Appendix A. Verilog Examples

A.1 Combinational Logic Structures

Continuous assignment statements are a very useful and compact language structure for specifying small collections of gates. The following examples are intended to go beyond those provided in the text and illustrate concepts that sometimes are found to be difficult.

Multi-Bit Gates

Arrays can be used to concisely specify multi-bit variables. Sets of gates can be specified to operate on all the bits in a bus, or just a subset. For example:

```
module multi1 (in1, in2; out1, out2);
input [3:0] in1, in2;
output [3:0] out1, out2;
wire [3:0] out1;
wire [1:0] out2;
assign out1 = in1 & in2;
assign out2[1:0] = in1[3:2] ^ in2[1:0];
endmodule
```

Note the following points:
- Sub-fields of arrays of bits can be specified on either side of the assignment.
- The operators, & and | are used rather than && and ||, as the former are intended to model gates while the latter are intended to construct boolean algebra on single-bit true-false variables.
- Technically the ‘wire’ declarations are not needed here as out1 and out2 are outputs and all outputs are assumed to be of a ‘wire’ type unless otherwise declared.

The following example specifies some more complicated multi-bit operators:

```
module multi2 (in1, in2, out1, out2);
input [7:0] in1;
input [1:0] in2;
output [1:0] out1;
output out2;
assign out1 = (in1 >> 2) | in2;
assign out2 = in1[in2];
endmodule
```

Note the following with respect to this example:
- The right shift operator, “>>” simply rearranges the wires so that bit 2 of in1 becomes bit 0, bit 3 becomes
bit 1, etc. Then since out1 is only two bits wide, only the lower two bits of out1 are used. The following statement would produce an identical result:

```
assign out1 = in1[3:2] | in2;
```

- In the second statement, a single bit of an eight-bit variable, in1, is selected by the value of the 3-bit variable, in2, thus specifying an 8:1 multiplexor. This is a simple way to specify larger muxes than can be produced using the sel statement.

**Unary Reduction Operators**

Unary reduction operators always produce one-bit outputs from multi-bit inputs. Consider the following example:

```vhdl
module (in1, in2, out1, out2);

input [2:0] in1;
input       in2;
output      out1, out2;
wire out1, out2;

assign out1 = |{in1, in2};
assign out2 = ^in1;
endmodule
```

Note the following:
- The use of concatenation {} to produce a 4-input gate.
- That a multi-input XOR gate is built out of a series of 2-input gates.

**Multiple-input Multiplexors**

Multiplexors requiring more than two inputs can also be specified using procedural code, usually by using a case or casex statement. The 8:1 multiplexor described in the example above can be specified procedurally as follows:

```vhdl
module (in1, in2, out2);

input [7:0] in1;
input [2:0] in2;
output out2;

always@(in1 or in2)
case (in2)
  2'b000 : out2 = in1[0];
  2'b001 : out2 = in1[1];
  2'b010 : out2 = in1[2];
  2'b011 : out2 = in1[3];
  2'b100 : out2 = in1[4];
  2'b101 : out2 = in1[5];
  2'b110 : out2 = in1[6];
  2'b111 : out2 = in1[7];
endcase
endmodule
```
Demultiplexor

A demultiplexor is the converse of a multiplexor. It takes one input and directs it to any of $N$ inputs, based on the number specified in the select lines. It could be captured either using procedural code or continuous assignment. Both of the following statements describe identical behavior.

```verilog
module (in1, sel, out2);

input [1:0] in1;
input [2:0] sel;
output [13:0] out2;

reg [15:0] out2;
integer I;

always@(in1 or sel)
begin
    out2 = 14'h0; /* default = 00 */
    for (I=0; I<=7; I=I+1)
        if (I == sel)
            begin
                out2[I] = in1[0];
                out2[I+1] = in1[1];
            end
end
endmodule

/*----------------------------*/
module (in1, sel, out2);

input [1:0] in1;
input [2:0] sel;
output [15:0] out2;
reg [7:0] select;
/* address decoder */
always@(sel)
case (sel)
    3'b000 : select = 8'b00000001;
    3'b001 : select = 8'b00000010;
    3'b010 : select = 8'b00000100;
    3'b011 : select = 8'b00001000;
    3'b100 : select = 8'b00010000;
    3'b101 : select = 8'b00100000;
    3'b110 : select = 8'b01000000;
    3'b111 : select = 8'b10000000;
endcase
assign out2[1:0] = in1 & select[0];
assign out2[3:2] = in1 & select[1];
assign out2[5:4] = in1 & select[2];
assign out2[7:6] = in1 & select[3];
assign out2[9:8] = in1 & select[4];
assign out2[11:10] = in1 & select[5];
```
```verilog
assign out2[13:12] = in1 & select[6];
assign out2[15:14] = in1 & select[7];
endmodule
```

Note the following in this example:

- It is assumed that we want a demux output to be 00 unless selected. The default at the start of the
  procedural block in the first example provides for this AND prevents specifying unintended latches.
- The use of the for loop in the first version to iterate across the output bits.
- Verilog does not permit ports containing two dimensional arrays. Thus out2 is declared as 16 bits wide
  rather than 8 x 2 bits, as the designer would prefer to think of it. A bit is a bit is a bit, so this does not
  matter.
- The second version has more structure. It specifies an explicit 3:8 decoder that takes a 3-bit input and
  sets one of 8 lines, as per the number specified in the input. It uses AND gates for the actual demux.
  This design is likely to be smaller and faster than the first design. It could also be described using a for
  loop. That exercise is left to the reader.

**Decoder**

A decoder takes a narrow input and decodes it to a wider range of bits. Typically they are part of the
control logic. The commands to individual units are encoded so as to save wires and then a decoder is used
to determine the individual control bits.

The previous example included a non-priority 3:8 decoder. Three inputs were decoded to choose one of
eight control lines. It is non-priority as none of the input combinations has priority over any of the others.
This decoder implemented the following truth table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel</td>
<td>select</td>
</tr>
<tr>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>001</td>
<td>00000010</td>
</tr>
<tr>
<td>010</td>
<td>00000100</td>
</tr>
<tr>
<td>011</td>
<td>00001000</td>
</tr>
<tr>
<td>100</td>
<td>00010000</td>
</tr>
<tr>
<td>101</td>
<td>00100000</td>
</tr>
<tr>
<td>110</td>
<td>01000000</td>
</tr>
<tr>
<td>111</td>
<td>10000000</td>
</tr>
</tbody>
</table>

In contrast. in a priority decoder, certain combinations of inputs are given priority over others. For
example, consider the following truth table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>xxxxx</td>
</tr>
<tr>
<td>11</td>
<td>00011</td>
</tr>
<tr>
<td>11</td>
<td>00001</td>
</tr>
<tr>
<td>otherwise</td>
<td>otherwise</td>
</tr>
</tbody>
</table>

This truth table specifies that if Op=01, then the outputs are set independently of the value of Funct. Op has
priority over Funct. The last line of the table merely indicates that for values of Op and Funct not
specified, we don’t care what the outputs are. This permits the logic optimization routines in the synthesis
tools to do a better job than if we gave the outputs 0 or 1 values in this line. (The reader might recall that 'x's in the Karnaugh map permit greater optimization of the logic. That is all we are specifying here.)

The following Verilog module captures the above truth table:

```verilog
module pri_encoder (Op, Funct, Sel, B);
input [1:0] Op;
input [4:0] Funct;
output [1:0] Sel;
output B;

always@(Op or Funct)
casex ({Op, Funct})
  {2'b01, 5'bx } : begin
    Sel = 2'b11;
    B = 1'b1;
  end
  {2'b11, 5'b00011 } : begin
    Sel = 2'b01;
    B = 1'b1;
  end
  {2'b11, 5'b00001 } : begin
    Sel = 2'b10;
    B = 1'b1;
  end
  default : begin
    Sel = 2'bxx;
    B = 1'bx;
  end
endcase
endmodule
```

Note the use of casex when x-valued inputs are specified.

**Encoder**

An encoder takes a multi-bit input and encodes it as a different, often shorter, bit stream. As such it is implementing a truth table with fewer outputs than inputs and it is coded much the same way as a decoder. An example is a gray-code encoder. In a gray code encoder the position of the ‘1’ bit in a N-bit data stream is captured with a code in which only one bit changes between adjacent values in the sequence. The following truth table uses a gray code in the output:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>000</td>
</tr>
<tr>
<td>00000010</td>
<td>001</td>
</tr>
<tr>
<td>00000100</td>
<td>011</td>
</tr>
<tr>
<td>00001000</td>
<td>010</td>
</tr>
<tr>
<td>00010000</td>
<td>110</td>
</tr>
<tr>
<td>00100000</td>
<td>111</td>
</tr>
<tr>
<td>01000000</td>
<td>101</td>
</tr>
<tr>
<td>10000000</td>
<td>100</td>
</tr>
</tbody>
</table>

A verilog module to capture this truth table can be specified as follows:
module encoder (value, gray_code);

input [7:0] value;
output [3:0] gray_code;

always@(value)
  case (value)
    8'b00000001 : gray_code = 3'b000;
    8'b00000010 : gray_code = 3'b001;
    8'b00000100 : gray_code = 3'b011;
    8'b00001000 : gray_code = 3'b010;
    8'b00010000 : gray_code = 3'b110;
    8'b00100000 : gray_code = 3'b111;
    8'b01000000 : gray_code = 3'b101;
    8'b10000000 : gray_code = 3'b100;
  endcase
endmodule

Priority encoders, when required, can be captured using a casex statement.

Proper Initialization

Modeling hardware initialization in Verilog can sometimes be a little tricky. You might recall that in hardware design, one purpose of the reset is to initialize the hardware to a known state. Showing that the reset is working correctly in pre-synthesis Verilog can be a little tricky due to subtle bugs. The Verilog simulator initializes all variables to unknown or 'x'. Thus, one would think that if the simulation is showing an 'x' where 0 or 1 is expected after the reset has been asserted, then there is a bug in the reset design that needs to be fixed. Unfortunately, an incorrect 'x' can be turned into a 0 or 1 and look correct, even when it is not.

Consider the following code fragment (for an encoder):

always@(a or b)
  case (a)
    2'b00 : b = 4'b0001;
    2'b01 : b = 4'b0010;
    2'b10 : b = 4'b0100;
    default : b = 4'b1000;
  endcase

At first sight, this looks fine. Let’s assume that a is meant to be initialized to 11 by the reset, and is thus propagated through this logic block to initialize b to 1000. However, what if there is a hardware bug that results in ‘a’ not being assigned to any 1-0 combination on reset? The Verilog simulator will leave a=xx after reset. However, when ‘a’ is propagated through the logic above, a=xx will catch the default and b will be set to 1000. The signal b will look fine and the fact that a is incorrect will not be caught. (If this is not caught before the hardware is made then a and b will actually take on some reandom combination of 0’s and 1’s, depenning on the gate details – ‘x’s don’t occur in the actual hardware.) A coding style that will catch this particular bug is as follows:

always@(a or b)
  caseex (a)
    2'b00 : b = 4'b0001;
    2'b01 : b = 4'b0010;
    2'b10 : b = 4'b0100;
    2'b11 : b = 4'b1000;
    default : b = 4'bxxxx;
endcase

Here, if `a` contains unknowns (x), then `b` will appear as x and the incorrect reset design is more likely to be caught. The case statement becomes a `casex` to ensure that an 'x' input is recognized.

**Tri-State Buffers and Buses**

In order to reduce the amount of overall wiring required, often multiple units share a common bus in order to communicate amongst themselves. On such a bus it is important that during any particular clock cycle only one unit can drive the bus, the others must disconnect any drivers during that cycle. Tri-state buffers are used to enable this. A tri-state buffer is a non-inverting driver that can adopt one of three output states:

- Logic-0 if its input is 0 and the enable is on; or
- Logic-1 if its input is 1 and the enable is on; or
- A high impedance, or disconnected state, if the enable is off, irrespective of the input logic value. In this state the driver is disconnected from the bus by opening the joining transistor switches. In Verilog, the high impedance state is represented by the symbol ‘z’.

The following Verilog fragment describes three units A, B and C, that wish to drive a common bus:

```verilog
tri [7:0] bus;
wire [7:0] Aout, Bout, Cout;
wire EnableA, EnableB, EnableC;
assign bus = EnableA ? Aout : 8'hzz;
assign bus = EnableB ? Bout : 8'hzz;
assign bus = EnableC ? Cout : 8'hzz;
```

Note that it is important that only one of EnableA, EnableB, or EnableC be active during any clock cycle. Sometimes, a specialized bus arbiter must be designed to make sure that each unit gets its proper share of the bus.

In general, buses should only be used when the savings in wiring are significant since they are more difficult to verify and debug that the equivalent set of point to point wirings.

**A.2 Sequential Logic Structures**

**Flip-flops**

Below is an example containing different types of flip-flops. Read the included comments for a description.

**Register Arrays**

Arrays of registers are often used for various functions, including the following:

- **Register Files.** A group of registers collected together, any one of which can be randomly accessed. Commonly used in computers to store the user-accessible registers.
- **Buffers.** Arrays of registers used to temporary store information. A common buffer is a First-In-First-Out (FIFO) buffer. In a FIFO, arrivals are stored in sequence and then retrieved so that their order is preserved.

Some examples follow:
The following is a 32-entry 3-port register file. This register file has two ports – i.e. one read and one write can take place every clock cycle. Note the two-dimensional specification of the storage registers themselves. Here, thirty two 8-bit registers are specified. Two examples are given. The first one has an implicit register address decoding scheme. It is implicit in the sense that the following line:

\[
\text{ReadData} = \text{RF}[\text{address}];
\]

implicitly builds the address decoder. Because not much structure is specified for the decoder, the resulting synthesized logic is larger and slower than it could be.

A better design is to specify details of the address decoder explicitly. This is done in the following complete example.

```verilog
module regfile(clock, reset, writeEnable, dest, source, dataIn, dataOut);

parameter WIDTH = 16;
parameter DEPTH = 32;
parameter ADDRESSWIDTH = 5;

integer i, j;
input clock, reset, writeEnable;
input [ADDRESSWIDTH-1 : 0] dest;
input [ADDRESSWIDTH-1 : 0] source;
input [WIDTH-1 : 0] dataIn;
output [WIDTH-1 : 0] dataOut;

reg [WIDTH-1 : 0] dataOut; // registered output
reg [WIDTH-1 : 0] rf [DEPTH-1 : 0];
wire [DEPTH-1 : 0] writeEnableDecoded;
assign writeEnableDecoded = (writeEnable << dest);

// flip-flop for data-out
always@(posedge clock)
begin
  if(!reset) dataOut <= 0;
  else dataOut <= rf[source];
end

// memory array
always@(posedge clock)
begin
  if(!reset)
  begin
    for(i = 0; i<DEPTH; i=i+1)
      rf[i] <= 0;
  end
  else
  begin
    for (j=0; j<DEPTH; j=j+1)
      ...
  end
end
```

```
if (writeEnableDecoded[j]) rf[j] <= dataIn;
end
end //always
endmodule

Note the use of the writeEnableDecoded signals to explicitly build a read address decoder. A write address decoder could be similarly built. Note also that by only changing the parameters, WIDTH, DEPTH, and ADDRESSWIDTH, different size memories can be quickly built. This is an example of design reuse.

Parameterized FIFO

This example illustrates a design that can be reused in many different situations simply by resetting two variables, the bit-width of the registers and the number of registers available in the buffer.

`include "regfile.v"

module fifo (clock, reset, inData, new_data, out_data, outData, full);
input clock;
input reset;
input [WIDTH-1 : 0] inData;
input new_data;
inout out_data;
output [WIDTH-1 : 0] outData;
output full;

parameter WIDTH = 16;
parameter DEPTH = 16;
parameter ADDRESSWIDTH = 5;

integer k; //index for "for" loops

reg [WIDTH-1 : 0] outData;
output full;

reg full; // registered output
wire fullD; // input to "full" flip-flop

reg [ADDRESSWIDTH-1 : 0] rear; // points to rear of list
reg [ADDRESSWIDTH-1 : 0] front; // points to front of list

// flip-flops to hold value of "rear"
// also increments the value of "rear" when "new_data" is high,
// checking
// the value of "rear" in lieu of a mod divide
always@(posedge clock)
begin
if (!reset) rear <= 0;
else if (new_data)
begin
if (rear == DEPTH) rear <= 0;
else rear <= rear+1;
end
end

// flip-flops to hold value of "front"
Linear Feedback Shift Register

In chapter X a simple feedback register was used as an example. That is but one case of a Linear Feedback Shift Register (LFSR). A LFSR is often used to generate a sequence of pseudo-random numbers (the sequence is only pseudo-random because it eventually repeats itself). A seed is fed into the design to provide the first number in the sequence. This example is parameterized in terms of the details of the feedback chain.

```verilog
module LFSR_TASK (clock, Reset, seed1, seed2, random1, random2);

input clock,
input [7:0] seed1,
output [7:0] random1, random2;

reg [7:0] random1, random2;

parameter [7:0] Chain1 = 8'b10001110;
parameter [7:0] Chain2 = 8'b10101110;

task LFSR_TAPS8_TASK;
    input [7:0] A;
    input [7:0] Chain;
    output [7:0] Next_LFSR_Reg;

    integer i;
    reg XorNor;

    reg [7:0] Next_LFSR_Reg;
```
begin
    XorNor = A[7] ^ ~| A[6:0];
    for (i=1; i<=7; i=i+1)
        if (Chain[i-1] == 1)
            Next_LFSR_Reg[i] = A[i-1] ^ XorNor;
        else
            Next_LFSR_Reg[i] = A[i-1];
    Next_LFSR_Reg[0] = XorNor;
end

task LFSR_TAP8_TASK
    begin
        // Implement LFSR_TAP8_TASK logic here
    endtask /* LFSR_TAP8_TASK */

/* Build 2 LFSRs using the LFSR_TAP8_TASK */

always @(posedge clock or negedge Reset)
    if (!Reset)
        random1 = seed1;
    else
        LFSR_TASK (random1, Chain1, random1);
always @(posedge clock or negedge Reset)
    if (!Reset)
        random2 = seed2;
    else
        LFSR_TASK (random2, Chain2, random2);
endmodule

Note the following in this example:
- Use of the task to build two LFSRs with different feedback chains
- Use of a for loop to iterate through the bits in the LFSR

**Bus Arbiter**

As described in the section above discussing tri-state buffers, a control unit is necessary to decide which
driver is going to be active during any particular bus cycle. Such a unit is referred to as a bus arbiter. The
following Verilog module describes a parameterized bus arbiter that can implement either of two common
schemes:
- **Round Robin Arbitration.** The driving units take turns owning the bus. If the unit does not have an
  active request in when its turn comes around, it misses its slot and the next requesting unit is enabled to
  drive the bus.
- **Priority Arbitration.** Different driving units are assigned priorities. During any cycle, the requesting
  unit with the highest priority is granted permission to drive the bus.

module arbiter(clock, reset, roundORpriority, request, priority, grant);
    integer i, j, k, p, q, r, s, t, u, v; // index for "for" loops
    // ---------------------------------------------------------------
    // parameters

//----------------------------------------------------------------------------------
--
parameter NUMUNITS = 8;
parameter ADDRESSWIDTH = 3; //number of bits needed to address NUMUNITS

//----------------------------------------------------------------------------------
--
// input and output declarations
//----------------------------------------------------------------------------------
--
input clock;
input reset;
input roundORpriority;
input [NUMUNITS-1 : 0] request;
input [ADDRESSWIDTH*NUMUNITS-1 : 0] priority;

output [NUMUNITS-1 : 0] grant;

//hack for 2-D input
reg [ADDRESSWIDTH-1 : 0] prio [NUMUNITS-1 : 0];
reg [ADDRESSWIDTH-1 : 0] tmp_prio;
always@ (priority)
begin
    for (i=0; i<NUMUNITS; i=i+1)
        begin
            for (j=0; j<ADDRESSWIDTH; j=j+1)
                tmp_prio[j] = priority[i*ADDRESSWIDTH + j];
            prio[i] = tmp_prio;
        end
end

reg [NUMUNITS-1 : 0] grant;   //registered output
reg [NUMUNITS-1 : 0] grantD;  //input to "grant" flip-flop

reg [ADDRESSWIDTH-1 : 0] next;  //index of next unit in round-robin
reg [ADDRESSWIDTH-1 : 0] nextNext;   //input to "next" flip-flop

reg [ADDRESSWIDTH-1 : 0] scan [NUMUNITS-1 : 0];
//stores info on the order in which to scan units for round-robin

reg [NUMUNITS-2 : 0] found;
//in round-robin search, stores info on where assignment is made

reg [ADDRESSWIDTH-1 : 0] selectPrio[NUMUNITS-1 : 0];
//holds the priorities of only those units requesting the bus

reg [ADDRESSWIDTH-1 : 0] min;
//holds the minimum priority of all units currently requesting the bus

reg [NUMUNITS-1 : 0] minPrio;
//units that have the minimum priority

wire [NUMUNITS-1 : 0] prioRequest;
//request signals for only those units with minimum priority
reg [NUMUNITS-1 : 0] finalRequest;
// requests actually examined depending on "roundORpriority"

// flip-flop for "grant" signals
always @(posedge clock)
begin
  if(!reset) grant <= 0;
  else grant <= grantD;
end

// flip-flop for "next" register
always @(posedge clock)
begin
  if(!reset) next <= 0;
  else next <= nextNext;
end

// selects the priorities of units sending requests
begin
  for(k=0; k<NUMUNITS; k=k+1)
    selectPrio[k] = request[k] ? prio[k] : NUMUNITS-1;
end

// selects priority or round robin operation
always @(prioRequest or request or roundORpriority)
begin
  for(r=0; r<NUMUNITS; r=r+1)
    finalRequest[r] = roundORpriority ? prioRequest[r] : request[r];
end

// this logic finds the minimum priority out of all units sending a request
begin
  min = selectPrio[0];
  for (p=1; p<NUMUNITS; p=p+1)
    if (selectPrio[p] < min) min = selectPrio[p];
end

// this logic decides if the units have minimum priority
begin
  for(q=0; q<NUMUNITS; q=q+1)
    minPrio[q] = (prio[q]==min) ? 1:0;
end

// produces request signals for units that have minimum priority
assign prioRequest = minPrio & request;

// produces the "scan" array
always @(next)
begin
    for(s=0; s<NUMUNITS; s=s+1)
        scan[s] = (next+s < NUMUNITS) ? next +s : next+s-NUMUNITS;
end

//produces the "found" array
    scan[3] or scan[2] or scan[1] or scan[0])
begin
    found[0] = finalRequest[scan[0]];
    for(t=1; t<NUMUNITS-1; t=t+1)
        found[t] = found[t-1] || finalRequest[scan[t]];
end

//produces inputs to "grant" flip-flops
always @(finalRequest or found or scan[7] or scan[6] or scan[5] or
begin
    grantD[scan[0]] = finalRequest[scan[0]];
    for(u=1; u<NUMUNITS; u=u+1)
        grantD[scan[u]] = finalRequest[scan[u]] && ~found[u-1];
end

always @(grantD)
begin
    nextNext = 0;
    for(v=0; v<NUMUNITS-1; v=v+1)
        if(grantD[v]) nextNext = v+1;
end
endmodule //arbiter