Solutions to Problems

Chapter 2

(2.1) The declarations are good. Verilog interprets `reg vert [7:0]` as a 7x1 memory, ie: like a register in the vertical direction. For the invocations, the left-hand side identifiers of course need to be pre-declared as `reg`. Otherwise, the indexes are all within the declared ranges: the index `n` is fixed at 3 by the `parameter` declaration, and even the variable `ptr` is OK since it cannot evaluate outside the declared range. This kind of construction is used to good effect in the cache specification of section 10.5. However this formulation is not appropriate for use inside the `for` construct.

(2.2) Note the ordering in the port lists in this case is not critical (compare text page 14)

```
module add_4_r (A, B, C_in, SUM, C_out);
  input [3:0] A, B;
  input C_in;
  output [3:0] SUM;
  output C_out;
  wire [3:0] A, B, SUM;
  fulladder FA3(.a(A[3]), .b(B[3]), c_in(C2), sum(SUM[3]), c_out(C_out));
  fulladder FA2(.a(A[2]), .b(B[2]), c_in(C1), sum(SUM[2]), c_out(C2));
  fulladder FA1(.a(A[1]), .b(B[1]), c_in(C0), sum(SUM[1]), c_out(C1));
  fulladder FA0(.a(A[0]), .b(B[0]), c_in(C_in), sum(SUM[0]), c_out(C0));
endmodule
```

(2.3)

```
module add_8_r (AA, BB, C_in, SS, C_out);
  input [7:0] AA, BB;
  input C_in;
  output [7:0] SS;
  output C_out;
  wire [7:0] AA, BB, SS;
  add_4_r four1(AA[7:4], BB[7:4], C, SS[7:4], C_out);
  add_4_r four0(AA[3:0], BB[3:0], C_in, SS[3:0], C);
endmodule
```

(2.4) Assume the propagation times through the modules are $T_{p_a}$, $T_{p_b}$.
To meet the hold time at the input of module B we need $T_{p_a} > T_{h_b}$.
To meet the set up time at the input to module B, $T_{cl} - T_{p_a} > T_{s_b}$.
So the minimum clock period, $T_{cl} > T_{p_a} + T_{s_b} > T_{h_b}$.
Likewise with A and B reversed. So the minimum clock period is
the sum of the hold and set up times for whichever module is slower.

(2.5) operation: `wire` (since input only), `data_in`: ditto
`data_out`, `status return`: either `reg` or `wire` (since output only)

(2.6) (a) op1[3]  (b) ADD4.A[3]  (c) ADD4.FA3.a
Chapter 3

(3.1)

Let incrementer inputs = w x y z, and outputs = W X Y Z, with w most significant. Let x’ indicate the complement of x, let . indicate the AND, and let + indicate OR. Then

\[ Z = z' \]
\[ Y = y.z' + y'.z \]
\[ X = x.y + x.z' + x'.y.z \]
\[ W = w.y' + w.z' + w.x' + w'.x.y.z \]
\[ \text{Cout} = w.x.y.z \]

Area = 12, worst delay = 5. (Compare to ripple formulation in 3.2 above.)

(3.2)

Area of one stage = 5
Area of four stages = 20
Worst delay in 4 stages = 5

(3.3)

Let incrementer inputs = w x y z, and outputs = W X Y Z, with w most significant. Let x’ indicate the complement of x, let . indicate the AND, and let + indicate OR. Then

\[ Z = z' \]
\[ Y = y.z' + y'.z \]
\[ X = x.y + x.z' + x'.y.z \]
\[ W = w.y' + w.z' + w.x' + w'.x.y.z \]
\[ \text{Cout} = w.x.y.z \]

Area = 12, worst delay = 5. (Compare to ripple formulation in 3.2 above.)

(3.4)

module mux2(slct, nsct, a, b, o1);
input a, b, slct, nsct;
output o1;
nor(c1, a, slct);
nor(c2, b, nsct);
mux2(a1, a2, slct0, nsct0, out);
endmodule

module mux4(slct1, slct0, n3, n2, n1, n0, out);
// inputs and output active low
input slct1, slct0, n1, n2, n3, n4;
output out;
not(n3slct1, slct1); not(n3slct0, slct0);
mux2(n1, n2, slct1, nsct1, a1);
mux2(n3, n4, slct1 nsct1, a2);
mux2(a1, a2, slct0, nsct0, out);
endmodule
The circuit above is usually derived from
the truth table below left via classical methods
for the minimization of a multi-output
function
although in this simple case it can be eyeballed
d7: 1 1 1 and then transformed to a net
d6: 1 1 0 of NAND/NOR gates
d5: 1 0 1 using the equivalence shown
d4: 1 0 0 at upper right.
d3: 0 1 1 In this case, the synthesizer
d2: 0 1 0 could not do better.
d1: 0 0 1 (That’s what everybody thinks)
d0: 0 0 0

module encode (onehot, code)
parameter r=3, n = 1<<r;
input [n-1:0] onehot;
output [r-1:0] code;
integer i;
always @(onehot)
begin
code =0;
for(i=n-1; i<0; i=i-1)
if(onehot[i]) code = i;
end
endmodule

module par2(a, b, o1);
input a, b;
output o1;
and(a, b, c1);
and(a c1, c2);
and(b, c1, c3);
and(c2, c3, o1);
endmodule

module par4(in3,in2,in1,in0,out);
input in3, in2, in1, in0;
output out;
par2(in3, in2, a1);
par2(in1, in0, a2);
par2(a1, a2, out);
endmodule
(3.7) The required property is associativity. The synthesizer would select a tree as being optimal in each case.

(3.8) The required property is associativity. The synthesizer would select a tree as being optimal in each case.

(3.9) See 3.15. Sorry about that.

(3.10) Yes, since this is a ripple circuit, the carry-with-less-delay principle should be applicable to it, although in this case the best that can be done is to transform the lower OR chain into a tree of depth $\log_2 n$.

wire [1:0] out;

Equations of the carry-lookahead-unit (CLA) needed are

\[
\begin{align*}
    c[0] &= r[3] \lor r[2] \lor r[1] \lor Cin \\
    G &= r[3] \lor r[2] \lor r[1] \lor r[0]
\end{align*}
\]

and a possible circuit realization is shown on the right:

The structure for a carry-with-less-delay organization using 5 such CLA modules is as follows:

A possible test-module follows:

```
module test_prioritizer;
    reg C_in;
    reg [15:0] R;
    wire [15:0] grant;
    wire GG;
    prioritizer prio(R, grant);

initial begin
    $display("C_in R[15:0] grant[15:0] GG");
    $display("C_in = 0");
    for ( R = 16'haaaa; R!=0; R=R>>1) #1;
    C_in = 1;
    for ( R = 16'hffff; R!=0; R=R>>1) #1;
    end
endmodule
```
One possible solution is as follows:

As for logical depth, assuming only 2-input gates available, and a mantissa of 16 bits, then the prioritizer would need 8 delays (problem 3.15), the decoder 3 delays (problem 3.5), and the shifter 8 x 2 delays (figure 3.6) making 27 in all.
Chapter 4

(4.1) compare with:
the specification #1 of mux 4 on page 19
the specification of full_par on page 54

module mux4 (slct, in3, in2, in1, in0, out)
input [2:0] S;
input in3, in2, in1, in;
output out;
not(nslct1,slct[1]), not(nslct0,slct[0]);
nand ( a3, in3, slct[1], slct[0] );
nand ( a3, in2, slct[1], nslct0 );
nand ( a1, in1, nslct1, s;ct[0] );
nand ( ao, in0, nslct1, nslct0 );
nand ( out, a3, a2, a1, a0 );
endmodule

module test_mux4;
reg [2:0] S;
reg [3:0] in;
wire out;
integer i;
mux4 M4 (S, in3,in2, in1, in0, out);
initial begin
$display("S, in,  out");
$monitor("%b  %h  %b", S, in, out);
for (i=3; i>=0; i=i-1)
begin
S <= i;
in[i]<=0;  // toggle ith input
in[i]<=1;
end
end
endmodule

Either the specifications #4 and #5 of section 5.4
can be tested with a test module similar to the following:
(could also write a spec based on figure 3.9(b) )

module test_comparator;
reg [3:0] A,B;
wire Cgt, Clt, Cne;
comparator #(4) comparator(A, B, Cgt, Clt, Cne);
initial begin
$display("A    B   Cgt Clt Cne ");
$monitor("%h %h %b %b %b", A, B, Cgt, Clt, Cne);
#1 A=-1; B=-1;
#1 A= 1; B= 1;
#1 A=-1; B= 1;
#1 A= 1; B= 1;
#1 A= 1; B= 0;
#1 A= 7; B=15;
#1 A=15; B =7;
end
endmodule

(4.2) module shifter(amt, Win, Wout);
// based on operation in table 3.1
parameter a = 3;
parameter w = (1<<a);
input [a-1:0] amt;
input [w-1:0] Win;
output [w-1:0] Wout;
reg [m-1:0] Dout;
always @(Din,amt)
Dout<= (Din<<amt);
endmodule

module shifter(amt, Win, Wout);
// based on structure of figure 3.6
parameter a = 3;
parameter m = (1<<a);
input [a-1:0] addr;
output [m-1:0] Dout;
mux2 m0(amt[0], Din, Din>>1, c0);
mux2 m1(amt[1], c0, c0>>2, c1);
mux2 m2(amt[2], c1, c1>>4, Dout);
endmodule

module test_shifter;
parameter a = 3;
parameter w = (1<<a);
reg [a-1:1] A; reg [w-1:1] Din;
wire [w-1:1] Dout;
shifter #(3) SH(A, Din, Dout);
initial begin
$display("A, Din  Dout");
$monitor("%d  %b %b", A, Din, Dout);
Din<=8'h10;
for (A=0; A<w; ) #1;
end
endmodule

(4.3) module shifter(amt, Win, Wout);
// based on operation in table 3.1
parameter a = 3;
parameter w = (1<<a);
input [a-1:0] amt;
input [w-1:0] Win;
output [w-1:0] Wout;
reg [m-1:0] Dout;
always @(Din,amt)
Dout<= (Din<<amt);
endmodule

module shifter(amt, Win, Wout);
// based on structure of figure 3.6
parameter a = 3;
parameter m = (1<<a);
input [a-1:0] addr;
output [m-1:0] Dout;
mux2 m0(amt[0], Din, Din>>1, c0);
mux2 m1(amt[1], c0, c0>>2, c1);
mux2 m2(amt[2], c1, c1>>4, Dout);
endmodule

module test_shifter;
parameter a = 3;
parameter w = (1<<a);
reg [a-1:1] A; reg [w-1:1] Din;
wire [w-1:1] Dout;
shifter #(3) SH(A, Din, Dout);
initial begin
$display("A, Din  Dout");
$monitor("%d  %b %b", A, Din, Dout);
Din<=8'h10;
for (A=0; A<w; ) #1;
end
endmodule
(4.4) According to the Verilog LRM, bitvector sizes should be good at least to 65536 or 2 to the power 16, and simulation vendors are supposed to adhere to the standard. So if it breaks you should complain.

```
module prioritizer (request,grant);
  parameter n = 16;
  input [n-1:1] request;
  output [n-1:1] grant;
  reg [n-1:1] grant; reg Chr;
  integer i;
  always @(request)
    begin: pri
      grant <=0;
      for (i=n-1; i>=0; i=i-1)
        begin
          if (request[i])
            Chr<= 1; grant[i]<=1;
            disable pri; // stop looping
        end
      // if this point is reached there are no requests
     Chr<=0;
    end
endmodule
```

(4.5)

```
module nothot (in, out)
  // structural spec based on prob 3.7;
  // compare behavioral spec #6, sect 5.4
  parameter n = 3;
  input [n-1:0] in;
  output out;
  wire [n-1:0] c, n, t;
  integer k;
  for (k=0; k<n;)
    begin
      and ( c[k], in[k], n[k] );
      or ( n[k+1], in[k], n[k] );
      or ( t[k+1], c[k], t[k] );
    end
  not ( d, n[2]); nor ( out, d, t[2] );
endmodule
```

(4.6) According to the Verilog LRM, bitvector sizes should be good at least to 65536 or 2 to the power 16, and simulation vendors are supposed to adhere to the standard. So if it breaks you should complain.
Chapter 5

(5.1) Multiple *always* statements, non-blocking assignments, and parallel invocations of modules all give the effect of concurrency and are synthesizable. *Fork-join* statements would also give the effect of concurrency but are not currently synthesizable.

(5.2) The following three alternatives are all synthesizable:

<table>
<thead>
<tr>
<th>begin</th>
<th>task1 invocation</th>
<th>task2 invocation</th>
<th>for (; ;) &lt;blocking assignment&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(5.3) (a) Apply some symbolic input x to the D input. Then essentially follow through the gates of the circuit, (labelling each node with 0 1 x x' as the case may be) as the clock input changes. Eg:

```
clock 0 1 0 0 1  
D  x x x' x'  
```

(b) Truth table

<table>
<thead>
<tr>
<th>Q0</th>
<th>S</th>
<th>R</th>
<th>Qj+1</th>
<th>Qd+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Karnaugh maps

```
JK
0 0 0 1 1 1 1 0
0 0 1 0 1 0 0 0
```

Excitation equations

\[ Q_{j+1} = Q'.J v Q.K' + J.K' \]

\[ Q_{d+1} = S v Q.R' \]

Corresponding circuits:

(5.4)

```
always while (x < y)
begin
 @(posedge clk)
 x = x + 1;
end
```

(5.5) See problem 4.4.
A possible solution is:

(a) \( \text{always } @(\text{clk or reset}) \)
\[
\begin{align*}
\text{if (reset) } & \text{ FF } <= 0; \\
\text{else } & \text{ FF } <= D;
\end{align*}
\]

(b) \( \text{always } @(\text{posedge clk}) \)
\[
\begin{align*}
\text{if (reset) } & \text{ FF } <= 0; \\
\text{else } & \text{ FF } <= D;
\end{align*}
\]

(c) If a latch is used the state machine will multi-transition during a single clock. To prevent this either a master-slave or edge triggered type flip-flop is necessary.

ordering is significant only for blocking assignments

\[
\begin{align*}
A = C; & \ B = A; \ C = B; \\
C <= B; & \ B <= A; \ A <= C;
\end{align*}
\]

\[
\begin{align*}
\text{always } @(x \text{ or } y \text{ or } z) \begin{align*}
\text{begin} \\
w &= y \& z; \\
u &= w \text{ } | \text{ } x;
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{always } @(\text{posedge } x \text{ or } \text{posedge } y \text{ or } \text{posedge } z) \begin{align*}
\text{begin} \\
w &= y \& z; \\
u &= w \text{ } | \text{ } x;
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{always } @(\text{posedge } x \text{ or } \text{posedge } y \text{ or } \text{posedge } z) \begin{align*}
\text{begin} \\
w &= y \& z; \\
u &= w \text{ } | \text{ } x;
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{module } \text{prienc}(\text{request,grantcode}); \\
\text{parameter } n=2; \ m = 1<<n; \\
\text{input } [m-1:1] \text{ request;} \\
\text{output } [n-1:0] \text{ grantcode;} \\
\text{reg } [n-1:0] \text{ grantcode;} \\
\text{always } @(\text{request}) \begin{align*}
\text{begin} \\
\text{sweep} \\
\text{grantcode } <= 0; \\
\text{for } (k=n, k>=0; k=k-1) \\
\text{if } (\text{request}=1<<k) \\
\text{begin} \\
\text{grantcode } <= k+1; \\
\text{disable} \text{ sweep;} \\
\text{end}
\text{end}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{ordering is significant only for blocking assignments}
\end{align*}
\]

\[
\begin{align*}
\text{ditto with D type flip-flops} \\
\text{order matters} \\
\text{assume } y, z, x \text{ declared as regs}
\end{align*}
\]

(5.9) A possible solution is:
(5.10):

always @(posedge clk)
casez (op)
2'b1? : // assignment block1
2'b?1 : // assignment block2
endcase

The case is not full - not parallel
The synthesized circuit will execute block1 in response to op=11, and in response to op=00 the output will retain its previous value as a result of a FF being synthesized.

(5.11)

always @(posedge clk)
if (push) begin
  ptr = ptr +1;
  Dout = RAM[ptr];
end

Would be synthesized with an edge triggered FF if available in which case Dout will get assigned with the previous value of the pointer. If this is not what the user intends, a posedge clk should be inserted between the assignments.

(5.12)

output for H =3'b110:    warm => logical 1    notsohot => A     tepid => previous output
The clock edge refers to the positive edge.

always@(posedge clock)
case ({J,K})
2'b00 : Q <= Q;
2'b01 : Q <= 0
2'b10 : Q <= 1;
2'b11 : Q = ~Q;
endcase
If the specification also called for a write operation, then tablelookup1 would need an additional demultiplexer.

For large table sizes the second version would start to become more economical.
The fast 64 bit up/down partitioned up/down counter in the style of Ercegovac and Tenca [Tenca97], [Stan98] was done by students as a term project at Stony Brook using twisted tail ring counter components as well as register-incrementer components. It is too long to reproduce here.

(5.20) Same as 5.5, Sorry about that!

(5.21)