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**s. 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)

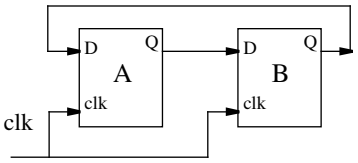
```
module add_4_r (A, B, C_in, SUM, C_out);
   input [3:0] A,B;
   input C_in;
   output [3:0] SUM;
   output C_out;
   wire [3:0] A, B, SUM;
    fulladder FA3(.a(A[3]), .b(B[3]), c_in(C2),  sum(SUM[3]), c_out(C_out));
    fulladder FA2(.a(A[2]), .b(B[2]), c_in(C1),  sum(SUM[2]), c_out(C2));
    fulladder FA1(.a(A[1]), .b(B[1]), c_in(C0),  sum(SUM[1]), c_out(C1));
    fulladder FA0(.a(A[0]), .bB([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 Tp_a, Tp_b.
To meet the hold time at the input of module B we need Tp_a > Tho_b.
To meet the set up time at the input to module B, Tcl - Tp_a >Tsu_b.
So the minimum clock period, Tcl > Tp_a + Tsu_b > Tho_b + Tsu_b.
Likewise with A and B reversed. So the minumum clock period is
the sum of the hold and set up times for whichever module is slower.

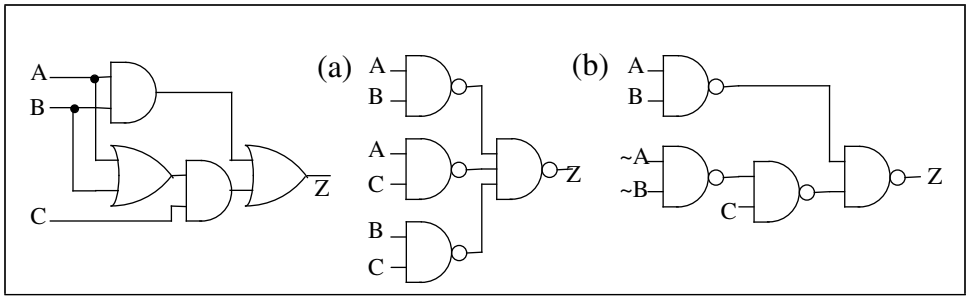


(2.5) operation : **wire** (since input only), data_in : ditto
data_out, status return: either **reg** or **wire** (since output only

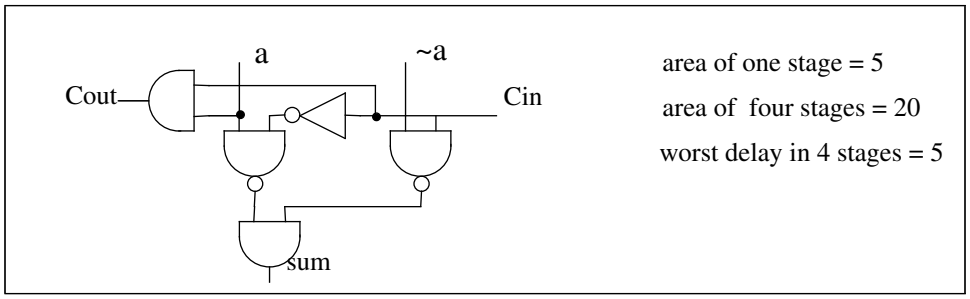
(2.6) (a) op1[3] (b) ADD4.A[3] (c) ADD4.FA3.a

Chapter 3

(3.1)



(3.2)



(3.3)

Let incrementer inputs = w x y z, and outputs = W X Y Z, with w most significant.

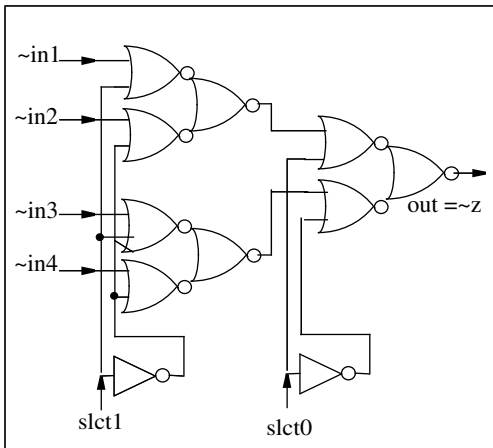
Let x' indicate the complement of x, let . indicate the AND, and let + indicate OR.

Then Z = z' Y = y.z' + y'.z X= x.y + x.z' + x'.y.z = x.(y.z)' + x'.(y.z)

W = w.y' + w.z' + w.x' + w'.x.y .z = w.(x.y.z)' + w'.(x.y.x) Cout = w.x.y.z

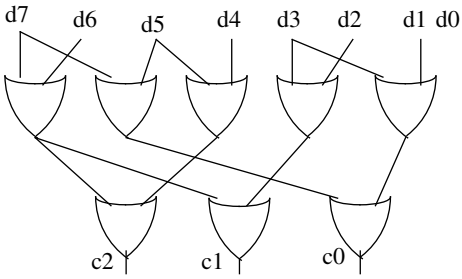
Area = 12, worst delay = 5. (Compare to ripple formulation in 3.2 above.)

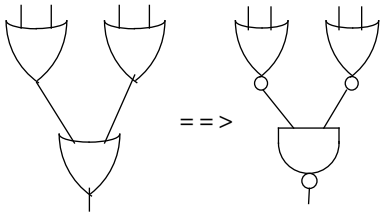
(3.4)




```
module mux2(slct, nslct, a, b, o1);
  input a, b, slct, nslct;
  output o1;
    nor(c1, a, slct); nor(c2, b, nslct);
    nor(o1, c1, c2);
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(nslct1,slct1); not(nslct0,slct0);
    mux2(n1, n2, slct1, nslct1, a1);
    mux2(n3, n4, slct1 nslct1, a2);
    mux2(a1, a2, slct0, nslct0, out);
 endmodule
```

(3.5)





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 and then transformed to a net of NAND/NOR gates using the equivalence shown at upper right.
In this case, the synthesizer could not do better.
(That's what everybody thinks)

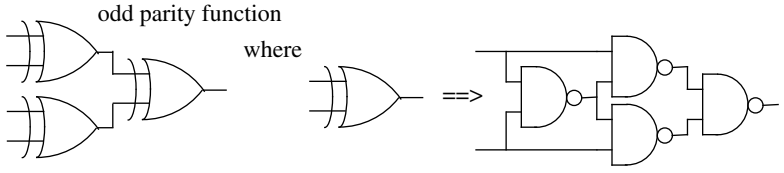
d7:	1 1 1
d6:	1 1 0
d5:	1 0 1
d4:	1 0 0
d3:	0 1 1
d2:	0 1 0
d1:	0 0 1
d0:	0 0 0

```
module encode (onehot, code)
  parameter r=3, n = 1<<r;
  input [n-1:0] onehot;
  output [r-1:0];
  reg [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
```

(3.6)

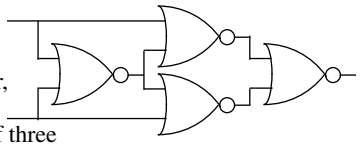
odd parity function

where



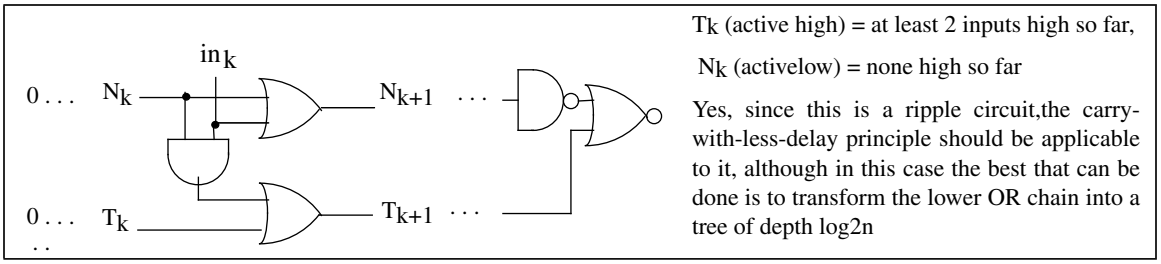
so whole 4 input odd parity function uses 12 nand gates (solution to part (ii))

We could reduce the delay from 6 to 5 by also computing the complements of the 2 input exor functions in the first layer, using the circuit on the right, and then combining the four outputs using a tree of three AND gates (but this increases the gate count to 19)



```
module par2(a, b, o1);
  input a, b;
  output o1;
    nand(a, b, c1);
    nand(a c1, c2); nand(b, c1, c3);
    nand(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)

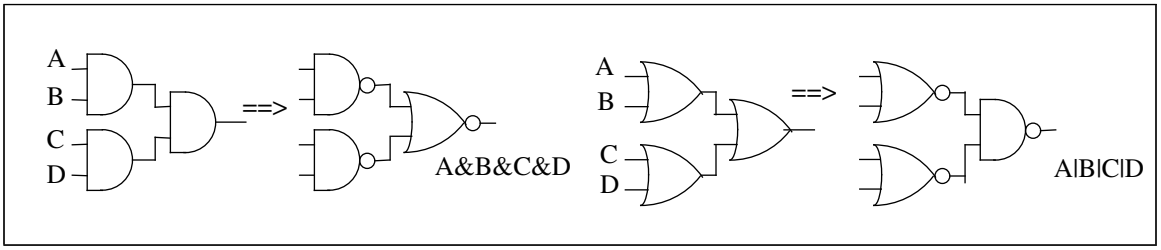


T_k (active high) = at least 2 inputs high so far,

N_k (activelow) = none high so far

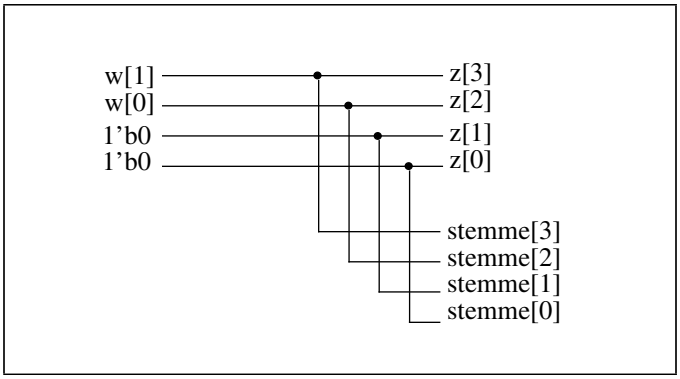
Yes, since this is a ripple circuit,the carry-with-less-delay principle should be applicable to it, although in this case the best that can be done is to transform the lower OR chain into a tree of depth log2n

(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.

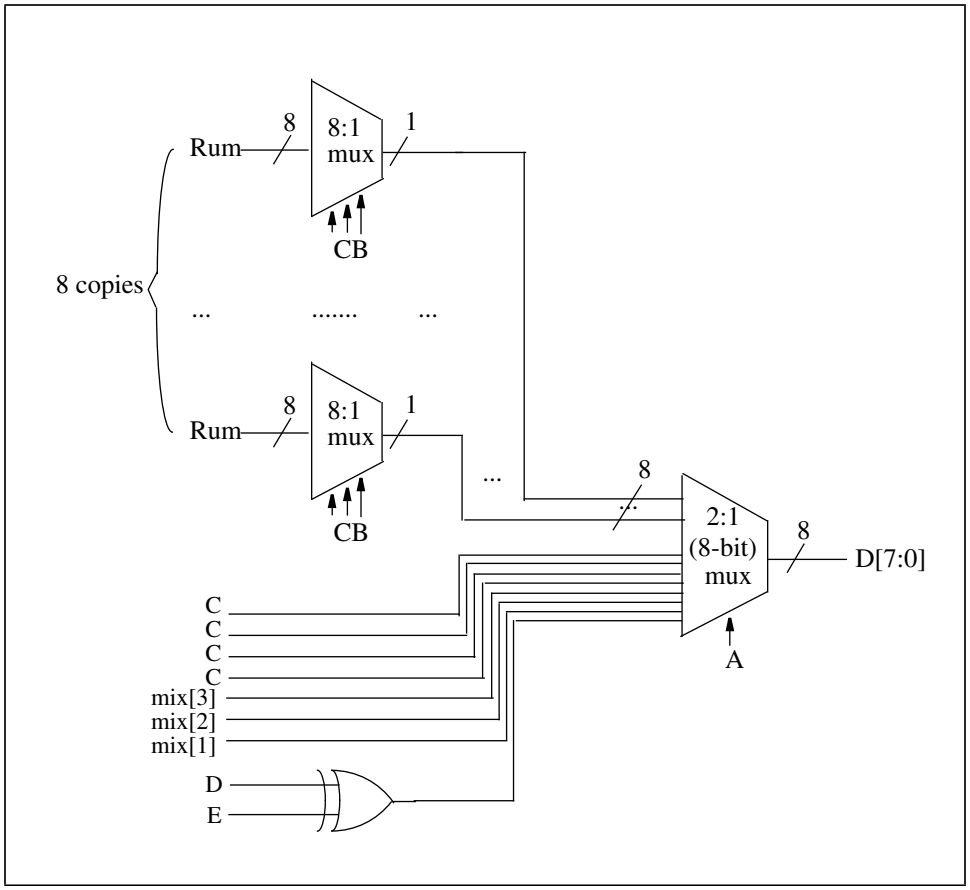
(3.10)



(3.11)

```
assign feedback = &{A[3] | B[3],| A[3] | B[2] , A[1] | B[1], A[0] | B[0]};
```

(3.12)



(3.13)

```
wire [1:0] out;
assign out = S[0] ? S[2:1] : {&{A[3:0],B[1:0]}, ^{B[3:2],C[3:0]}};
```

(3.14)

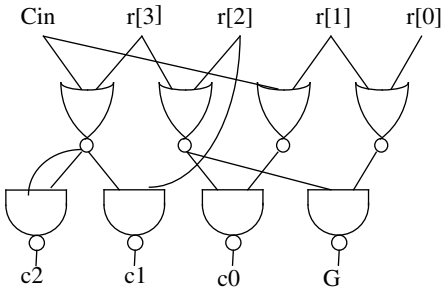
```
assign flag = data[3]&data[2] | data[2]&data[1] | data[1]&data[0];
```

(3.15) :

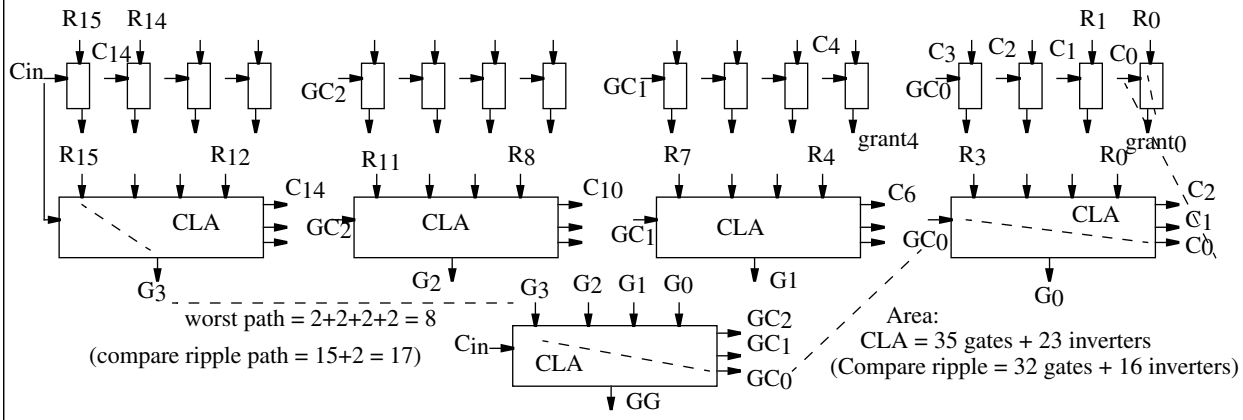
Equations of the carry-lookahead-unit (CLA) needed are

$$c[2] = r[3] \vee Cin$$
$$c[1] = r\{3] \vee r[2] \vee Cin$$
$$c[0] = r[3] \vee r[2] \vee r[1] \vee Cin$$
$$G = r[3] \vee r[2] \vee r[1] \vee r[0]$$

and a possible circuit realization is shown on the right:



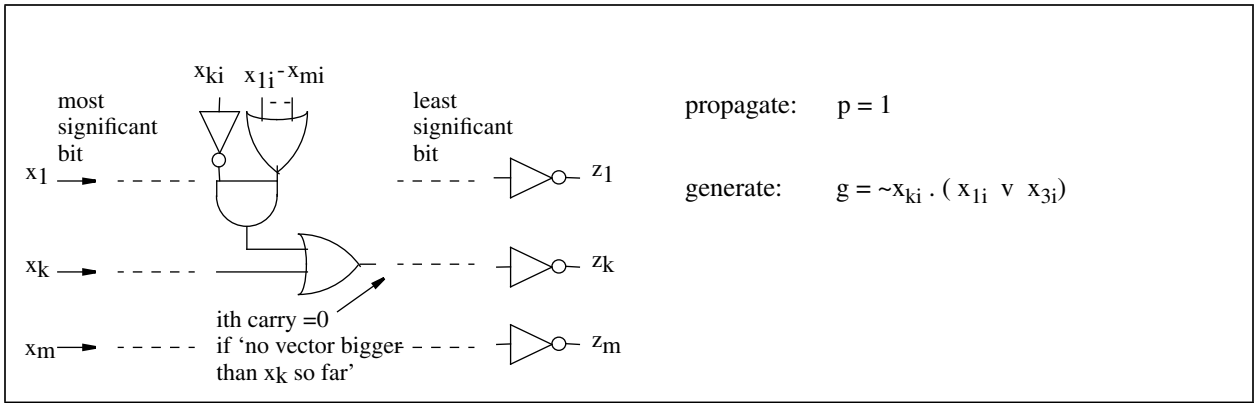
The structure for a carry-with-less-delay organization using 5 such CLA modules is as follows:



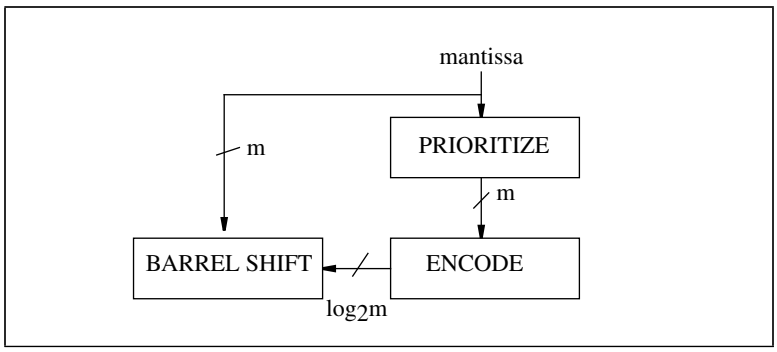
A possible test-module follows:

```
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");
   $monitor(" %b          %b              %b            %b", C_in, R, grant, GG);
   $display("C_in = 0");
   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
```

(3.16)



(3.17) 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, in[3],in[2], in[1], in[0], 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;
      #1 in[i]<=0;  // toggle ith input
      #1 in[i]<=1;
    end
 end
endmodule
```

(4.2)

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
```

(4.3)

```
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<=8h'c0;
    for (A=0; A<w; ) #1;
   end endmodule
```

(4.4)

```
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])
          begin Chr<= 1; grant[i]<=1;
             disable pri; // stop looping
          end
       end
     // if this point is reached there are no requests
     Chr<=0;
  end
endmodule
```

```
module test_prioritizer;
parameter n = 16, m=7;
reg [n-1:1] rq; // request vector
wire [n-1:1] grant;
reg testreg;
 prioritizer #(16)  PR(rq,grant);
initial
 begin
  $display("request          grant");
  $monitor("%b  %b", rq,grant);
  for (rq=-1; rq!=0; rq=rq>>1) #1;
  for (rq=423643; rq!=0; rq = rq>>1) #1;
 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
```

```
module test_nothot;
parameter n = 3;
reg [n-1:0] IN; // input vector
wire OUT;
integer i;
 nothot #(n)  NH(IN, OUT);
initial
 begin
  $display("in          out");
  $monitor("%b  %b", IN, OUT);
    for (i=0; i<(1<<n); i=i+1)
    #1 IN <= i;
 end
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 arre not currently synthesizable.

(5.2) The following three alternatives are all synthesizable:

```
begin                          task1 invocation
 <blocking assignments)>       task2 invocation        for ( ; ; ) <blocking assignment>
end
```

(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:

clk 0 1 0 0 1
D x x x x' x'

(b)

truth table

inputs			outputs	
Q^n	S	R	Q_{jk}^{n+1}	Q_{sd}^{n+1}
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

Karnaugh maps

JK

Q \	00	10	11	01
0	0	1	1	0
1	1	1	0	0

SR

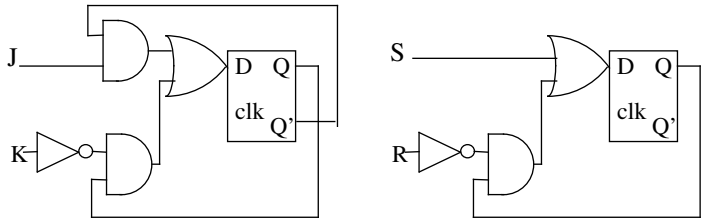
Q \	00	10	11	01
0	0	1	1	0
1	1	1	1	0

Excitation equations

$Q_{jk}^{n+1} = Q'.J \vee Q.K' + J.K'$

$Q_{sd}^{n+1} = S \vee Q.R'$

Corresponding circuits:



(5.4)

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

(5.5) See problem 4.4.

(5.6)

(a)

```
always @(clk or reset)
 if (reset) FF <=0;
 else FF <= D;
```

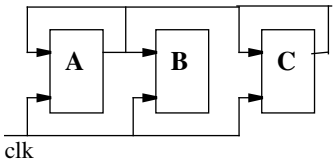
(b)

```
always @(posedge clk)
 if (reset) FF <=0;
 else FF <= D;
```

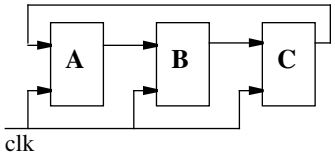
(c) If a latch is used the state machine will multi-transition during a single clock. To prevent this either a master-slave or edge triggered type flip-flop is necessary.

(5.7)

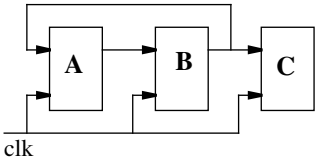
A=C; B=A; C=B;



C<=B; B<=A; A<=C;



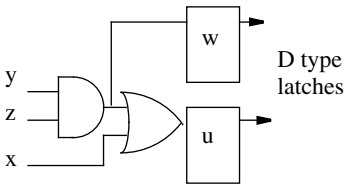
C=B; B=A; A=C;



ordering is significant only for blocking assignments

(5.8)

```
always @ (x or y or z)
  begin
    w = y & z;
    u  = w | x;
  end
```



D type latches

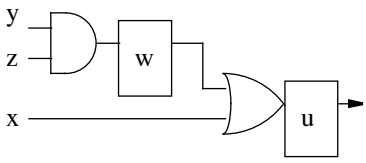
```
always @ (posedge x or posedge y or
posedge z)
  begin
    w = y & z;
    u  = w | x;
  end
```

ditto with D type flip-flops

order matters

assume y,z,x declared as regs

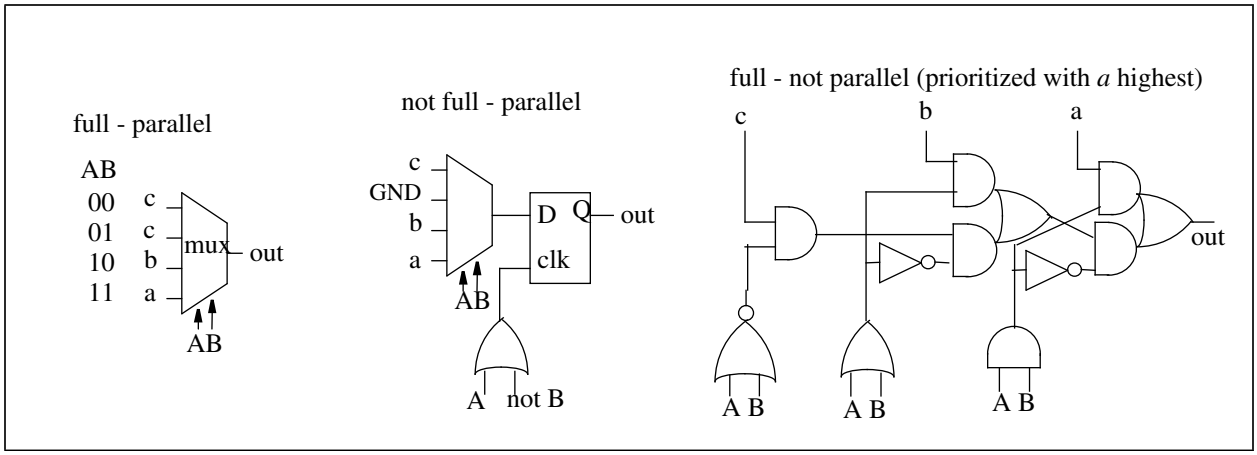
```
always @ (posedge x or posedge y or
posedge z)
  begin
    w <= y & z;
    u  <= w | x;
  end
```



(5.9) A possible solution is:

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

(5.10) :



(5.11)

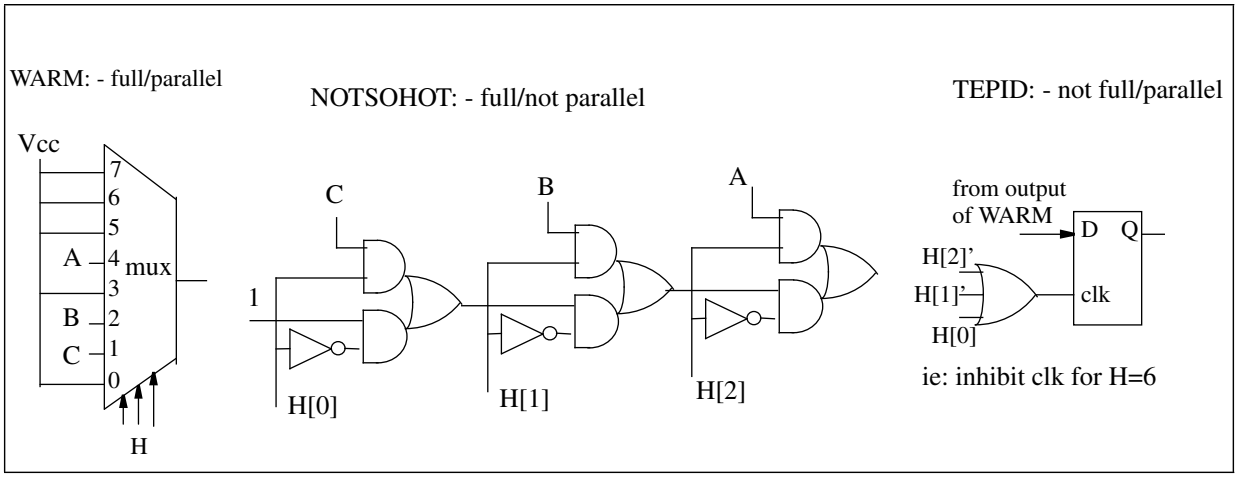
```
always @(posedge clk)
  casez (op)
    2'b1? : // assignment block1
    2'b?1 : // assignment block2
  endcase
```

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.

```
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

(5.13)

```
reg [1:0] A;
reg B, C, Next1, Next2;

always@(A or B or C)
 begin
  Next1 = 0;
  Next2 = 0;
  casex({A,B,C})
   4'b11xx : begin Next1=1;  Next2=1; end
   4'b1011 : Next2 = 1;
   4'b0100 : ;
   4'b001x : begin Next1=1;  Next2=1; end
   default : Next1 = 1;
  endcase
 end
```

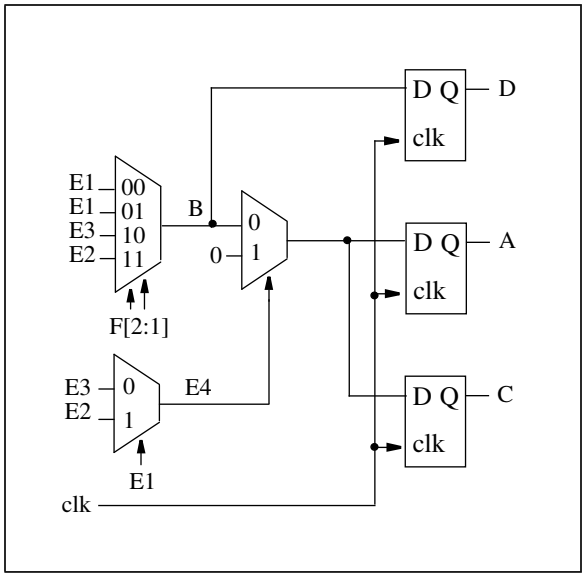
(5.14)

```
reg [1:0] A;
reg B, C, nextNext1, nextNext2, Next1, Next2;

always@(posedge clock)
 begin
  Next1 <= nextNext1;
  Next2 <= nextNext2;
 end

always@(A or B or C)
 begin
  nextNext1 = 0;
  nextNext2 = 0;
  casex({A,B,C})
   4'b11xx : begin nextNext1=1;  nextNext2=Next2; end
   4'b1011 : nextNext2 = 1;
   4'b0100 : nextNext1 = Next1;
   4'b001x : begin nextNext1=1;  nextNext2=1; end
   default :   begin nextNext1=Next1;  nextNext2=Next2; end
  endcase
 end
```

(5.15) The clock edge refers to the positive edge.

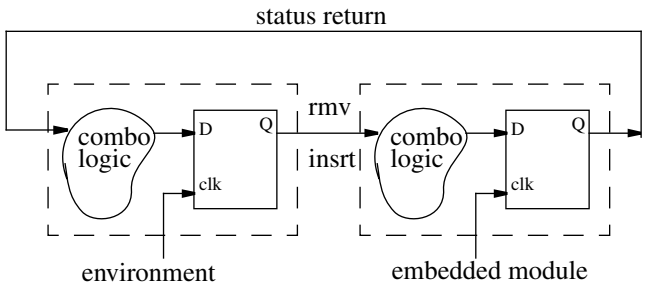
```
always@(posedge clock)
case ({J,K})
   2'b00 : Q <= Q;
   2'b01 : Q <= 0
   2'b10 : Q <= 1;
   2'b11 : Q = ~Q;
 endcase
```

(5.16)

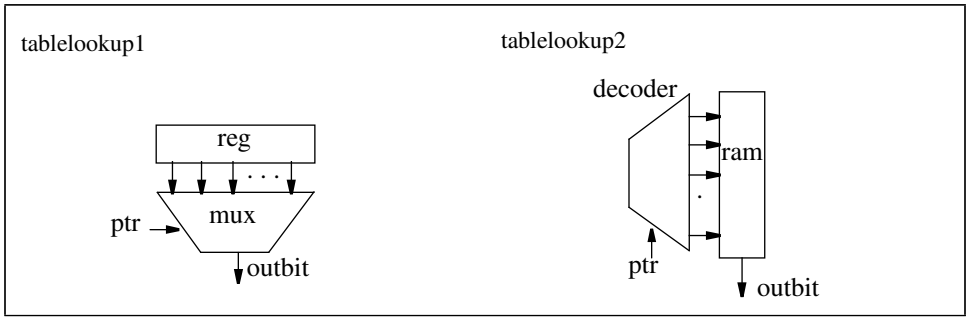


(5.17)

The problem is essentially the same as 2.4
The minumum clock period is
the sum of the hold and set up times
for whichever module is slower.



(5.18)



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

(5.19)

```
module decoder(addr, Dout);
  parameter a = 3;
  parameter m = (1<<a);
  input [a-1:0] addr;
  output [m-1:0] Dout;
  reg [m-1:0] Dout;

    always @(addr)
    Dout<= (1<<addr);

endmodule
```

```
‘include “decoder.v”
module ram1(clk, WE, addr,
 data_in, data_out);
 parameter n = 8;
 parameter a = 3;
 parameter w = 4;
 input  clk, WE;
 input  [a-1:0] addr;
 input  [w-1:0] data_in;
 output [w-1:0] data_out;
 reg    [w-1:0] data_out;
 reg    [w-1:0] ram_data [n-1:0];
 wire [n-1:0] row;
 integer i;
  decoder #(a,n) DCD(addr,row);

  always @(posedge clk)
   for (i=0; i<n; i=i+1
    if (row[i]==1)
     if (WE==1) ram_data[i]<=data_in;
           else data_out<=ram_data[i];
endmodule
```

```
‘timescale 1 us / 100 ps
module test_ram1;
 parameter n=8, a=3, w=4, dt = 30;
 reg  [w-1:0] Win;
 reg [a-1:0] addr;
 reg clk, WE;
 integer i;
 wire [w-1:0] Wout;

 ram1 #(n,a,w) RAM(clk, WE, addr, Win, Wout);

always begin #dt clk=1; #dt  clk=0; end

 initial begin
 $display(“time addr Win Wout”);
 $monitor(“%4g   %d   %d  %d”,
            $time, addr, Win, Wout);
 clk<=0; WE<=1; Win<=0; addr<=0;
 for (i=0; i<n; i=i+1) begin @(negedge clk);
                          Win<=Win+1; addr<=addr+1;
                         end
 WE<=0; Win<=0; addr<=0;
 for (i=0; i<n; i=i+1) begin @(negedge clk);
                           addr<=addr+1;
                          end
 $finish;
 end
endmodule
```

(5.20) Same as 5.5, Sorry about that!

(5.21)

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.