```
module mainfsm(clock, globalreset, displaymode,
writemode, wandenable, wandreset, inputreset,
MSGmuxsel, writeenable, loadenable, loaddone,
writefinished);



input clock, globalreset, displaymode, writemode,
writefinished, loaddone;


output inputreset, wandreset, MSGmuxsel, loadenable,
writeenable, wandenable;
reg inputreset, wandreset, MSGmuxsel, loadenable,
writeenable, wandenable;


reg [2:0] state;

parameter Init = 0;
parameter Wait = 1;
parameter Write = 2;
parameter WEreset = 3;
parameter Display = 4;
parameter LEset = 5;
parameter LEreset = 6;
parameter WandEnable = 7;

always@ (posedge clock)
        if (globalreset) state <= Init;
                else
                        case(state)
                                Init:   begin

wandenable <= 0; wandreset <= 1; inputreset <= 1;

MSGmuxsel <= 0; writeenable <= 0; loadenable <= 0;

state <= Wait;
                                                end

                                Wait:   begin
```

```
wandreset <= 0; inputreset <= 0;
if (writemode) state <= Write;
        else if (displaymode) state <= Display;
        else state <= Wait;
                                                end
                                Write:  begin
writeenable <= 1;
if (writefinished) state <= WEreset;
        else state <= Write;
                                                end
                                WEreset:        begin
        writeenable <= 0; state <= Wait;
end

                                Display:        begin
        writeenable <= 0; MSGmuxsel <= 0;
        state <= LEset;
end

                                LEset:  begin
loadenable <= 1; state <= LEreset;
                                                end
                                LEreset:        begin
        loadenable <= 0;
```

```
        if (loaddone) state <= WandEnable;
                else state <= LEreset;
end

                                WandEnable:     begin
        MSGmuxsel <= 1; wandenable <= 1;
end

                                default: state <=
Wait;
                        endcase
endmodule
```

```
# Reading E:/Modeltech_6.0b/tcl/vsim/pref.tcl
# //  ModelSim SE 6.0b Dec  1 2004
# //
# //  Copyright Mentor Graphics Corporation 2004
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
#  OpenFile "//eecslabs/labsroot/users/byte/Desktop/TW5/backtimer.v"
```

```
# Reading E:/Modeltech_6.0b/tcl/vsim/pref.tcl
# //  ModelSim SE 6.0b Dec  1 2004
# //
# //  Copyright Mentor Graphics Corporation 2004
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
#  OpenFile "//eecslabs/labsroot/users/byte/Desktop/TW5/backtimer.v"
```

```
module loadstage(Clock, Reset, LoadEnable, LoadDone,
WriteEnable, WriteAddr, WriteData, ReadAddr,
ReadData);



input Clock, Reset, LoadEnable;
input [6:0] ReadData;
output LoadDone, WriteEnable;
output [3:0] ReadAddr;
output [6:0] WriteData;
output [6:0] WriteAddr;

wire [2:0] Shift;
wire [39:0] Data;
wire [6:0] ROMAddr;


loadmsgfsm loadmsgfsm1(Clock, Reset, LoadEnable,
LoadDone, ReadAddr, ReadData, ROMAddr, Shift,
WriteEnable, WriteAddr);
sr sr1(Clock, Shift, Data, WriteData);
charrom charrom1(ROMAddr, Clock, Data);

endmodule
```

```
module outputstage(Clock, WandReset, WandEnable, Left,
Right, MsgRamAddr, LEDdata, leds);


input Clock, WandReset, WandEnable, Left, Right;
input [6:0] LEDdata;
output [5:0] MsgRamAddr;
output [6:0] leds;


wire CountUp, CountDown, ReStart, StartTimer,
LightsOff, NextLine, ledMuxSel;


wandoutfsm wandoutfsm1(Clock, WandReset, ledMuxSel,
StartTimer, LightsOff, NextLine, CountUp, CountDown,
ReStart, Left, Right, MsgRamAddr, WandEnable);
backtimer backtimer1(Clock, WandReset, CountUp,
CountDown, ReStart);
timer timer1(Clock, StartTimer, LightsOff, NextLine);
ledmux ledmux1(Clock, LEDdata, leds, ledMuxSel);

endmodule
```

```
module loadmsgfsm(clock, reset, LoadEnable, LoadDone,
ReadAddr, Char, ROMAddr, Shift, WriteEnable,
WriteAddr);



input clock, reset, LoadEnable;
input [6:0] Char;  //ASCII Character from buffer
output [6:0] ROMAddr, WriteAddr;
output [3:0] ReadAddr;
output [2:0] Shift;
output WriteEnable, LoadDone;
reg WriteEnable, LoadDone;
reg [6:0] ROMAddr, WriteAddr;
reg [3:0] ReadAddr;
reg [2:0] Shift;
reg [3:0] state;

parameter initialize = 1;
parameter Wait = 2;
parameter CheckChar = 3;
parameter StartWrite = 4;
parameter Wait1 = 5;
parameter Wait2 = 6;
parameter WriteRAM = 7;
parameter WriteDone = 8;
parameter IncShift = 9;
parameter CheckShift = 10;
parameter IncrementChar = 11;
parameter Done = 12;
parameter Wait3 = 13;

always @ (posedge clock)
        begin
                if (reset)   begin
                                                        state
<= initialize;
                                                end
                else case(state)

                        initialize:
                                                        begin

ReadAddr <= 0;
```

```
WriteAddr <= 1;

ROMAddr <= 0;

LoadDone <= 0;

state <= Wait;
                                                    end

                        Wait:
                                begin
                                        if
(LoadEnable) state <= CheckChar;
                                        else state <=
Wait;
                                end

                        CheckChar:
                                begin
                                        if (ReadAddr
== 8) state <= Done;
                                        else state <=
StartWrite;
                                end

                        Done:
                                begin
                                        LoadDone <= 1;
                                        state <=
initialize;
                                end

                        StartWrite:
                                begin
                                        ROMAddr <=
Char;
                                        Shift <= 0;
                                        state <=
Wait1;
                                end

                        Wait1:  state <= Wait2;
```

```
                        Wait2: state <= WriteRAM;

                        WriteRAM:
                                begin
                                        WriteEnable <=
1;
                                        state <=
WriteDone;
                                end

                        WriteDone:
                                begin
                                        WriteEnable <=
0;
                                        state <=
IncShift;
                                end

                        IncShift:
                                begin
                                        Shift <= Shift
+ 1;
                                        WriteAddr <=
WriteAddr + 1;
                                        state <=
CheckShift;
                                end

                        CheckShift:
                                begin
                                        if (Shift ==
5) state <= IncrementChar;
                                        else state <=
Wait2;
                                end

                        IncrementChar:
                                begin
                                        ReadAddr <=
ReadAddr + 1;
                                        state <=
Wait3;
```

```
                                        end

                                Wait3:  state <= CheckChar;
                        endcase
                end
endmodule
```

```
module wandoutfsm(Clock, Reset_Sync, LEDenable,
StartTimer, LightsOff, NextLine, CountUp, CountDown,
StartAgain, Left, Right, ReadAddr, EnableWand);



input Clock, Reset_Sync, LightsOff, NextLine, Left,
Right, StartAgain, EnableWand;
output StartTimer, CountUp, CountDown, LEDenable;
output [5:0] ReadAddr;
reg [5:0] ReadAddr;
reg StartTimer, direction, CountUp, CountDown,
LEDenable;
reg [3:0] state;



parameter initialize = 1;
parameter ResetTimer = 2;
parameter OutputLine = 4;
parameter LEDsOff = 5;
parameter UpdateAddr = 6;
parameter WaitNextLine = 7;
parameter Done = 8;
parameter WaitForLeft = 9;
parameter WaitForRight = 10;
parameter SetDirection = 11;
parameter WaitBackTimer = 12;
parameter ResetTimer2 = 13;


always @ (posedge Clock)
        begin
                if (Reset_Sync)         begin

direction <= 0;

state <= initialize;
                                                end
                else case(state)

                        initialize:
                                                begin
```

```
ReadAddr <= 1;
LEDenable <= 0;
StartTimer <= 0;
CountUp <= 0;
CountDown <= 0;
if (EnableWand) state <= ResetTimer;
else state <= initialize;
                                                    end

                        ResetTimer:
                                                    begin
StartTimer <= 1;
                                                    if
(ReadAddr == 0)                                     state <= Done;
                                                    else
if  (ReadAddr == 48)    state <= Done;
                                                    else
state <= ResetTimer2;
                                                    end
                        ResetTimer2:
                                                    begin
StartTimer <= 0;
                                                    state
<= OutputLine;
                                                    end
                        OutputLine:
                                                    begin
LEDenable <= 1;
                                                    if
(LightsOff)     state <= LEDsOff;
```

```
                                        else
state <= OutputLine;
                                    end
                        LEDsOff:
                                    begin
LEDenable <= 0;
                                        state
<= UpdateAddr;
                                    end
                        UpdateAddr:
                                    begin
                                        state
<= WaitNextLine;
                                        if
(~direction) ReadAddr <= ReadAddr + 1;
                                        else
ReadAddr <= ReadAddr - 1;
                                    end
                        WaitNextLine:
                                    begin
                                        if
(NextLine) state <= ResetTimer;
                                        else
state <= WaitNextLine;
                                    end

                        Done:
                                    begin
CountUp <= 1;
                                        if
(~direction) state <= WaitForLeft;
                                        else
state <= WaitForRight;
                                    end
                        WaitForLeft:
                                    begin
CountUp <= 0;
                                        if
(Left)  state <= SetDirection;
```

```
                                        else
        state <= WaitForLeft;
                                    end
                    WaitForRight:
                                    begin
CountUp <= 0;
                                        if
(Right)         state <= SetDirection;
                                        else
        state <= WaitForRight;
                                    end
                    SetDirection:
                                    begin
CountDown <= 1;
direction <= ~direction;
                                        state
<= WaitBackTimer;
                                        if
(ReadAddr == 48)        ReadAddr <= 47;
                                        else
ReadAddr <= 1;
                                    end
                    WaitBackTimer:
                                    begin
CountDown <= 0;
                                        if
(StartAgain) state <= ResetTimer;
                                        else
state <= WaitBackTimer;
                                    end

                endcase
        end
endmodule
```

```
module inputfsm1(clock, enable, reset, gotkey,
keydata, query_done, start_query, start_timer,
end_timer, Rout, char_ready);



input clock, reset, gotkey, query_done, end_timer,
enable;
input [4:0] keydata;
output start_query, start_timer, char_ready;
reg start_query, start_timer, char_ready;

output [6:0] Rout;

reg [5:0] state;
reg [4:0] R1, R2;
reg [6:0] Rout, Rtemp;

parameter start = 20;
parameter input1 = 21;
parameter query1 = 22;
parameter query1wait = 23;
parameter timer1 = 24;
parameter input2 = 25;
parameter query2 = 26;
parameter query2wait = 27;
parameter compare1 = 28;
parameter LDRAM1 = 29;
parameter swapreg1 = 30;
parameter secondchar = 31;
parameter timer2 = 32;
parameter input3 = 33;
parameter query3 = 34;
parameter query3wait = 35;
parameter compare2 = 36;
parameter LDRAM2 = 37;
parameter swapreg2 = 38;
parameter thirdchar = 39;
parameter timer3 = 40;
parameter input4 = 41;
parameter query4 = 42;
parameter query4wait = 43;
parameter compare3 = 44;
```

```
parameter LDRAM3 = 45;
parameter swapreg3 = 46;
parameter fourthchar = 47;
parameter idle = 48;

always@ (posedge clock) begin
        if (reset) state <= start;
        else if (!enable) state <= idle;
        else
                case(state)
                        start:  begin

start_timer <= 1; char_ready <=0; state <= input1;
Rout <= 0;
                                        end

                        input1: begin

start_timer <= 0;
                                                if
(end_timer == 0) state <= query1;
//loop around through query mode until timer expires.
When it does, re-start timer and wait again

else state <= start;
                                        end

                        query1: begin

start_query <= 1; state <= query1wait;
                                        end
//begin query mode

                        query1wait:     if(gotkey)
begin

                R1 <= keydata; state <= query1wait;

            end
//if the query comes back with a keystroke, store key
data in R1 and loop back once more

else if (query_done == 1 && keydata == 0) state <=
input1;
```

```
//if data is 0 (indicating NO key was pressed by the
user during that query cycle), go back to input1 and
loop again

else if (query_done == 1 && keydata != 0) state <=
timer1;
//if data is not 0 (a button was pressed by the user),
continue on

else begin

                start_query <= 0; state <= query1wait;

         end
//otherwise wait here until query cycle finishes,
reset start_query signal


                        timer1: begin

start_timer <= 1; state <= input2;
                                        end

                        input2: begin

start_timer <= 0;
                                                if
(end_timer == 0) state <= query2;

else begin

                //timer expired, so store R1 in output
register, decode to ascii standard.  send char_ready
signal

                case(R1)

                        1:      Rout <= 7'b0100000;
//ascii space

                        2:      Rout <= 7'b1000001;
//ascii A

                        3:  Rout <= 7'b1000100;
//ascii D
```

```
                        4:  Rout <= 7'b1000111;
//ascii G


                        5:  Rout <= 7'b1001010;
//ascii J

                        6:  Rout <= 7'b1001101;
//ascii M

                        7:  Rout <= 7'b1010000;
//ascii P

                        8:  Rout <= 7'b1010100;
//ascii T

                        9:  Rout <= 7'b1010111;
//ascii W

                        10: Rout <= 7'b0001101;
//ascii Carriage Return

                endcase

                char_ready <= 1;

                state <= start;

          end
                                        end

                        query2: begin

start_query <= 1; char_ready <= 0; state <=
query2wait;
                                        end

                        query2wait:     if(gotkey)
begin

                R2 <= keydata;

                state <= query2wait;
```

```
            end
//store new keystroke in R2 for later comparison

else if (query_done == 1 && keydata == 0) state <=
input2;

else if (query_done == 1 && keydata != 0) state <=
compare1;
//loop back to query mode if no key, otherwise
continue

else begin

                start_query <= 0; state <= query2wait;

         end


                        compare1:       if (R2 == R1)
state <= secondchar;

else state <= LDRAM1;


                        LDRAM1: begin

//store R1 in output register, decode to ascii
standard.  send char_ready signal

case(R1)

1:      Rout <= 7'b0100000; //ascii space

2:      Rout <= 7'b1000001; //ascii A

3:  Rout <= 7'b1000100; //ascii D

4:  Rout <= 7'b1000111; //ascii G



5:  Rout <= 7'b1001010; //ascii J
```

```
6:  Rout <= 7'b1001101; //ascii M

7:  Rout <= 7'b1010000; //ascii P

8:  Rout <= 7'b1010100; //ascii T

9:  Rout <= 7'b1010111; //ascii W

10: Rout <= 7'b0001101; //ascii Carriage Return

endcase

char_ready <= 1;
                                                    state
<= swapreg1;
                                            end
//since the two keystrokes were different, we can
store the first letter into the SRAM buffer


                        swapreg1:       begin

R1 <= R2; char_ready <=0; state <= timer1;
                                                    end
//now that the first letter is stored in the RAM
buffer, transfer the second Register contents
//into R1 and go back to restart timer and allow user
to work with second letter


                        secondchar:     begin

//since the same key was pressed twice, decode to
second letter and store in Rtemp

case(R1)

        1:      Rtemp <= 7'b0001000; //ascii backspace

        2:      Rtemp <= 7'b1000010; //ascii B

        3:  Rtemp <= 7'b1000101; //ascii E
```

```
        4:  Rtemp <= 7'b1001000; //ascii H

        5:  Rtemp <= 7'b1001011; //ascii K

        6:  Rtemp <= 7'b1001110; //ascii N

        7:  Rtemp <= 7'b1010010; //ascii R

        8:  Rtemp <= 7'b1010101; //ascii U

        9:  Rtemp <= 7'b1011000; //ascii X

        10: Rtemp <= 7'b0001101; //ascii Carriage
Return

endcase

state <= timer2;
                                                    end

                        timer2: begin

start_timer <= 1; state <= input3;
                                          end

                        input3: begin

start_timer <= 0;
                                                    if
(end_timer == 0) state <= query3;

else begin

//timer expired, so store R1 in output register,
decode to ascii standard.  send char_ready signal

                case(R1)

                        1:      Rout <= 7'b0001000;
//ascii backspace

                        2:      Rout <= 7'b1000010;
//ascii B
```

```
                        3:  Rout <= 7'b1000101;
//ascii E
                        4:  Rout <= 7'b1001000;
//ascii H
                        5:  Rout <= 7'b1001011;
//ascii K
                        6:  Rout <= 7'b1001110;
//ascii N
                        7:  Rout <= 7'b1010010;
//ascii R
                        8:  Rout <= 7'b1010101;
//ascii U
                        9:  Rout <= 7'b1011000;
//ascii X
                        10: Rout <= 7'b0001101;
//ascii Carriage Return
                endcase
                char_ready <= 1;
                state <= start;
        end
                                        end
                        query3: begin
start_query <= 1; state <= query3wait;
                                        end
                        query3wait:     if(gotkey)
begin
                R2 <= keydata;
```

```
                state <= query3wait;

            end
//store new keystroke in R2 for later comparison

else if (query_done == 1 && keydata == 0) state <=
input3;

else if (query_done == 1 && keydata != 0) state <=
compare2;
//loop back to query mode if no key, otherwise
continue

else begin

                start_query <= 0; state <= query3wait;

         end


                        compare2:       if (R2 == R1)
state <= thirdchar;

else state <= LDRAM2;


                        LDRAM2: begin

//store Rtemp into Rout, send char_ready
                                                Rout
<= Rtemp;

char_ready <= 1;
                                                state
<= swapreg2;
                                            end
//since the two keystrokes were different, we can
store the temp Register into the SRAM buffer


                        swapreg2:       begin

R1 <= R2; char_ready <=0; state <= timer1;
```

```
                                                    end
//now that the first letter is stored in the RAM
buffer, transfer the second Register contents
//into R1 and go back to restart timer and allow user
to work with second letter


                        thirdchar:      begin

//since the same key was pressed three times, decode
to third letter and store in Rtemp

case(R1)

        1:      Rtemp <= 7'b0001000; //ascii backspace

        2:      Rtemp <= 7'b1000011; //ascii C

        3:  Rtemp <= 7'b1000110; //ascii F

        4:  Rtemp <= 7'b1001001; //ascii I

        5:  Rtemp <= 7'b1001100; //ascii L

        6:  Rtemp <= 7'b1001111; //ascii O

        7:  Rtemp <= 7'b1010011; //ascii S

        8:  Rtemp <= 7'b1010110; //ascii V

        9:  Rtemp <= 7'b1011001; //ascii Y

        10: Rtemp <= 7'b0001101; //ascii Carriage
Return

endcase

state <= timer3;
                                                    end

                        timer3: begin

start_timer <= 1; state <= input4;
```

```
                                        end

                        input4: begin

start_timer <= 0;
                                                if
(end_timer == 0) state <= query4;

else begin

//timer expired, so store R1 in output register,
decode to ascii standard.  send char_ready signal

                case(R1)

                        1:      Rout <= 7'b0001000;
//ascii backspace

                        2:      Rout <= 7'b1000011;
//ascii C

                        3:  Rout <= 7'b1000110;
//ascii F

                        4:  Rout <= 7'b1001001;
//ascii I

                        5:  Rout <= 7'b1001100;
//ascii L

                        6:  Rout <= 7'b1001111;
//ascii O

                        7:  Rout <= 7'b1010011;
//ascii S

                        8:  Rout <= 7'b1010110;
//ascii V

                        9:  Rout <= 7'b1011001;
//ascii Y

                        10: Rout <= 7'b0001101;
//ascii Carriage Return
```

```
                endcase

                char_ready <= 1;

                state <= start;

        end
                                        end

                        query4: begin

start_query <= 1; state <= query4wait;
                                        end

                        query4wait:     if(gotkey)
begin

                R2 <= keydata;

                state <= query4wait;

           end
//store new keystroke in R2 for later comparison

else if (query_done == 1 && keydata == 0) state <=
input4;

else if (query_done == 1 && keydata != 0) state <=
compare3;
//loop back to query mode if no key, otherwise
continue

else begin

                start_query <= 0; state <= query4wait;

        end


                        compare3:       if (R2 == R1)
state <= fourthchar;

else state <= LDRAM3;
```

```
                        LDRAM3: begin

//store Rtemp into Rout, send char_ready
                                                Rout
<= Rtemp;

char_ready <= 1;
                                                state
<= swapreg3;
                                            end
//since the two keystrokes were different, we can
store the first Register into the SRAM buffer


                        swapreg3:       begin

R1 <= R2; char_ready <= 0; state <= timer1;
                                                end
//since we have hit the same key 4 times now

                        fourthchar:     begin

//since the same key was pressed four times, decode to
4th char and store in Rout. send char_ready

case(R1)

        1:      Rout <= 7'b0110001; //ascii 1

        2:      Rout <= 7'b0110010; //ascii 2

        3:  Rout <= 7'b0110011; //ascii 3

        4:  Rout <= 7'b0110100; //ascii 4

        5:  Rout <= 7'b0110101; //ascii 5

        6:  Rout <= 7'b0110110; //ascii 6

        7:  Rout <= 7'b0110111; //ascii 7

        8:  Rout <= 7'b0111000; //ascii 8
```

```
        9:  Rout <= 7'b0111001; //ascii 9

        10: Rout <= 7'b0001101; //ascii Carriage
Return

endcase

char_ready <= 1;

state <= start;
                                                    end

                        idle:   state <= idle;

                        default: state <= start;
                endcase
        end
endmodule
```

```
module inputtop(clock, enable, reset, IN1, IN2, IN3,
IN4, IN5, IN6, IN7, IN8, IN9, IN10, ready, char);


input clock, enable, reset, IN1, IN2, IN3, IN4, IN5,
IN6, IN7, IN8, IN9, IN10;
output ready;
//reg ready;

output [6:0] char;
//reg [6:0] char;

wire startQ, LD_Key, done, Tstart, Tstop, reset_S,
enable_S, IN1_S, IN2_S, IN3_S, IN4_S, IN5_S, IN6_S,
IN7_S, IN8_S, IN9_S, IN10_S;
wire [4:0] key;

synch S(clock, reset, enable, IN1, IN2, IN3, IN4, IN5,
IN6, IN7, IN8, IN9, IN10, reset_S, enable_S, IN1_S,
IN2_S, IN3_S, IN4_S, IN5_S, IN6_S, IN7_S, IN8_S,
IN9_S, IN10_S);




inputfsm1 IN(clock, enable_S, reset_S, LD_Key, key,
done, startQ, Tstart, Tstop, char, ready);


keytimer KT(clock, Tstart, Tstop);

keypadcheck1 KP(clock, reset_S, startQ, key, LD_Key,
done, IN1_S, IN2_S, IN3_S, IN4_S, IN5_S, IN6_S, IN7_S,
IN8_S, IN9_S, IN10_S);



endmodule
```

```
module keypadcheck1(clock, reset, keyquery,
keypressed, gotinput, finished, in1, in2, in3, in4,
in5, in6, in7, in8, in9, in10);



input clock, reset, keyquery, in1, in2, in3, in4, in5,
in6, in7, in8, in9, in10;


output gotinput, finished;
reg gotinput, finished;

output [4:0] keypressed;
reg [4:0] keypressed;

reg [3:0] state;

parameter IDLE = 0;
parameter START = 1;
parameter WAIT1 = 2;
parameter WAIT2 = 3;
parameter WAIT3 = 4;
parameter WAIT4 = 5;
parameter WAIT5 = 6;
parameter WAIT6 = 7;
parameter WAIT7 = 8;
parameter WAIT8 = 9;
parameter WAIT9 = 10;
parameter WAIT10 = 11;
parameter GOTIN = 12;
parameter FINISH = 13;


always@ (posedge clock)
        if (reset) state <= IDLE;
                else
                        case(state)
                                IDLE:   begin

finished <= 0; gotinput <= 0;

        if(keyquery) state <= START;
```

```
                else state <= IDLE;
                                                end

                                    START:  if (!in1)
state <= WAIT1;
else if (!in2) state <= WAIT2;
else if (!in3) state <= WAIT3;
else if (!in4) state <= WAIT4;
else if (!in5) state <= WAIT5;
else if (!in6) state <= WAIT6;
else if (!in7) state <= WAIT7;
else if (!in8) state <= WAIT8;
else if (!in9) state <= WAIT9;
else if (!in10) state <= WAIT10;

else begin
                keypressed <= 0; finished <= 1; state
<= IDLE;
        end

                                    WAIT1:  if (!in1)
state <= WAIT1;
else begin
                keypressed <= 1; state <= GOTIN;
        end
```

```
                                WAIT2:  if (!in2)
state <= WAIT2;
else begin
                keypressed <= 2; state <= GOTIN;
        end

                                WAIT3:  if (!in3)
state <= WAIT3;
else begin
                keypressed <= 3; state <= GOTIN;
        end

                                WAIT4:  if (!in4)
state <= WAIT4;
else begin
                keypressed <= 4; state <= GOTIN;
        end

                                WAIT5:  if (!in5)
state <= WAIT5;
else begin
                keypressed <= 5; state <= GOTIN;
        end

                                WAIT6:  if (!in6)
state <= WAIT6;
else begin
```

```
                keypressed <= 6; state <= GOTIN;
        end

                                WAIT7:  if (!in7)
state <= WAIT7;
else begin
                keypressed <= 7; state <= GOTIN;
        end

                                WAIT8:  if (!in8)
state <= WAIT8;
else begin
                keypressed <= 8; state <= GOTIN;
        end

                                WAIT9:  if (!in9)
state <= WAIT9;
else begin
                keypressed <= 9; state <= GOTIN;
        end

                                WAIT10: if (!in10)
state <= WAIT10;
else begin
                keypressed <= 10; state <= GOTIN;
        end
```

```
                                GOTIN:  begin
gotinput <= 1; state <= FINISH;
                                                    end
                                FINISH: begin
gotinput <= 0; finished <= 1; state <=IDLE;
                                                    end
                                default: state <=
IDLE;
                        endcase
endmodule
```

```
module keytimer(clock, startkey, endkey);

input clock, startkey;
output endkey;
reg endkey;
reg [30:0] counter;

parameter keytime = 27000000;

always @ (posedge clock) begin
        if (startkey) begin
                                counter <= 0;
                                endkey <= 0;
                        end
        else if (counter == keytime) endkey <=1;
        else counter <= counter + 1;
end

endmodule
```

```
module ledmux(Clock, LEDdata, LEDout, ledMuxSel);

input Clock, ledMuxSel;
input [6:0] LEDdata;
output [6:0] LEDout;
reg [6:0] LEDout;

always @ (posedge Clock)
        begin
                if (ledMuxSel)  LEDout <= LEDdata;
                else             LEDout <= 0;
        end

endmodule
```

```
module msgRamMux(Clock, WriteAddr, ReadAddr, RAMaddr,
MsgMuxSel);


input Clock, MsgMuxSel;
input [5:0] WriteAddr, ReadAddr;
output [5:0] RAMaddr;
reg [5:0] RAMaddr;

always @ (posedge Clock)
        begin
                if (MsgMuxSel)  RAMaddr <= ReadAddr;
                else                    RAMaddr <=
WriteAddr;
        end

endmodule
```

```
module sr(clock, Shift, Data, Out);

input clock;
input [2:0] Shift;
input [39:0] Data;
output [6:0] Out;
reg [6:0] Out;

always @ (posedge clock)
        begin
                case(Shift)

                        0:      begin
                                        Out <=
Data[38:32];
                                end
                        1:      begin
                                        Out <=
Data[30:24];
                                end
                        2:      begin
                                        Out <=
Data[22:16];
                                end
                        3:      begin
                                        Out <=
Data[14:8];
                                end
                        4:      begin
                                        Out <=
Data[6:0];
                                end
                endcase
        end
endmodule
```

```
module synch2(Clock, Reset, Left, Right, Reset_Sync,
Left_Sync, Right_Sync, DispMode, WriteMode,
DispModeS,WriteModeS);



input Clock, Reset, Left, Right, DispMode, WriteMode;
output Reset_Sync, Left_Sync, Right_Sync, DispModeS,
WriteModeS;
reg Reset_Sync, Left_Sync, Right_Sync, DispModeS,
WriteModeS;


always @ (posedge Clock)
        begin
                Reset_Sync <= Reset;
                Left_Sync <= Left;
                Right_Sync <= Right;
                DispModeS <= DispMode;
                WriteModeS <= WriteMode;
        end

endmodule
```

```
module synch(clock, reset, enable, IN1, IN2, IN3, IN4,
IN5, IN6, IN7, IN8, IN9, IN10, reset_s, enable_s,
IN1_s, IN2_s, IN3_s, IN4_s, IN5_s, IN6_s, IN7_s,
IN8_s, IN9_s, IN10_s);



input clock, enable, reset, IN1, IN2, IN3, IN4, IN5,
IN6, IN7, IN8, IN9, IN10;


output reset_s, enable_s, IN1_s, IN2_s, IN3_s, IN4_s,
IN5_s, IN6_s, IN7_s, IN8_s, IN9_s, IN10_s;
reg reset_s, enable_s, IN1_s, IN2_s, IN3_s, IN4_s,
IN5_s, IN6_s, IN7_s, IN8_s, IN9_s, IN10_s;


always@ (posedge clock)
        begin
                reset_s <= reset;
                enable_s <= enable;
                IN1_s <= IN1;
                IN2_s <= IN2;
                IN3_s <= IN3;
                IN4_s <= IN4;
                IN5_s <= IN5;
                IN6_s <= IN6;
                IN7_s <= IN7;
                IN8_s <= IN8;
                IN9_s <= IN9;
                IN10_s <= IN10;
        end

endmodule
```

```
module timer(clock, StartTimer, LightsOff, NextLine);

input clock, StartTimer;
output LightsOff, NextLine;
reg LightsOff, NextLine;
reg [30:0] counter;

/*parameter ontime = 20;
parameter offtime = 40;*/

parameter ontime = 8000000;
parameter offtime = 32000000;

always @ (posedge clock)
  begin
        if (StartTimer)
                begin
                        counter <= 4'd0;
                        LightsOff <= 0;
                        NextLine <= 0;
                end
        else if (counter == ontime)
                begin
                        LightsOff <= 1;
                        counter <= counter + 1;
                end
        else if (counter == offtime)
                begin
                        NextLine <= 1;
                end
        else    counter <= counter + 1;

  end

endmodule
```

```
module top(Clock, reset, Left, Right, IN1, IN2, IN3,
IN4, IN5, IN6, IN7, IN8, IN9, IN10, leds, DispMode,
WriteMode);



input Clock, reset, Left, Right, DispMode, WriteMode;
input IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9,
IN10;
output [6:0] leds;
wire Clock, Reset_Sync, Left_Sync, Right_Sync,
LoadEnable;
wire LoadDone, MsgWriteEnable, WandEnable, WandReset,
WriteEnable, InputReset, ready, MsgMuxSel, DispModeS,
WriteModeS;
wire writedone;
wire [3:0] ReadCharAddr;
wire [5:0] ReadAddr, WriteAddr, RAMaddr;
wire [6:0] LineData, char, RamChar, LEDdata;



outputstage outputstage1(Clock, WandReset, WandEnable,
Left_Sync, Right_Sync, ReadAddr, LEDdata, leds);
synch2 synch3(Clock, reset, Left, Right, Reset_Sync,
Left_Sync, Right_Sync, DispMode, WriteMode,
DispModeS,WriteModeS);
loadstage loadstage1(Clock, Reset_Sync, LoadEnable,
LoadDone, MsgWriteEnable, WriteAddr, LineData,
ReadCharAddr, RamChar);
msgRamMux msgRamMux1(Clock, WriteAddr, ReadAddr,
RAMaddr, MsgMuxSel);
msgram msgram1 (RAMaddr, Clock, LineData, LEDdata,
MsgWriteEnable);
bufferstage bufferstage1(Clock, InputReset,
WriteEnable, char, ready, writedone, RamChar,
ReadCharAddr);
inputtop inputtop1(Clock, WriteEnable, InputReset,
IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9, IN10,
ready, char);
mainfsm mainfsm1(Clock, Reset_Sync, DispModeS,
WriteModeS, WandEnable, WandReset, InputReset,
MsgMuxSel, WriteEnable, LoadEnable, LoadDone,
writedone);
```

```
endmodule
```

```
# Reading E:/Modeltech_6.0b/tcl/vsim/pref.tcl
# //  ModelSim SE 6.0b Dec  1 2004
# //
# //  Copyright Mentor Graphics Corporation 2004
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
#  OpenFile "//eecslabs/labsroot/users/byte/Desktop/TW5/backtimer.v"
```