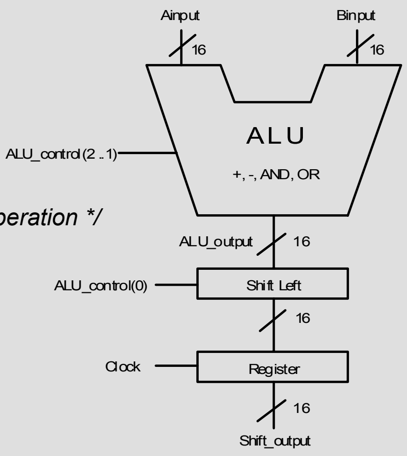
CHAPTER 7

Using Verilog for Synthesis of Digital Hardware

```
module ALU ( ALU_control, Ainput, Binput, Clock, Shift_output);
        input [2:0] ALU_control;
        input [15:0] Ainput;
        input [15:0] Binput;
        input Clock;
        output[15:0] Shift_output;
        reg [15:0] Shift_output;
        reg [15:0] ALU_output;

                /* Select ALU Arithmetic/Logical Operation */
always @(ALU_control or Ainput or Binput)
        case (ALU_control[2:1])
                0: ALU_output = Ainput + Binput;
                1: ALU_output = Ainput - Binput;
                2: ALU_output = Ainput & Binput;
                3: ALU_output = Ainput | Binput;
                default: ALU_output = 0;
         endcase

                /* Shift bits left using shift left operator if required and load register */
always @(posedge Clock)
        if (ALU_control[0]==1)
                Shift_output = ALU_output << 1;
        else
                Shift_output = ALU_output;
endmodule
```



7 Using Verilog for Synthesis of Digital Hardware

Verilog is another language that, like VHDL, is widely used to model and design digital hardware. In the early years, Verilog was a proprietary language developed by one CAD vendor, Gateway. Verilog was developed in the 1980's and was initially used to model high-end ASIC devices. In 1990, Verilog was released into the public domain, and Verilog now is the subject of IEEE standard 1364. Today, Verilog is supported by numerous CAD tool and programmable logic vendors. Verilog has a syntax style similar to the C programming language. Schools are more likely to cover VHDL since it was in the public domain several years earlier; however, in the FPGA industry, VHDL and Verilog have an almost equal market share for new design development.

Conventional programming languages are based on a sequential operation model. Digital hardware devices by their very nature operate in parallel. This means that conventional programming languages cannot accurately describe or model the operation of digital hardware since they are based on the sequential execution of statements. Like VHDL, Verilog is designed to model parallel operations.

IT IS **CRUCIAL** TO REMEMBER THAT VERILOG MODULES AND CONCURRENT STATEMENTS ALL OPERATE IN PARALLEL.

In this section, a brief introduction to Verilog for logic synthesis will be presented. It is assumed that the reader is already familiar with basic digital logic devices and some basic C syntax.

Whenever you need help with Verilog syntax, Verilog templates of common statements are available in the Quartus II online help. In the text editor, just click the right mouse button and **Insert** ⇨ **Templates** select Verilog.

7.1 Verilog Data Types

For logic synthesis, Verilog has simple data types. The net data type, wire, and the register data type, reg. A model with a net data type, wire, has a corresponding electrical connection or wire in the modeled device. Type reg is updated under the control of the surrounding procedural flow constructs typically inside an always statement. Type reg does not necessarily imply that the synthesized hardware for a signal contains a register, digital storage device, or flip-flop. It can also be purely combinational logic.

Table 7.1 lists the Verilog operators and their common function in Verilog synthesis tools.

7.2 Verilog Based Synthesis of Digital Hardware

Verilog can be used to construct models at a variety of abstraction levels such as structural, behavioral, register transfer level (RTL), and timing. An RTL model of a circuit described in Verilog describes the input/output relationship in terms of dataflow operations on signal and register values. If registers are required, a synchronous clocking scheme is normally used. Sometimes an RTL model is also referred to as a dataflow-style model.

Verilog simulation models often include physical device time delays. In Verilog models written for logic synthesis, timing information should not be provided.

For timing simulations, the CAD tools automatically include the actual timing delays for the synthesized logic circuit. An FPGA timing model supplied by the CAD tool vendor is used to automatically generate the physical device time delays inside the FPGA. Sometimes this timing model is also written in Verilog. For a quick overview of Verilog, several constructs that can be used to synthesize common digital hardware devices will be presented.

7.3 Verilog Operators

Table 7.1 lists the Verilog operators and their common function in Verilog synthesis tools.

Table 7.1 Verilog Operators.

Verilog Operator	Operation
+	Addition
-	Subtraction
*	Multiplication*
/	Division*
%	Modulus*
{ }	Concatenation – used to combine bits
<<	rotate left
>>	rotate right
=	equality
!=	Inequality
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
!	logical negation
&&	logical AND
||	logical OR
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Negation

*Not supported in some Verilog synthesis tools. In the Quartus II tools, multiply , divide, and mod of integer values is supported. Efficient design of multiply or divide hardware may require the user to specify the arithmetic algorithm and design in Verilog.

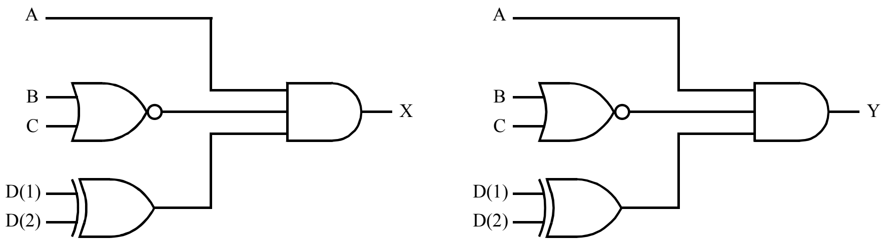
7.4 Verilog Synthesis Models of Gate Networks

The first example consists of a simple gate network. In this model, both a concurrent assignment statement and a sequential always block are shown that generate the same gate network. X is the output on one network and Y is the output on the other gate network. The two gate networks operate in parallel.

In Verilog synthesis, inputs and outputs from the module will become I/O pins on the programmable logic device. For comments "//" makes the rest of a line a comment and "/*" and "*/" can be used to make a block of lines a comment.

The Quartus II editor performs syntax coloring and is useful to quickly find major problems with Verilog syntax. Verilog is case sensitive just like C.

Verilog concurrent statements are executed in parallel. Inside an always statement, statements are executed in sequential order, and all of the always statements are executed in parallel. The *always* statement is Verilog's equivalent of a *process* in VHDL.

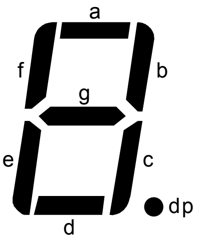


```
module gatenetwork(A, B, C, D, X, Y);
   input A;
   input B;
   input C;
   input [2:1] D;
   output X, Y;
   reg Y;
         // concurrent assignment statement
   wire X = A & ~(B|C) & (D[1] ^ D[2]);
         /* Always concurrent statement- sequential execution inside */
   always @( A or B or C or D)
         Y = A & ~(B|C) & (D[1] ^ D[2]);

endmodule
```

7.5 Verilog Synthesis Model of a Seven-segment LED Decoder

The following Verilog code implements a seven-segment decoder for seven-segment LED displays. A 7-bit vector is used to assign the value of all seven bits in a single case statement. In the 7-bit logic vector, the most-significant bit is segment 'a' and the least-significant bit is segment 'g'. The logic synthesis CAD tool automatically minimizes the logic required for implementation. The signal Hex_digit contains the 4-bit binary value to be displayed in hexadecimal.

```
module DEC_7SEG(Hex_digit, segment_a, segment_b, segment_c,
          segment_d, segment_e, segment_f, segment_g);
    input [3:0] Hex_digit;
    output segment_a, segment_b, segment_c, segment_d;
    output segment_e, segment_f, segment_g;
    reg [6:0] segment_data;

    always @(Hex_digit)
              /* Case statement implements a logic truth table using gates*/
     case (Hex_digit)
          4'b 0000:   segment_data = 7'b 1111110;
          4'b 0001:   segment_data = 7'b 0110000;
          4'b 0010:   segment_data = 7'b 1101101;
          4'b 0011:   segment_data = 7'b 1111001;
          4'b 0100:   segment_data = 7'b 0110011;
          4'b 0101:   segment_data = 7'b 1011011;
          4'b 0110:   segment_data = 7'b 1011111;
          4'b 0111:   segment_data = 7'b 1110000;
          4'b 1000:   segment_data = 7'b 1111111;
          4'b 1001:   segment_data = 7'b 1111011;
          4'b 1010:   segment_data = 7'b 1110111;
          4'b 1011:   segment_data = 7'b 0011111;
          4'b 1100:   segment_data = 7'b 1001110;
          4'b 1101:   segment_data = 7'b 0111101;
          4'b 1110:   segment_data = 7'b 1001111;
          4'b 1111:   segment_data = 7'b 1000111;
          default:    segment_data = 7'b 0111110;
     endcase
```



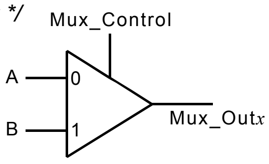
The following Verilog concurrent assignment statements extract the seven 1-bit values needed to connect the individual segments. The not operator (~) is used since a logic zero actually turns on most LEDs. Automatic minimization in the synthesis process will eliminate the extra inverter in the logic circuit.

```
                            /* extract segment data bits and invert */
                            /* LED driver circuit is inverted */
   wire segment_a = ~segment_data[6];
   wire segment_b = ~segment_data[5];
   wire segment_c = ~segment_data[4];
   wire segment_d = ~segment_data[3];
   wire segment_e = ~segment_data[2];
   wire segment_f =  ~segment_data[1];
   wire segment_g = ~segment_data[0];
endmodule
```

7.6 Verilog Synthesis Model of a Multiplexer

The next example shows several alternative ways to synthesize a 2-to-1 multiplexer in Verilog. Three identical multiplexers that operate in parallel are synthesized by this example. The wire conditional continuous assignment statement can be used for a 2-to-1 mux. A concurrent assign statement can also be used instead of wire, if the output signal is already declared. In Verilog, IF and CASE statements must be inside an always statement. The inputs and outputs from the multiplexers could be changed to bit vectors if an entire bus is multiplexed. Multiplexers with more than two inputs can also be easily constructed and a case statement is preferred. Nested IF statements generate priority-encoded logic that requires more hardware and produce a slower circuit than a CASE statement.

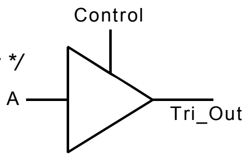
```
                /* Multiplexer example shows three ways to model a 2 to 1 mux */
module multiplexer(A, B, mux_control, mux_out1, mux_out2, mux_out3);
  input A;                    /* Input Signals and Mux Control  */
  input B;
  input mux_control;
  output mux_out1,mux_out2, mux_out3;
  reg mux_out2, mux_out3;
                              /* Conditional Continuous Assignment Statement */
                              /*   works like an IF - ELSE */
  wire mux_out1 = (mux_control)? B:A;
                              /* If statement inside always statement */
    always @(A or B or mux_control)
        if (mux_control)
            mux_out2 = B;
        else
            mux_out2 = A;
                              /* Case statement inside always statement */
    always @(A or B or mux_control)
        case (mux_control)
            0: mux_out3 = A;
            1: mux_out3 = B;
            default: mux_out3 = A;
        endcase
endmodule
```



7.7 Verilog Synthesis Model of Tri-State Output

Tri-state gates are supported in Verilog synthesis tools and are supported in many programmable logic devices. Most programmable logic devices have tri-state output pins. Some programmable logic devices do not support internal tri-state logic. Here is a Verilog example of a tri-state output. In Verilog, the assignment of the value "Z" to a signal produces a tri-state output.

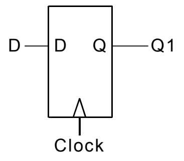
```
  module tristate (a, control, tri_out);
    input a, control;
    output tri_out;
    reg tri_out;
    always @(control or a)
        if (control)
            /* Assignment of Z value generates a tri-state output */
            tri_out = 1'bZ;
        else
            tri_out = a;
  endmodule
```

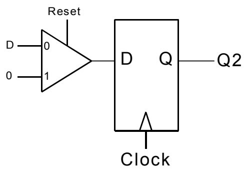


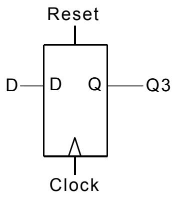
7.8 Verilog Synthesis Models of Flip-flops and Registers

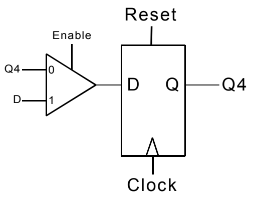
In the next example, several flip-flops will be generated. Unlike earlier combinational hardware devices, a flip-flop can only be synthesized inside an always statement. The positive clock edge is selected by **posedge clock** and positive edge triggered D flip-flops will be used for synthesis. The following module contains a variety of Reset and Enable options on positive edge-triggered D flip-flops. The negative clock edge is selected by **negedge clock** and negative edge-triggered D flip-flops used during synthesis.

```
module DFFs(D, clock, reset, enable, Q1, Q2, Q3, Q4);
input D;
input clock;
input reset;
input enable;
output Q1, Q2, Q3, Q4;
reg Q1, Q2, Q3, Q4;
                      /* Positive edge triggered D flip-flop */
    always @(posedge clock)
          Q1 = D;
                      /* Positive edge triggered D flip-flop */
                      /*    with synchronous reset */
    always @(posedge clock)
          if (reset)
             Q2 = 0;
          else
             Q2 = D;
                      /* Positive edge triggered D flip-flop */
                      /*    with asynchronous reset */
    always @(posedge clock or posedge reset)
          if (reset)
             Q3 = 0;
          else
             Q3 = D;
                      /* Positive edge triggered D flip-flop */
                      /* with asynchronous reset and enable */
    always @(posedge clock or posedge reset)
          if (reset)
             Q4 = 0;
          else if (enable)
             Q4 = D;
endmodule
```









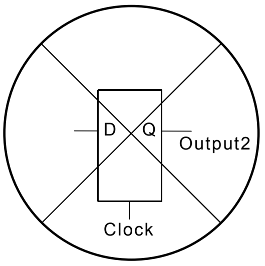
In Verilog, as in any digital logic designs, it is not good design practice to AND or gate other signals with the clock. Use a flip-flop with a clock enable instead to avoid timing and clock skew problems. In some limited cases, such as power management, a single level of clock gating can be used. This works only when a small amount of clock skew can be tolerated and the signal gated with the clock is known to be hazard or glitch free. A particular programmable logic device may not support every flip-flop or latch type and all of the Set/Reset and Enable options.

If D and Q are replaced by bit vectors in any of these examples, registers with the correct number of bits will be generated instead of individual flip-flops.

7.9 Accidental Synthesis of Inferred Latches

Here is a very common problem to be aware of when coding Verilog for synthesis. If a non-clocked process has any path that does not assign a value to an output, Verilog assumes you want to use the previous value. A level triggered latch is automatically generated or inferred by the synthesis tool to save the previous value. In many cases, this can cause serious errors in the design. Edge-triggered flip-flops should not be mixed with level-triggered latches in a design or serious timing problems will result. Typically this can happen in CASE statements or nested IF statements. In the following example, the signal Output2 infers a latch when synthesized. Assigning a value to Output2 in the last ELSE clause will eliminate the inferred latch. Warning messages may be generated during compilation when a latch is inferred on some tools. Note the use of *begin…end* is somewhat different than the use of braces in C.

```
module ilatch( A, B, Output1, Output2);
input A, B;
output Output1, Output2;
reg Output1, Output2;

always@( A or B)
     if (!A)
         begin
                 Output1 = 0;
                 Output2 = 0;
         end
     else
         if (B)
              begin
                  Output1 = 1;
                  Output2 = 1;
              end
         else                       /*latch inferred since no value */
                 Output1 = 0;       /*is assigned to Output2 here */
endmodule
```



7.10 Verilog Synthesis Model of a Counter

Here is an 8-bit counter design. Compare operations such as "<" are supported and they generate a comparator logic circuit to test for the maximum count value. The assignment count = count+1; synthesizes an 8-bit incrementer. An incrementer circuit requires less hardware than an adder that adds one. The operation, "+1", is treated as a special incrementer case by synthesis tools.

```
module counter(clock, reset, max_count, count);
        input clock;
        input reset;
        input  [7:0] max_count;
        output [7:0] count;
        reg [7:0] count;
                        /* use positive clock edge for counter */
        always @(posedge clock or posedge reset)
                begin
                        if (reset)
                                count = 0;                  /* Reset  Counter */
                        else if (count < max_count)         /* Check for maximum count */
                                count = count + 1;          /* Increment Counter */
                        else
                                count = 0;                  /*  Counter set back to 0*/
                end
        endmodule
```

7.11 Verilog Synthesis Model of a State Machine

The next example shows a Moore state machine with three states, two inputs and a single output. A state diagram of the example state machine is shown in Figure 7.1. Unlike VHDL, A direct assignment of the state values is required in Verilog's parameter statement. The first Always block assigns the next state using a case statement that is updated on the positive clock edge, posedge.

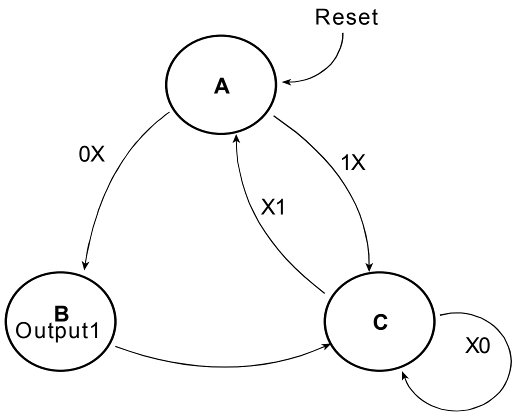


Figure 7.1 State Diagram for state_mach Verilog example

```
module state_mach (clk, reset, input1, input2 ,output1);
        input clk, reset, input1, input2;
        output output1;
        reg output1;
        reg [1:0] state;
```

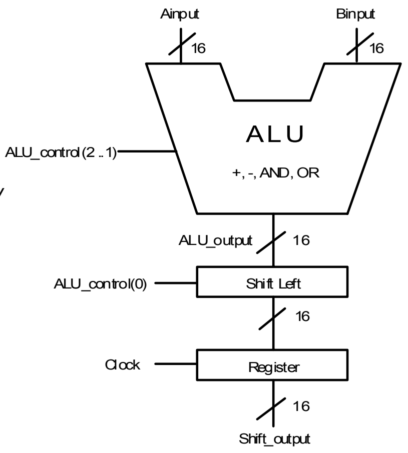
```
                    /* Make State Assignments */
        parameter [1:0] state_A = 0, state_B = 1, state_C = 2;

    always@(posedge clk or posedge reset)
        begin
            if (reset)
                state = state_A;
            else
                    /* Define Next State Transitions using a Case */
                    /* Statement based on the Current State */
                case (state)
                    state_A:
                        if (input1==0)
                            state = state_B;
                        else
                            state = state_C;
                    state_B:
                        state = state_C;
                    state_C:
                        if (input2) state = state_A;
                    default:  state = state_A;
                endcase
        end
                    /* Define State Machine Outputs */
    always @(state)
        begin
            case (state)
                state_A: output1 = 0;
                state_B: output1 = 1;
                state_C: output1 = 0;
                default:  output1 = 0;
            endcase
        end
    endmodule
```

7.12 Verilog Synthesis Model of an ALU with an Adder/Subtractor and a Shifter

Here is an 8-bit arithmetic logic unit (ALU) that adds, subtracts, bitwise ANDs, or bitwise ORs, two operands and then performs an optional shift on the output. The most-significant two bits of the Op-code select the arithmetic logical operation. If the least-significant bit of the op_code equals ‘1’ a 1-bit left-shift operation is performed. An addition and subtraction circuit is synthesized for the "+" and "-" operator.

Depending on the number of bits and the speed versus area settings in the synthesis tool, ripple carry or carry-lookahead circuits will be used. Several "+" and "-" operations in multiple assignment statements may generate multiple ALUs and increase the hardware size, depending on the Verilog CAD tool and compiler settings used. If a single ALU is desired, muxes can be placed at the inputs and the "+" operator would be used only in a single assignment statement.

```
module ALU ( ALU_control, Ainput, Binput, Clock, Shift_output);
    input [2:0] ALU_control;
    input [15:0] Ainput;
     input [15:0] Binput;
    input Clock;
    output[15:0] Shift_output;
    reg [15:0] Shift_output;
    reg [15:0] ALU_output;

        /* Select ALU Arithmetic/Logical Operation */
always @(ALU_control or Ainput or Binput)
    case (ALU_control[2:1])
        0: ALU_output = Ainput + Binput;
        1: ALU_output = Ainput - Binput;
        2: ALU_output = Ainput & Binput;
        3: ALU_output = Ainput | Binput;
        default: ALU_output = 0;
     endcase

        /* Shift bits left using shift left operator if required and load register */
always @(posedge Clock)
    if (ALU_control[0]==1)
        Shift_output = ALU_output << 1;
    else
        Shift_output = ALU_output;
endmodule
```



7.13 Verilog Synthesis of Multiply and Divide Hardware

In the Quartus II tool, integer multiply and divide is supported using Verilog's "*" and "/" operators. In current generation tools, efficient design of multiply or divide hardware typically requires the use of a vendor-specific library function or even the specification of the arithmetic algorithm and hardware implementation in Verilog.

A wide variety of multiply and divide algorithms that trade off time versus hardware size can be found in most computer arithmetic texts. Several such references are listed at the end of this chapter. These algorithms require a sequence of add/subtract and shift operations that can be easily synthesized in Verilog using the standard operators. The LPM_MULT function in Quartus II can be used to synthesize integer multipliers. LPM_DIVIDE, is also available. When using LPM functions, **Tools ⇨ MegaWizard Plug-in Manager** can be used to help generate Verilog code. The LPM functions also support pipeline options. Array multiply and divide hardware for more than a few bits requires extensive hardware and a large FPGA. A few large FPGAs now contain multiplier blocks.

```
module mult (dataa, datab, result);
      input   [7:0]  dataa;
      input   [7:0]  datab;
      output [15:0]  result;

      wire [15:0] sub_wire0;
      wire [15:0] result = sub_wire0[15:0];
                        /* Altera LPM 8x8 multiply function  result = dataa * datab */
      lpm_mult lpm_mult_component (
                  .dataa (dataa),
                  .datab (datab),
                  .result (sub_wire0) );
      defparam
        lpm_mult_component.lpm_widtha = 8,
        lpm_mult_component.lpm_widthb = 8,
        lpm_mult_component.lpm_widthp = 16,
        lpm_mult_component.lpm_widths = 1,
        lpm_mult_component.lpm_type = "LPM_MULT",
        lpm_mult_component.lpm_representation = "UNSIGNED",

endmodule
```

Floating-point operations can be implemented on very large FPGAs; however, performance is lower than current floating-point DSP and microprocessor chips. The floating-point algorithms must be coded by the user in Verilog using integer add, multiply, divide, and shift operations. The LPM_CLSHIFT function is useful for the barrel shifter needed in a floating-point ALU. Some floating point IP cores are starting to appear. Many FPGA vendors also have optimized arithmetic packages for DSP applications such as FIR filters.

7.14 Verilog Synthesis Models for Memory

Typically, it is more efficient to call a vendor-specific function to synthesize RAM. These functions typically use the FPGA's internal RAM blocks rather than building a RAM using FPGA logic elements. The memory function in the Altera toolset is the ALTSYNCRAM function. On the UP2 board's older FPGA, the LPM_RAM_DQ memory function should be used. The memory can be set to an initial value using a separate memory initialization file with the extension *.mif. A similar call, LPM_ROM, can be used to synthesize ROM.

If small blocks of multi-ported or other special-purpose RAM are needed, they can be synthesized using registers with address decoders for the write operation and multiplexers for the read operation. Additional read or write ports can be added to synthesize RAM. An example of this approach is a dual-ported register file for a computer processor core. Most RISC processors need to read two registers on each clock cycle and write to a third register.

Verilog Memory Model - Example One

The first memory example synthesizes a memory that can perform a read and a write operation every clock cycle. Memory is built using arrays of positive edge-triggered D flip-flops. Memory write, memwrite, is gated with an address decoder output and used as an enable to load each memory location during a write operation. A synchronous write operation is more reliable. Asynchronous write operations respond to any logic hazards or momentary level changes on the write signal. As in any synchronous memory, the write address must be stable before the rising edge of the clock signal. A non-clocked mux is used for the read operation. If desired, memory can be initialized by a reset signal.

```
module memory(read_data, read_address, write_data, write_address,
                    memwrite, clock, reset);
output [7:0] read_data;
input [2:0] read_address;
input [7:0] write_data;
input [2:0] write_address;
input memwrite;
input clock;
input reset;
reg [7:0] read_data, mem0, mem1;

              /* Block for memory read */
always @(read_address or mem0 or mem1)
    begin
        case(read_address)
            3'b 000: read_data = mem0;
            3'b 001: read_data = mem1;
            /* Unimplemented memory */
            default: read_data = 8'h FF;
        endcase
    end

              /* Block for memory write */
always @(posedge clock or posedge reset)
    begin
     if (reset)
        begin
            /* Initial values for memory (optional) */
            mem0 = 8'h AA ;
            mem1 = 8'h 55;
        end
     else if (memwrite)
            /* write new value to memory */
        case (write_address)
            3'b 000 : mem0 = write_data;
            3'b 001 : mem1 = write_data;
        endcase
    end
endmodule
```

Verilog Memory Model - Example Two

The second example shows the use of Altera's ALTSYNCRAM megafunction to implement a block of memory. For more information on the megafunctions see the online help guide in the Quartus II tool. In single port mode, the ALTSYNCRAM memory can do either a read or a write operation in a single clock cycle since there is only one address bus. In dual port mode, it can do both a read and write. If this is the only memory operation needed, the ALTSYNCRAM function produces a more efficient hardware implementation than synthesis of the memory in Verilog. In the ALTSYNCRAM megafunction, the memory address must be clocked into a dedicated address register located inside the FPGA's synchronous memory block. Asynchronous memory operations without a clock can cause timing problems and are not supported on many FPGAs including the Cyclone.

```
module amemory ( write_data, write_enable, address, clock, read_data);

input [7:0]  write_data;
input  write_enable;
input [2:0]  address;
input  clock;
output [7:0]  read_data;
wire [7:0] sub_wire0;
wire [7:0] read_data = sub_wire0[7:0];
                    /* Use Altera Altsyncram function for memory */
altsyncram  altsyncram_component (
             .wren_a (write_enable),
             .clock0 (clock),
             .address_a (address),
             .data_a (write_data),
             .q_a (sub_wire0));
defparam
     altsyncram_component.operation_mode = "SINGLE_PORT",
                    /* 8 data bits, 3 address bits, and no register on read data */
     altsyncram_component.width_a = 8,
     altsyncram_component.widthad_a = 3,
     altsyncram_component.outdata_reg_a = "UNREGISTERED",
                    /* Reads in mif file for initial memory data values (optional) */
     altsyncram_component.init_file = "memory.mif";
endmodule
```

On the Cyclone FPGA chip, the memory can be implemented using the M4K memory blocks, which are separate from the FPGA's logic cells. In the Cyclone EP1C6 chip there are 20 M4K RAM blocks at 4Kbits each for a total of 92,160 bits. In the Cyclone EP1C12 there are 52 M4K blocks for a total of 239,616 bits. Each M4K block can be setup to be 4K by 1, 2K by 2, 1K by 4, 512 by 8, 256 by 16,256 by 18, 128 by 32 or 128 by 36 bits wide. The **Tools ⇨ Megawizard Plug-in Manager** feature is useful to configure the Altsyncram parameters.

7.15 Hierarchy in Verilog Synthesis Models

Large Verilog models should be split into a hierarchy using a top-level structural model in Verilog or by using the symbol and graphic editor in the Quartus II tool. In the graphical editor, a Verilog file can be used to define the contents of a symbol block. Synthesis tools run faster using a hierarchy on large models and it is easier to write, understand, and maintain a large design when it is broken up into smaller modules.

An example of a hierarchical design with three submodules is seen in the schematic in Figure 7.2. Following the schematic, the same design using a top-level Verilog structural model is shown. This Verilog structural model provides the same connection information as the schematic seen in Figure 7.2.

Debounce, Onepulse, and Clk_div are the names of the Verilog submodules. Each one of these submodules has a separate Verilog source file. In the Quartus II tool, compiling the top-level module will automatically compile the lower-level modules.

In the example Verilog structural-model example for Figure 7.2, note the use of a component instantiation statement for each of the three submodules. The component instantiation statement declares the module name and connects inputs and outputs of the module. New internal signal names used for interconnections of modules should also be declared at the beginning of the top level module.

The order of each module’s signal names must be the same as in the Verilog submodule files. Each instantiation of a module is given a unique name so that a single module can be used several times. As an example, the single instantiation of the debounce module is called debounce1 in the example code.

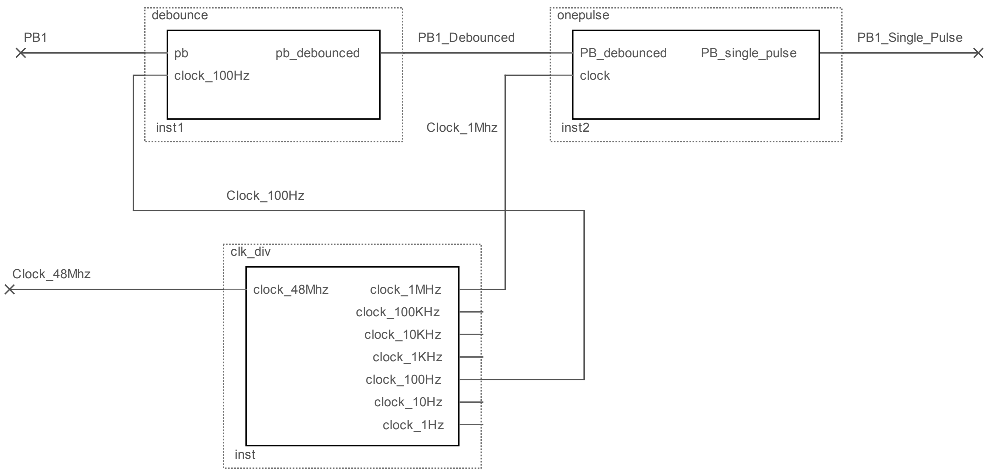


Figure 7.2 Schematic of Hierarchical Design Example

Note that node names in the schematic or signals in Verilog used to interconnect modules need not always have the same names as the signals in the components they connect. As an example, PB_debounced on the debounce component connects to an internal signal with a different name, PB1_debounced.

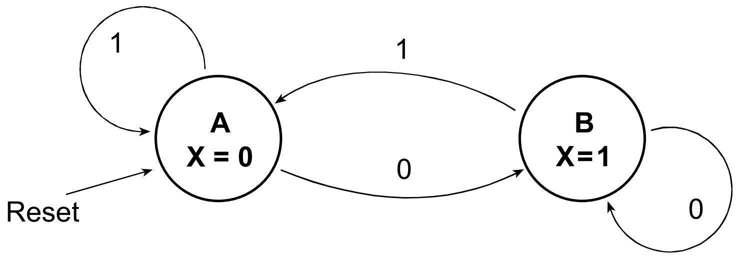
```
module hierarch(Clock_48MHz, PB1, PB1_Single_Pulse);
  input Clock_48MHz, PB1;
  output PB1_Single_Pulse;
              /* Declare internal interconnect signals */
  reg Clock_100Hz, Clock_1MHz, PB1_Debounced;

              /* declare and connect all  three modules in the hierarchy */
   debounce debounce1( PB1, Clock_100Hz, PB1_Debounced);

   clk_div clk_div1( Clock_48MHz, Clock_1MHz, Clock_100Hz);

   onepulse onepulse1( PB1_Debounced, Clock_100Hz, PB1_Single_Pulse);

endmodule
```

7.16 For additional information

The chapter has introduced the basics of using Verilog for digital synthesis. It has not explored all of the language options available. The Altera online help contains Verilog syntax and templates. A number of Verilog reference textbooks are also available. Unfortunately, not all of them currently contain Verilog models that can be used for digital logic synthesis. Two recommendations are *HDL Chip Design* by Douglas J. Smith, Doone Publications, 1996 and *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL* by Michael Ciletti, 1999. An interesting free VHDL to Verilog conversion program is also available at www.ocean-logic.com/downloads.htm.

7.17 Laboratory Exercises

1. Write a Verilog model for the state machine shown in the following state diagram and verify correct operation with a simulation using the Altera CAD tools. A and B are the two states, X is the output, and Y is the input. Use the timing analyzer to determine the maximum clock frequency on the Cyclone EP1C6Q240C8 device.



2. Write a Verilog model for a 32-bit, arithmetic logic unit (ALU). Verify correct operation with a simulation using the Altera CAD tools. A and B are 32-bit inputs to the ALU, and Y is the output. A shift operation follows the arithmetic and logical operation. The opcode controls ALU functions as follows:

Opcode	Operation	Function
000XX	ALU_OUT <= A	Pass A
001XX	ALU_OUT <= A + B	Add
010XX	ALU_OUT <= A-B	Subtract
011XX	ALU_OUT <= A AND B	Logical AND
100XX	ALU_OUT <= A OR B	Logical OR
101XX	ALU_OUT <= A + 1	Increment A
110XX	ALU_OUT <= A-1	Decrement A
111XX	ALU_OUT <= B	Pass B
XXX00	Y <= ALU_OUT	Pass ALU_OUT
XXX01	Y<= SHL(ALU_OUT)	Shift Left
XXX10	Y<=SHR(ALU_OUT)	Shift Right (unsigned-zero fill)
XXX11	Y<= 0	Pass 0's

3. Use the Cyclone chip as the target device. Determine the worst case time delay of the ALU using the timing analyzer. Examine the report file and find the device utilization. Use the logic element (LE) device utilization percentage found in the compilation report to compare the size of the designs.

4. Explore different synthesis options for the ALU from problem 3. Change the area and speed synthesis settings in the compiler under **Assignments ⇨Settings ⇨Analysis and Synthesis Settings**, rerun the timing analyzer to determine speed, and examine the report file for hardware size estimates. Include data points for the default, optimized for speed, balanced, and optimized for area settings. Build a plot showing the speed versus area trade-offs possible in the synthesis tool. Use the logic element (LE) device utilization percentage found in the compilation report to compare the size of the designs.

5. Develop a Verilog model of one of the TTL chips listed below. The model should be functionally equivalent, but there will be timing differences. Compare the timing differences between the Verilog FPGA implementation and the TTL chip. Use a data book or find a data sheet using the World Wide Web.

   F. 7400 Quad nand gate

   G. 74LS241 Octal buffer with tri-state output

   H. 74LS273 Octal D flip-flop with Clear

   I. 74163 4-bit binary counter

   J. 74LS181 4-bit ALU

6. Replace the 8count block used in the tutorial in Chapter 4, with a new counter module written in Verilog. Simulate the design and download a test program to the UP3 board.

7. Implement a 128 by 32 RAM using Verilog and the Altsyncram function. Do not use registered output options. Target the design to the Cyclone II device. Use the timing analysis tools to determine the worst-case read and write access times for the memory.