the later assignment is dominant. In the case of multiple assignments to the same variable in different synchronous blocks, Synopsys infers two separate flip-flops which are ANDed together to produce a single output.

```
module test(clk, load1, a1, load2, a2, q);
input clk, load1, load2, a1, a2;
output q;
reg q;

always @ (posedge clk) begin
     if (load1)
          q <= a1;
end

always @ (posedge clk) begin
     if (load2)
          q <= a2;
end
endmodule
```

Putting both assignments in the same block avoids this problem. In this case, `load2` is dominant since it occurs later in the block. The logic to load variable `a1` is `load1` and `~load2`. If both inputs are mutually exclusive, then use the case `always` instead.

```
always @ (posedge clk) begin
     if (load1)
          q <= a1;
     if (load2)
          q <= a2;
end
```

In the case of vectors and 2-D arrays, different locations in the array or vector can be updated without contention. For example:

```
reg[31:0] flag;

always @(posedge clk) begin
```

```
        if (set_flag)
             flag[set_index] <= 1'b1;
        if (reset_flag)
             flag[reset_index] <= 1'b0;
end
```

If `set_index` and `reset_index` specify the same location, the selected bit will be reset since that function corresponds to the last assignment in the block.

The following statement invokes the functions `dec32L` and `dec32`, which map to a user-defined decode module or DesignWare in order to create an optimized version of the previous example.

```
// Version using function invocation & map_to_module (not shown)
flag <= flag & dec32L(reset_index) | dec32(set_index);

// Version using DesignWare & module instantiation
wire [31:0] reset_dec, set_dec;
DW01_decode #(5) dec32_0 (reset_index, reset_dec);
DW01_decode #(5) dec32_1 (set_index, set_dec);
flag <= flag & ~reset_dec | set_dec;
```

Using `case` Statements

The following case statement implements a selector function:

```
always @(sela or selb or selc or seld or a or b or c or d)
     case ({sela, selb, selc, seld})
          4'b1000: dout = a;
          4'b0100: dout = b;
          4'b0010: dout = c;
          4'b0001: dout = d;
     endcase
```

If multiple select lines are active, then none of the case items will be selected and the variable `dout` will be unchanged from it's previous value. This results in the creation of a latch for dout in order conform with the Verilog language specification. Latch inference can be avoided either with the use of the Synopsys `full_case` compiler directive or with the use of the default clause. The `full_case` directive tells the tool that all valid states are represented. During simulation, if the case expression evaluates to a value that is not covered by the case-items, the Verilog simulation and the gate-level simulation will not compare. The `default` statement can also prevent latch inference but its meaning is different from the

full_case directive. The default statement is used for ambiguity handling. It specifies the output for any input values not defined. In v3.0, Design Compiler will automatically use full and/or parallel case when appropriate, provided all the case item expressions are constants. Assigning the output to ‘bx in the default statement allows unknown propagation for Verilog simulation and specifies don’t-care conditions for synthesis. This usually generates fewer gates. For these reasons, the default statement is preferred over the full_case directive.

```
default: dout = 3’bx; // for ambiguity handling
```

```
Given that the select signals are mutually exclusive, a more optimal selector design can be implemented using the casez statement.

always @(sela or selb or selc or seld or a or b or c or d)
     casez ({sela, selb, selc, seld}) // synopsys parallel_case
          4’b1???: dout = a;
          4’b?1??: dout = b;
          4’b??1?: dout = c;
          4’b???1: dout = d;
          default: dout = ‘bx;
     endcase
```

The Verilog case statement is evaluated by comparing the expression following the case keyword to the case-item expressions in the exact order they are given. The statement corresponding to the first case item that matches the case expression is executed. If all the comparisons fail, the default statement is executed. This may not be what the designer intended. The parallel_case directive instructs the Design Compiler to evaluate all case items in parallel and for all case items that match the case expression, to execute the corresponding statements. Without this directive, the logic generated implements a type of priority encode logic for each case item. If more than one case item evaluates true, the generated gate-level design will not match the behavior of the original Verilog source. Without the parallel_case directive, the selector function using the casez would actually be equivalent to the following selector description:

```
always @(sela or selb or selc or seld or a or b or c or d)
     casez ({sela, selb, selc, seld}) // equivalent design
          4’b1???: dout = a;
          4’b01??: dout = b;
          4’b001?: dout = c;
          4’b0001: dout = d;
```

```
            default: dout = 'bx;
        endcase
```

Use the case `always` *to implement selector type functions.*

The selector function can be more concisely specified as:

```
// note: inputs must be mutually exclusive
always @(sela or selb or selc or seld or a or b or c or d)
     case(1'b1) // synopsys parallel_case
          sela: dout = a;
          selb: dout = b;
          selc: dout = c;
          seld: dout = d;
          default: dout = 'bx;
     endcase
```

This construct is best used whenever the inputs are mutually exclusive. The following example shows the execute unit of a V7 SPARC integer unit.

```
wire       // instruction decode
  AND = op==9'h81, ANDN = op==9'h85,
  OR = op== 9'h82, ORN = op==9'h86,
  XOR = op==9'h83, XNOR = op==9'h87,
  ANDCC = op==9'h91, ANDNCC = op==9'h95,
  ORCC = op==9'h92, ORNCC = op==9'h96,
  XORCC = op==9'h93, XNORCC = op==9'h97; // etc...

always @(operand1 or operand2 or Y or PSR or WIM or TBR or
  AND or ANDN or ANDCC or ANDNCC OR or ORN or ORCC or ORNCC or
  XOR or XNOR or XORCC or XNORCC or WRY or WRPSR or WRWIM or WRTBR or
  MULSCC or SLL or SRL or SRA or RDY or RDPSR or RDWIM or RDTBR)
  case(1) // synopsys parallel_case
     AND, ANDN, ANDCC, ANDNCC:
        result = operand1 & operand2;
     OR, ORN, ORCC, ORNCC:
        result = operand1 | operand2;
     XOR, XNOR, XORCC, XNORCC, WRY, WRPSR, WRWIM, WRTBR:
        result = operand1 ^ operand2;
     MULSCC: result = Y[0] ? sum : operand1;
     SLL: result = operand1 << operand2[4:0];
     SRL: result = operand1 >> operand2[4:0];
     SRA: result = {{31{operand1[31]}}, operand1} >> oper-
and2[4:0];
     RDY: result = Y;
     RDPSR: result = PSR;
```

```
        RDWIM: result = WIM;
        RDTBR: result = {TBR, 4'd0};
        default: result = sum; // for all other instructions
    endcase
```

Compiler Directives

Imbed dont_touch directives in the Verilog source code.

Compiler commands can be placed in the Verilog source code using the directives `dc_script_begin` and `dc_script_end`. The Synopsys Design Compiler will automatically compile submodules unless the `dont_touch` attribute is placed on the cell instances. For modules that are instantiated more than once in a design and not uniquified the `dont_touch` attribute is required. When required, `dont_touch` directives should be placed in the Verilog source code.

```
// synopsys dc_script_begin
// dont_touch {cx cl ap sc ll rb rt pb}
// synopsys dc_script_end
```

Parameterized Designs

Verilog provides the capability to build parameterized designs by changing parameter values in any module instance. The method supported by HDL Compiler is to use the module instance parameter value assignment instead of the `defparam` statement. In the following design, Design Compiler builds a 32-bit version of the 2-to-1 mux and uses this instance in the shiftLeft design. In v3.0 analyze and elaborate replace the use of templates.

```
module shiftLeft(a, shiftCnt, shifta);
input [31:0] a;
input [4:0] shiftCnt;
output [31:0] shifta;
wire [31:0] d0, d1, d2, d3;
// assign shifta = a << shiftCnt;
mux2to1 #(32) m21_0 (a, {a[30:0], 1'b0}, shiftCnt[0], d0);
mux2to1 #(32) m21_1 (d0, {d0[29:0], 2'b0}, shiftCnt[1], d1);
mux2to1 #(32) m21_2 (d1, {d1[27:0], 4'b0}, shiftCnt[2], d2);
mux2to1 #(32) m21_3 (d2, {d2[23:0], 8'b0}, shiftCnt[3], d3);
mux2to1 #(32) m21_4 (d3, {d3[15:0], 16'b0}, shiftCnt[4], shif-
ta);
endmodule

module mux2to1 (a, b, s, z);
parameter width = 2;
```

```
input [width-1:0] a, b;
input s;
output [width-1:0] z;
// synopsys template
assign z = s ? b : a;
endmodule
```

Tasks

For the purposes of synthesis, task statements are similar to functions in Verilog except they can have more than one output or no outputs, and they can have inout ports. Only `reg` variables can receive the output from a task, unlike functions.The tasks logic becomes part of the module from which it is invoked, so it is not necessary to input all the variables explicitly.

```
always task_invocation (in1, in2, reg1, reg2);
```

Although perhaps not as readable, a function can also return multiple values by concatenating the results together and then using an assign statement or procedural assignment to separate the values in the calling module.

```
// continuous assignment
assign {wire1, wire2, wire3} = function_invocation(in1, in2);

always @(posedge clk ) // procedural assignment
     {reg1, reg2, reg3} <= function_invocation(in1, in2);
```

State Machine Design

A state machine can be specified using a number of different formats: Verilog, Synopsys State Table Design Format, or the PLA Design Format. The Synopsys Verilog HDL Compiler can “extract” the state table from a Verilog description if the `state_vector` and `enum` directives are used. The `state_vector` directive requires the use of inferred flop-flops within the same module as the state machine specification. However, an extracted state table is not necessary in order to compile a state machine. The use of an extracted state table does provide the following benefits:

- It provides good documentation of the state machine behavior
- State minimization can be performed
- Don’t-care conditions are utilized since FSM extraction includes flattening
- Tradeoffs between different encoding styles can easily be made
- Don’t-care state codes are automatically derived

The state machine compiler is especially effective when using a one-hot encoding style. After using the state machine compiler, Design Compiler can also be used for further optimization. (When using an encoded state vector with few don't-care states, there is not much benefit in using the state machine compiler.) When a state table is extracted, Design Compiler enumerates all state transitions. For example, a reset function generates an explicit transition from every possible state to the reset state. This can potentially cause the state table to explode in size. Even though invalid input combinations or mutually exclusive inputs can be specified using the `casez` construct or the `parallel_case` directive, they cannot be concisely represented in the state table format. However, the PLA format provides a way to specify these cases concisely, potentially resulting in a more optimal design. The PLA format can also specify the various don't-care conditions and have these utilized without flattening the design. (The PLA format is already a flat, two-level sum of products.)

In a Mealy machine, the outputs depend on the current inputs. In a Moore machine the outputs are either registered or depend only on the current state. An "incompletely specified state machine" means that the transition behavior is not specified for all possible input conditions and there exists a next-state don't-care set. The next-state assignments can be performed inside a sequential block or in a combinational block. The combinational block can either be an `always` block or a function. If all the outputs are registered, the output assignments can be included in a sequential block; otherwise they must be in a combinational block. State machine outputs that drive asynchronous logic or preset and clear inputs of flip-flops must not glitch. These outputs must be registered or else gray-code state encoding must be used for the corresponding state transitions.

The example below describes a simple state machine with an inferred state register.

```
// moore machine
module fsm_example(clk, reset, c, valid);
input clk, reset, c;
output valid;

parameter [2:0] // synopsys enum fsm_states
      idle = 3'd0,
      one_str = 3'd1,
      zero_str = 3'd2,
      valid_str = 3'd4,
      invalid_str = 3'd3;

reg [2:0] /* synopsys enum fsm_states */ state;
// synopsys state_vector state

// next state assignments in sequential block
```

```
always @(posedge clk)
      if (reset)
            state <= idle;
      else case(state)
            idle: state <= c ? one_str : invalid_str;
            one_str: state <= c ? one_str : zero_str;
            zero_str: state <= c ? valid_str : zero_str;
            valid_str: state <= c ? valid_str : invalid_str;
            invalid_str: state <= invalid_str;
            default: state <= 3'bx; // dont_care conditions
      endcase

// note: valid becomes msb of state register
assign valid = state == valid_str;
assign first_one = (state == idle) & c;
endmodule
```

The next-state assignments can also be done in a combinational block with the next-state-to-current-state assignment done in a sequential block. Using this technique, unregistered outputs can be assigned along with the next state. These techniques are illustrated in the following example:

```
// mealy machine
module fsm_example(clk, reset, c, valid);
input clk, reset, c;
output valid;

parameter [3:0] // synopsys enum fsm_states
      idle = 3'd0,
      one_str = 3'd1,
      zero_str = 3'd2,
      valid_str = 3'd4,
      invalid_str = 3'd3;

reg valid, first_one; // a wire
reg [2:0] /* synopsys enum fsm_states */ state;
// synopsys state_vector state

// next state assignments in combinational block
always @(c or state or reset)
      if (reset) begin
            nxt_state = idle;
            valid = 0;
            first_one = 0;
      end
      else begin
```

```
                valid = 0; // put defaults here
                first_one = 0; // put defaults here

                case(state)
                     idle:
                          if (c) begin
                               nxt_state = one_str;
                               first_one = 1;
                          end else
                               next_state = idle;
                     one_str:
                          if (c) nxt_state = one_str;
                          else nxt_state = zero_str;
                     zero_str:
                          if (c) nxt_state = valid_str;
                          else nxt_state = zero_str;
                     valid_str: begin
                          if (c) nxt_state = valid_str;
                          else nxt_state = invalid_str;
                          valid = 1;
                     end
                     invalid_str: begin
                          nxt_state = invalid_str;
                          valid = 0;
                     end
                     default: nxt_state = 3'bx; // dont_care condi-
tions
                endcase
           end

     // an additional sequential block is needed
     always @(posedge clk)
          state <= next_state;
     endmodule
```

Outputs that depend only on the current state can be decoded using a `case` statement with case variable "`state`". Outputs that depend on the state transition (which implies a dependency on both the current state and the inputs) can be conditionally asserted in a particular state or they can be decoded from the variable "`nxt_state`".

Putting It All Together (To Build a SPARC Processor)

Where should you start in order to design a SPARC processor. Start high-level and then refine to meet performance and speed requirements. It is possible to describe a synthesizable Verilog description of a SPARC V7 processor in a few pages of code. The minimum design consists of a two-stage pipeline: a fetch stage and a decode-execute-write stage. Most

instructions will execute in a single cycle. This description can then be refined as necessary to include more pipeline stages and additional structure to meet timing and area objectives. One place to start is with the ISP description contained in the SPARC architecture manual. A direct translation to Verilog can be made if some simple guidelines are followed. In an ISP or C-like description, both wires and registers are represented as variables that are assigned values. In order to translate such a description into Verilog, the designer must first decide which variables are registers that retain state between clocks and which are temporary variables that do not retain state. The temporary variables become wires, which are updated by either a continuous assignment or by a procedural assignment contained within an `always` block. Registers can only be updated with a procedural assignment contained within an `always` block that uses the `posedge` or `negedge` clock construct. Therefore, two separate blocks are required.

```
// SPARC V7 Verilog instruction set definition
// incomplete listing

// SPARC registers
reg [31:0] nPC, PC;        // next PC and PC
reg [31:0] ir, Y;          // instruction reg, Y Reg
reg [7:0] WIM; // window invalid mask
reg [31:4] TBR;            // trap base register
reg [5:0] p1;              // unused bits of psr
reg [1:0] p2;              // unused bits of psr
reg [2:0] CWP;  // current window pointer
reg [3:0] CCR;             // condition code register
reg [3:0] PIL;             // processor interrupt level
reg [31:0] r[0:135];       // register file
reg EC, EF, S, PS, ET;          // psr bits
reg [3:0] ver, imp;        // version & implementation

wire [31:0] PSR = {ver, imp, CCR, p1, EC, EF, PIL, S, PS, ET, p2,
CWP};

// other registers
reg annul, trap1, trap2;

// define operands
wire [7:0] imm_asi;
wire [5:0] op3;
wire [4:0] rd, rs1, rs2;
wire [1:0] op1;
wire s = annul | trap1 | trap2;
wire [8:0] op = {s, op1, op3};  // opcode
wire N, Z, V, C;                // condition code bits
```

```
wire i;                          // immediate bit
wire a = ir[29];                 // annul bit
wire BA = ir[28:25]==4'b1000;    // Branch Always
assign {op1, rd, op3, rs1, i, imm_asi, rs2} = ir;

// synopsys translate_off
initial r[128] = 0; // global zero set to zero
// synopsys translate_on

wire // instruction decode
     CALL = op[8:6]==3'b001, SETHI = op[8:3]==6'b000100,
     BICC = op[8:3]==6'b000010, FBCC = op[8:3]==6'b000110; // etc...

// define execute logic variables/wires
reg [31:0] st_data, st_datax;
reg [31:0] result, da, addr;
reg [4:0] dest;            // register file destination
reg [3:0] next_cwp;
reg next_annul, next_s, next_et, next_ps;
reg unimplemented;
reg icc_valid, fcc_valid;
reg fp_exc, fp_dis;
reg unaligned, tagged_overflow;
reg win_overflow, win_underflow;
reg illegal_inst, trap_inst;
reg U, G, L, E;
reg dest_valid;

wire [31:0] src1, src2, operand1, operand2, sum;
wire [31:0] st_operand, st_operandx;
wire [3:0] nzvc;           // next condition code bits
wire cin, cout;

// fetch operands
assign
     src1 = r[win_dec(rs1, CWP)],
     src2 = r[win_dec(rs2, CWP)],
     operand1 = MULSCC ? {(N ^ V), src1[31:1]} : src1,
     operand2 = complement ^ (i ? {{20{ir[12]}}, ir[11:0]} :
src2),
     st_operand = r[win_dec(rd, CWP)],
     st_operandx = r[win_dec(rd | 1'b1, CWP)];

// define result adder and condition code bits
assign
     cin = subtract | subx&~C | addx&C,
     {cout, sum} = operand1 + operand2 + cin,
```

```
      nzvc = {result[31], result==0, // n, z
            add&operand1[31]&operand2[31]&~result[31] | // v
            add&~operand1[31]&~operand2[31]&result[31] | // v
            tagged&(|{operand1[1:0], operand2[1:0]}), // v
            add && complement^cout }; // c

assign {N,Z,V,C} = CCR;         // assign condition code bits

// Instruction Definitions
always @(clk) begin  // clk replaces sensitivity list
      addr = nPC + 3'd4;    // default instruction address
      result = sum;         // default result is sum
      da = sum;             // default data address
      dest = rd;            // default destination
      next_annul = 0;       // default annul
      next_cwp = CWP;       // hold value
      next_s = S;           // hold value
      next_ps = PS;         // hold value
      next_et = ET;         // hold value
      unimplemented = 0;    // default is valid instruction
      icc_valid = 0;
      fcc_valid = 0;
      fp_exc = 0;
      fp_dis = 0;
      unaligned = 0;
      tagged_overflow = 0;
      win_overflow = 0;
      win_underflow = 0;
      illegal_inst = 0;
      trap_inst = 0;
      st_data = st_operand;
      st_datax = st_operandx;
      dest_valid = 1;       // write result to register file
      {U, G, L, E} = 0;
      case(1) // synopsys parallel_case
      trap1: begin
            dest = 17;
            result = PC;
            next_s = 1;
            next_ps = S;
            next_et = 0;
      end
      trap2: begin
            addr = {TBR[31:4], 4'd0};
            dest = 18;
            result = PC;
            next_annul = 1;
```

```
        end
        SETHI: result = {ir[21:0], 10'd0};
        CALL: begin
              addr = PC + {ir[29:0], 2'b00};
              result = PC;
              dest = 15;
        end
        BICC: begin
              icc_valid = ir[28] ^ |((1'b1 << ir[27:25]) &
                    {V, N, C, C | Z, N ^ V, Z | (N ^ V), Z, 1'b0});
              if (icc_valid) begin
                    addr = PC + {{8{ir[21]}}, ir[21:0], 2'b00};
                    if (BA & a) next_annul = 1;
              end else if (a)
                    next_annul = 1;
              dest_valid = 0;
        end
        FBCC: begin
              {U, G, L, E} = 1'b1 << fcc;
              fcc_valid = |((1'b1 << ir[28:25]) & {E|G|L,U|L|E,E|L,
                    E|U|G,E|G,E|U,E,1'b1,U,G,G|U,L,L|U,L|G,L|G|U,1'b0}
);
              if (fcc_valid) begin
                    addr = PC + {{8{ir[21]}}, ir[21:0], 2'b00};
                    if (BA & a) next_annul = 1;
              end else if (a)
                    next_annul = 1;
              dest_valid = 0;
              fp_exc = fexc;
              fp_dis = fp_ | ~EF;
        end
        LDSB, LDUB, LDSH, LDUH, LD, LDA, LDSBA, LDUBA, LDSHA, LDU-
HA,
        LDD, LDDA: begin
              da = sum;
              unaligned = hw&&da[0] | wd&&da[1:0] | db&&da[2:0];
              result = load_align(data,bt,hw,wd,signedload,da[1:0]);
        end
        SWAP, SWAPA, LDSTB, LDSTBA: begin
              da = sum;
              unaligned = hw&&da[0] | wd&&da[1:0] | db&&da[2:0];
              result = data;
              if (LDSTB | LDSTBA)
                    st_data = 32'hffffffff;
              else st_data = st_operand;
        end
        STH, STHA, ST, STA, STD, STDA: begin
```

```
                da = sum;
                unaligned = hw&&da[0] | wd&&da[1:0] | db&&da[2:0];
                st_data =  store_align(st_operand,bt,hw,wd,da[1:0]);
                dest_valid = 0;
                if (STD | STDA)
                     st_datax = st_operandx;
          end
          LDF, LDFSR, STF, STFSR, STDFQ: begin
                da = sum;
                fp_exc = fexc;
                fp_dis = fp_ | ~EF;
                unaligned = hw&&da[0] | wd&&da[1:0] | db&&da[2:0];
                dest_valid = 0;
          end
          ADDCC, ADDXCC, SUBCC, SUBXCC, TADDCC, TSUBCC:
                result = sum;
          TADDCCTV, TSUBCCTV: begin
                tagged_overflow = V;
                result = sum;
          end
          AND, ANDN, ANDCC, ANDNCC:
                result = operand1 & operand2;
          OR, ORN, ORCC, ORNCC:
                result = operand1 | operand2;
          XOR, XNOR, XORCC, XNORCC:
                result = operand1 ^ operand2;
          WRY, WRPSR, WRWIM, WRTBR: begin
                result = operand1 ^ operand2;
                dest_valid = 0;
          end
          MULSCC: result = Y[0] ? sum : operand1;
          SLL: result = operand1 << operand2[4:0];
          SRL: result = operand1 >> operand2[4:0];
          SRA: result = {{32{operand1[31]}}, operand1} >> oper-
and2[4:0];
          RDY: result = Y;
          RDPSR: result = PSR;
          RDWIM: result = WIM;
          RDTBR: result = {TBR, 4'd0};
          SAVE: begin
                next_cwp = CWP - 1'b1;
                win_overflow = WIM[next_cwp];
          end
          RESTORE: begin
                next_cwp = CWP + 1'b1;
                win_underflow = WIM[next_cwp];
          end
```

```
        JMP: begin
             addr = sum;
             result = PC;
        end
        RETT: begin
             addr = sum;
             next_cwp = CWP + 1'b1;
             win_underflow = WIM[next_cwp];
             next_s = PS;
             next_et = 1;
             illegal_inst = S & ET;
             dest_valid = 0;
        end
        TICC: begin
             icc_valid = ir[28] ^ |((1'b1 << ir[27:25]) &
                  {V, N, C, C | Z, N ^ V, Z | (N ^ V), Z, 1'b0});
             trap_inst = icc_valid;
             dest_valid = 0;
        end
        IFLUSH: begin
             da = sum; // flush_cache_line(da);
             dest_valid = 0;
        end
        FPOP, FPCMP: begin
             fp_exc = fexc;
             fp_dis = fp_ | ~EF;
             dest_valid = 0;
        end
        annul: dest_valid = 0;
        default: unimplemented = 1;
        endcase
   end

   wire trap = fp_exc | fp_dis | unaligned | trap_inst |
        tagged_overflow | win_overflow | win_underflow | ille-
   gal_inst;

   always @(posedge clk) begin
        ir <= inst;          // load the instruction register
        if (~trap)      PC <= nPC;      // 2 stage pc chain
        trap1 <= trap;
        trap2 <= trap1;
   end

   always @(posedge clk)
        if (reset) begin
             S <= 1;
```

```
        ET <= 0;
        nPC <= 0;
    end else if (trap) begin
        TBR[11:4] <= trap_type;      // update TBR before using
        CWP <= CWP - 1'b1;     // dec CWP before saving PC & nPC
    end else begin
        nPC <= addr;
        if (setcc) CCR <= nzvc;      // only setcc ops write CCR
        if (WRY) Y <= result;
        if (MULSCC) Y <= {operand1[0], Y[31:1]};
        if (WRPSR)
        {ver, imp, CCR, p1, EC, EF, PIL, S, PS, ET, p2, CWP} <=
        result;
        else begin
            CWP <= next_cwp;
            S <= next_s;
            PS <= next_ps;
            ET <= next_et;
        end
        if (WRWIM) WIM <= result[7:0];
        if (WRTBR) TBR[31:12] <= result[31:12];
        if (dest_valid && dest) r[win_dec(dest, CWP)] <= result;
        if (LDD | LDDA) r[win_dec(dest | 1'b1, CWP)] <= datax;
    end
```

The Design Compiler is capable of sharing resources. Three sharing methods are available: automatic sharing, automatic sharing with manual controls, and manual sharing. However, for this example, resource sharing is avoided for the most part by specifying the sharing explicitly in the Verilog source code. Several simplifications have already been made. The operands are computed once for all instructions. The multiply step instruction needs to modify `operand1`, so this function is incorporated into the `operand1` assignment. Otherwise an additional add operator would be required. An adder and subtracter cannot be shared, but a subtract function can be performed by complementing `operand2` and forcing a carry-in to the adder. This function has been incorporated into the specification. The sum is computed once and then used for many different instructions; it is the default when no other result is specified. Many instructions set the condition codes based on the result computed. This logic was taken out of the `case` statement so it could be shared by many instructions. The integer condition code is evaluated for both BICC and TICC. This evaluation can be taken out of the `always` block and made into a continuous assignment. Only those instructions that evaluate the integer condition codes need to test this value. Some instructions will require more than one cycle, for example the SWAP instruction (this logic is not shown). Synopsys can synthesize the register file, but this is not practical for a real implementation because of its size. A register file megacell needs to be defined.

Given the various methods of improving the quality of results, here is how can this design be improved:

Module partitioning—Partition design in decode stage, execute stage, memory stage, and write back stage. The results from each stage are stored in a register.

Adding structure—Structured versions of various SPARC components have already been defined: arithmetic shift right, shift left, integer condition code evaluation. The critical path can be analyzed and structured versions can be added.

Horizontal partitioning—A bit-sliced carry save adder design can be used.

Performing operations in parallel—The condition code generation can be made faster by performing this computation in parallel with the result computation.

Use multiplexers for logic implementation—Use a multiplexer for evaluating the integer condition codes (see previous example).

Conclusion

By judicious partitioning of the design, using various combinations of horizontal and vertical partitioning as well as the addition of hierarchy, a designer can control the synthesis process. Horizontal partitioning breaks up a design into smaller slices that are more easily synthesized. Vertical partitioning tries to keep an entire critical path in one module and tries to register outputs of modules. The addition of hierarchy preserves a user-specified structure. It is possible to achieve good results and still have a technology-transparent design by using user-defined macrocell libraries, the GTECH library, and DesignWare. These hand-crafted libraries become the building blocks for success. Another guideline is to start simple and refine the structure as needed to meet speed and area objectives. This enables timing analysis and subsequent design changes to be made early in the design process. Verilog constructs should be used in a manner that prevents simulation mismatch between RTL and gate-level versions of the design. You should have complete sensitivity lists, use non-blocking assignments, and don’t mix blocking and non-blocking assignments. When using `always` blocks, be sure to either specify all the default assignments at the front of the block or check to insure all variables are assigned a value regardless of the input combinations. For `case` statements, the use of a default clause may not be sufficient. The use of procedural assignments within sequential blocks can simplify the code because default assignments are not needed. However, only registered outputs can be used in a sequential block; nonregistered outputs must be specified either with a continuous assignment or in a combinational `always` block.