A Traffic Light Controller

Dan Ports

October 8, 2003

Contents

List of Figures

List of Tables

1 Overview

This device controls the two red/yellow/green traffic signals at an intersection, based on programmable timing parameters and two sensors. One street is designated as the main street and the other as the side street. A signal input is provided for a traffic sensor that allows the side street's green light interval, normally shorter than that of the main street, to be extended. A walk request button is provided, which allows the intersection to display a walk signal for an interval after the main street light turns red. This walk signal is indicated by activating the red and yellow lights on both street lights.

2 Operation

The user can operate this controller via two mode switches (F_1 and F_0), a GO button which executes the command indicated by F_1 and F_0, and a RESET button that returns the controller to its default idle state. Possible commands and the corresponding settings for the F switches are listed in Table 1.

Table 1: Commands

F_1	F_0	Command
0	0	Read Memory Location
0	1	Write Memory Location
1	0	Run Traffic Lights
1	1	Blink Traffic Lights

To read a timing parameter from memory, the controller must be placed in Read Memory Location mode by setting the mode switches appropriately and pressing GO. The location can then specified using the two address switches, A_1 and A_0, and the output will be displayed on a hex LED. To write to a memory location, the A switches are used to specify the location, and the new value is entered using the four C switches. After selecting the write command and pressing GO, the new value is written to memory and read back on the hex LED for verification. For a list of memory locations, see Table 2.

Table 2: Memory locations

A_1	A_0	Name	Nominal value
0	0	TYEL	3
0	1	TBASE	6
1	0	TEXT	6
1	1	TBLINK	1

When the Run Traffic Lights command is entered on the F switches, and the GO button is pressed, the green light on the main street and red light on the side street turn on. After TBASE + TEXT seconds have elapsed, the main light switches to yellow for TYEL seconds. Afterwards, if the walk button has been pressed, both lights will display a walk signal (red and yellow both on) for TEXT seconds. If the walk button has not been pressed, or after the walk interval is completed, the main light turns red and the side light turns green for TBASE seconds. If the sensor is active at the end of this interval, the side green interval will be extended for another TBASE seconds.

Afterwards, the side light turns yellow for TYEL seconds. Then the side light turns red and the main light turns green, and the cycle repeats.

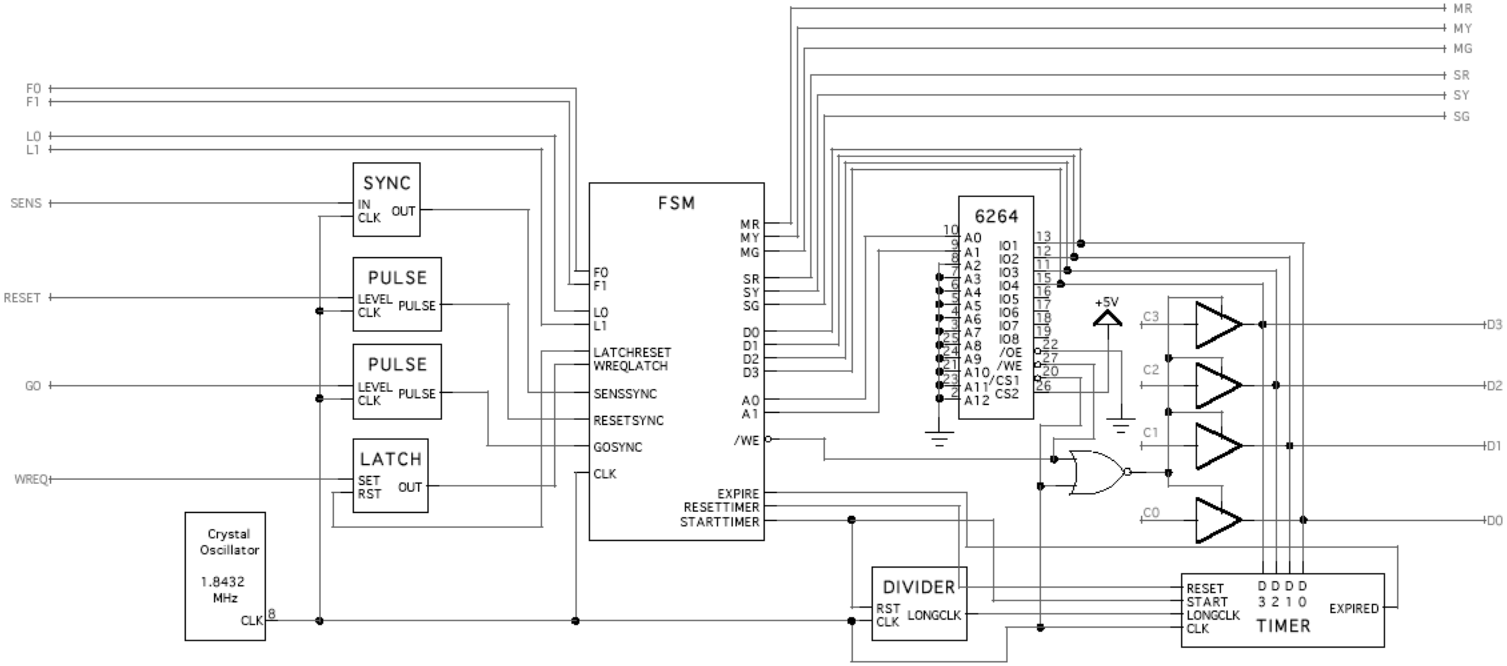
If the Blink Traffic Lights command is selected and the GO button is pressed, the red light on the side street and the yellow light on the main street blink repeatedly, turning on for TBLINK seconds then off for another TBLINK seconds.

3 Design

3.1 Overview

The design of this traffic light controller is represented by the block diagram shown in Figure 1. All synchronous components operate on a 1.8432 MHz clock signal provided by a crystal oscillator. The system is controlled by a finite state machine (FSM) that keeps track of the current state of the system and generates the appropriate traffic light outputs. The FSM is controlled by the user through the switches, which are conditioned by passing through the Synchronizer, Level to Pulse, and Latch modules. These ensure that all input signals are synchronized to the system clock, that push-button inputs are only active for one clock cycle per button-press, and that the walk request signal is stored until it can be serviced. Timing parameters are stored in a 6264 static RAM (SRAM) module. The FSM controls the SRAM, selecting addresses and the read or write mode as appropriate. A Divider module converts the high-frequency clock signal into a once-per-second "long clock" signal. The Timer module uses this long clock and the timing parameters read from SRAM to measure the appropriate time interval, then signal the FSM.

Figure 1: System block diagram



All components except for the SRAM and clock are implemented using an Altera FLEX 10K10 FPGA, as depicted in the wiring diagram in Figure 2. The VHDL files used to program the FPGA are attached in an appendix. The clock is a 1.8432 MHz crystal oscillator, and the SRAM module is a MCM6264C 8K x 8 Bit static RAM. Inputs are provided using debounced switches for the F, L, C, and SENSOR inputs, and debounced push-buttons for the GO, RESET, and WalkRequest signals. The data (D) on the SRAM input/output bus are displayed on a hex LED, and the traffic signals are displayed on a set of six LEDs. In addition, an 8-pin DIP connector is provided for controlling an external traffic light, with pinout listed in Table 3.

Figure 2: Hardware wiring diagram

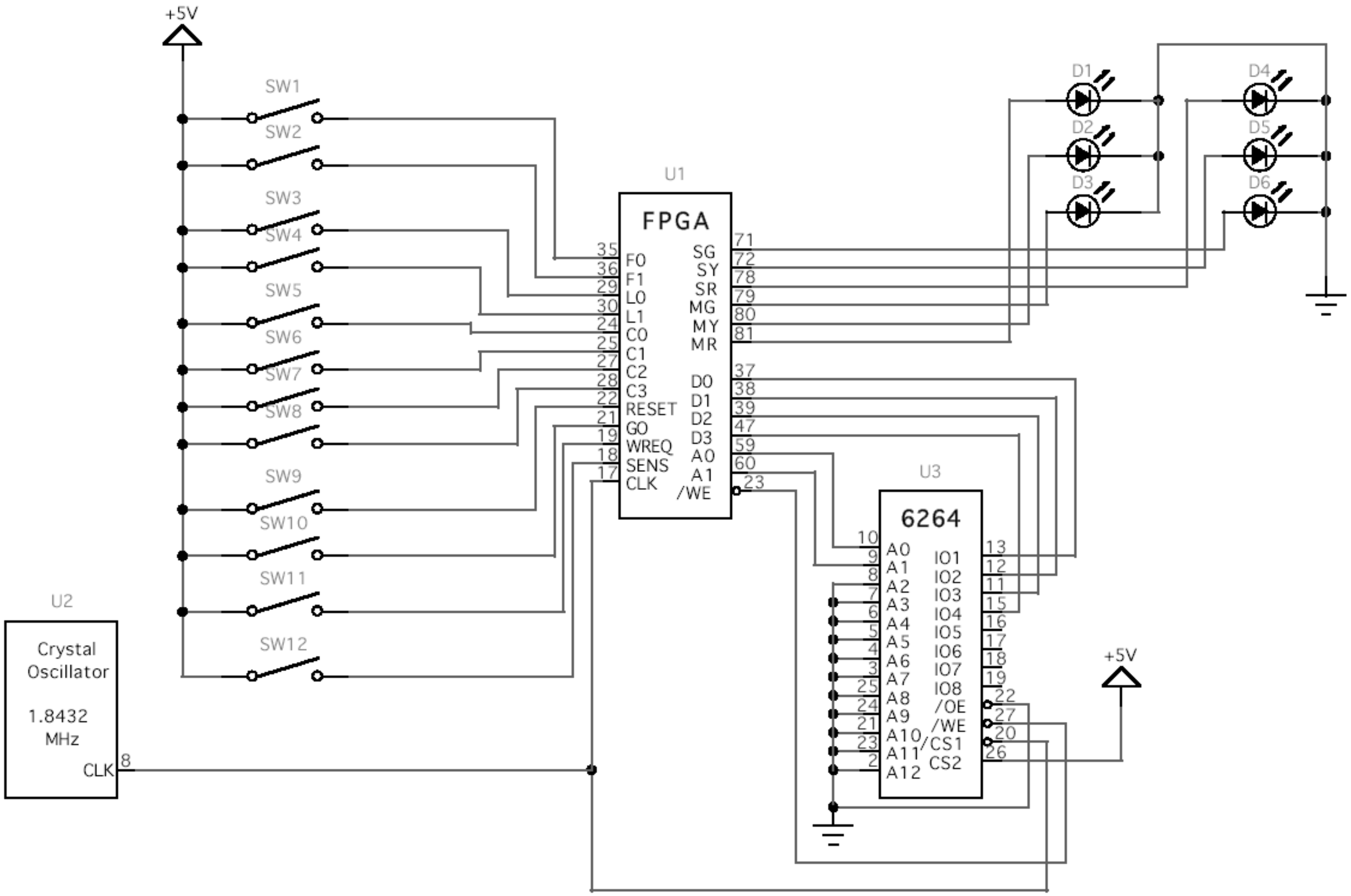


Table 3: Traffic signal light connections

Pin number	Signal
1	+5V
2	Main Street Green
3	Main Street Yellow
4	Main Street Red
5	Side Street Green
6	Side Street Yellow
7	Side Street Red
8	Ground

3.2 Clock

Nearly all components in this system operate on the rising edge of a global clock signal. Because the standard 10 MHz NuBus clock signal was too fast to operate the SRAM chip with acceptable access time, an external 1.8432 MHz crystal oscillator was used to generate the clock signal.

3.3 FSM

The synchronous finite state machine used in this system has twelve possible states. These states, their outputs, and the transitions between them are shown in the state diagram in Figure 3. All outputs not listed in a state's bubble in the diagram are set to their default value (0 for active high signals and 1 for active high signals).

Internally, the FSM operates by maintaining signals that keep track of the current state and the next state. The FSM is defined by three processes that operate concurrently. One process determines the next state signal asynchronously as a function of the current state and the inputs. Another process determines the outputs as a function of the current state. Because the outputs depend only on the current state of the FSM, the FSM is a Moore machine.

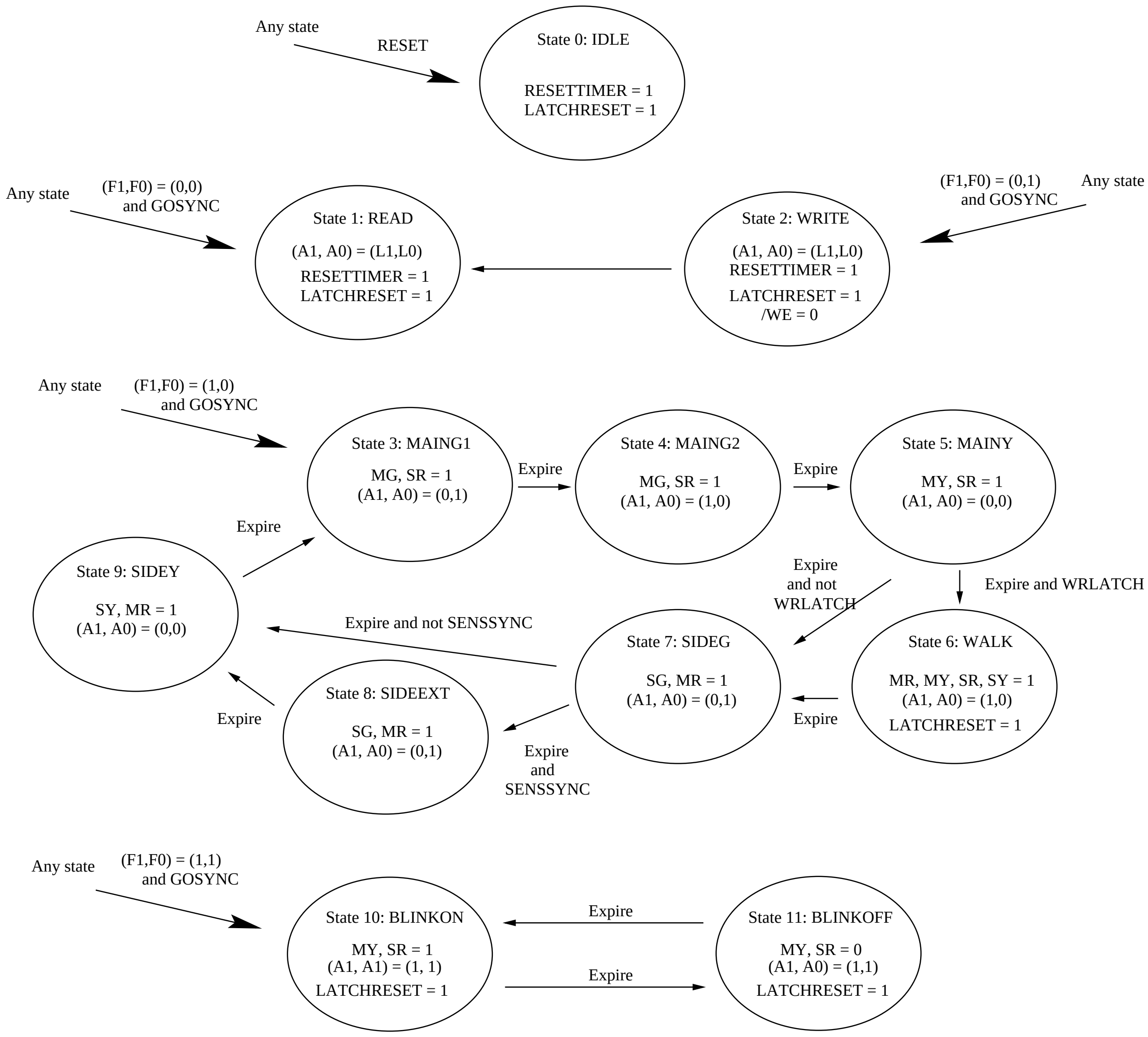
The third process handles state transitions. State transitions can only occur on a rising edge of the clock. Normally, the transitions also require that the EXPIRED signal from the timer is high; however, if the RESET or GOSYNC signals are high, or the current state is IDLE, READ, or WRITE, the state transition happens the next rising edge of the clock. This logic ensures that, regardless of the current state, pressing the RESET button always immediately returns the system to the IDLE state and pressing the GO button immediately moves the FSM to either the READ, WRITE, MAING1, or BLINKON states, depending on the current values of F_1 and F_0. This process also generates the STARTTIMER signal. This signal is high for one clock period after a state transition, except for transitions into the IDLE, READ, or WRITE stage.

The system starts up in the IDLE state. All traffic light outputs are disabled. The FSM can be returned to this state at any time by pressing the IDLE button.

To read a value, the user sets $F = 00$ and presses GO. This moves the system to the READ state. In this state the FSM's RAM address output line A matches the location input set on the L switches. The value output by the RAM will be displayed on the hex LED connected to the RAM I/O lines. The FSM stays in the READ state until the RESET or GO button is pressed. The L inputs are not latched, so changing the L switches while the FSM is in READ mode allows a different address location to be read.

To read a value, the user sets $F = 01$ and presses GO. This moves the FSM to the WRITE state. As in the READ state, the FSM's RAM address output line A matches the location input set on the L switches. The $\overline{\mathrm{WE}}$ write enable signal is set low to instruct the SRAM to write. After one

Figure 3: State diagram for FSM



clock cycle has elapsed, the FSM moves into the READ state to read back the value it has written to the SRAM and allow the user to verify that it is correct.

The RESETTIMER and LATCHRESET signals are set high during the IDLE, READ, and WRITE states. This ensures that the timer is not counting, and that any walk request signals received during these states will be discarded.

Setting $F = 10$ and pressing GO puts the FSM in the MAING1 state, beginning the traffic light operation. In this state, the main street is green and the side street is red; $A = 01$, so this state is held for TBASE seconds. Then the FSM moves to the MAING2 state, which holds the same traffic light outputs for another TEXT seconds ($A = 10$). When the timer expires, the next state is MAINY, in which the main light is yellow and the side street is red. The timer is set for TYEL seconds ($A = 01$). If WREQLATCH is high when the timer expires, the next state is WALK; otherwise, the WALK state is skipped and the FSM moves directly to the SIDEG state. In the WALK state, the red and yellow lights on both streets are on, $A = 10$ to set the timer for TEXT seconds, and the LATCHRESET signal is set in order to clear the latched walk signal. After the WALK state completes, or if it is skipped, the next state is SIDEG, in which the side light is green and the main light is red. When the timer expires after TBASE ($A = 01$) seconds, the next state is SIDEEXT if SENSSYNC is high, or SIDEY otherwise. In the SIDEEXT state, all outputs are the same as in SIDEG; this holds the side light green for another TBASE seconds. Afterwards, the FSM moves to the SIDEY state, in which the side light is yellow and the main light is red. A is set to 00 to hold this state for TYEL seconds; when the timer expires, the FSM moves back to the MAING1 state, and repeats the cycle.

When the user sets $F = 11$ and presses GO, the FSM begins a blink sequence with the BLINKON state. The yellow light on the main street and red light on the side street are turned on. The timer is set for TBLINK seconds ($A = 11$). When the timer expires, the FSM moves to the BLINKOFF state, in which all lights are off. In this state, A is also set to 11. When the timer expires, the FSM moves back to BLINKON. Thus a loop is created in which the red and yellow lights are blink on for TBLINK seconds and off for TBLINK seconds.

Figure 4: Timing diagram for FSM — IDLE, READ, WRITE, and BLINK modes

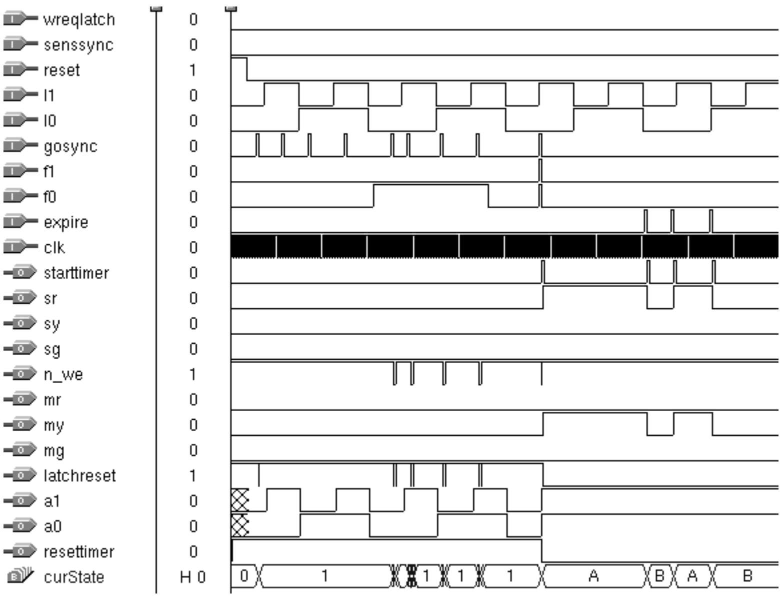
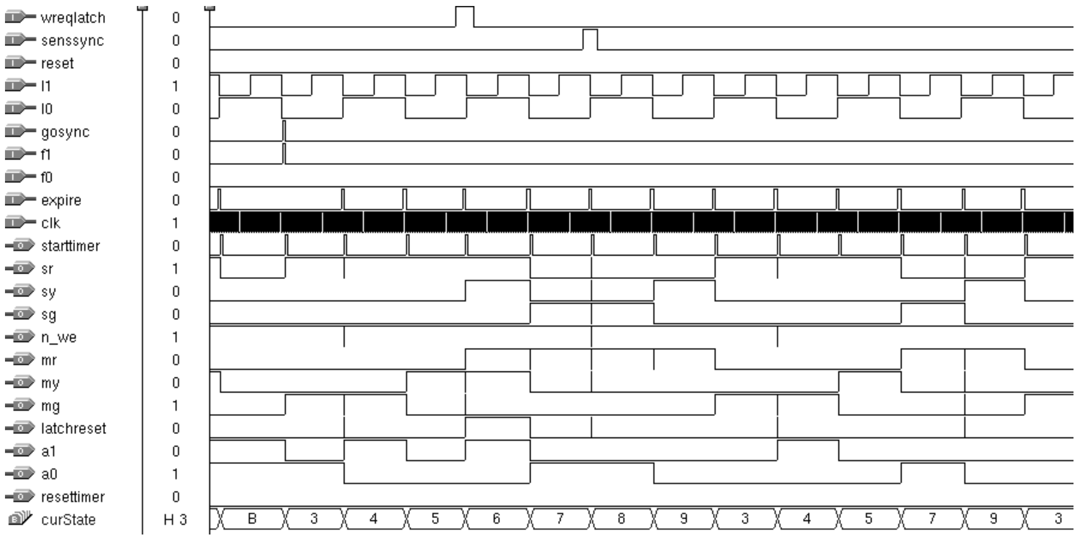


Figure 5: Timing diagram for FSM — RUN mode



3.4 Synchronizer

The input signals to the system are asynchronous with respect to the clock. A synchronizer is used to generate a SENSSYNC signal, which is a synchronized version of the SENSOR input. The synchonized output is guaranteed not to change except on rising edges of the clock.

The implementation of the synchronizer is simply a D flip-flop, as in Figure 6.

Figure 6: Synchronizer logic diagram

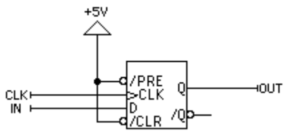
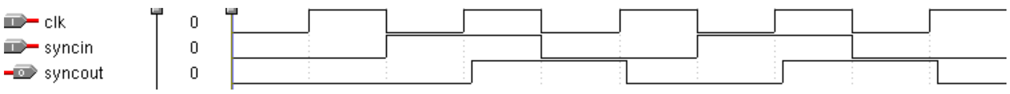


Figure 7: Synchronizer timing diagram



3.5 Level to Pulse

In addition to being asynchronous, the signals connected to push buttons (GO and RESET) will be high for many clock cycles (since one clock cycle at this frequency is about 500 nanoseconds, and it certainly is not possible to press a button this quickly). The level-to-pulse module compensates for this, generating a pulse one clock cycle in width when the input level changes from low to high.

The implementation of the Level to Pulse module uses three D flip-flops in sequence and an AND gate (Figure 8. The first flip-flop synchronizes the input to prevent metastability issues. The next two flip-flops are used to generate a pulse. By taking the AND of the second flip-flop's non-inverting output and third's inverting output, we ensure that the output can only be high for one clock cycle: when the high input signal has propagated to the second flip-flop but not the third. This generates a pulse.

Two identical Level to Pulse modules are used in this device, each synchronizing one input signal, as shown in Figure 1.

Figure 8: Level to Pulse logic diagram

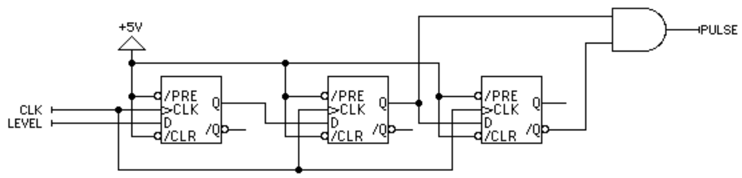
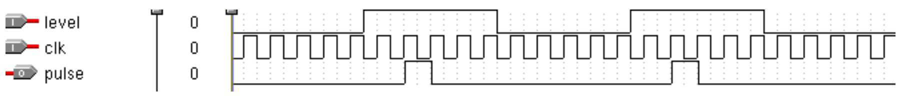


Figure 9: Level to Pulse timing diagram



3.6 Latch

The latch module is used to keep track of whether the walk request button has been pressed. This is a SR latch, which sets its output value high when the SET signal (connected to the push-button) goes high, and sets its output low when the RESET signal (connected to the FSM) goes high. if both the SET and RST signals are asserted at the same time, the output is low. An implementation is shown in Figure 10.

Figure 10: Latch logic diagram

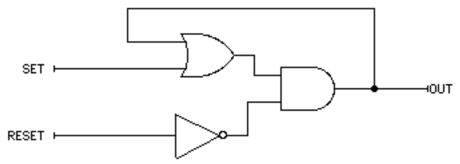
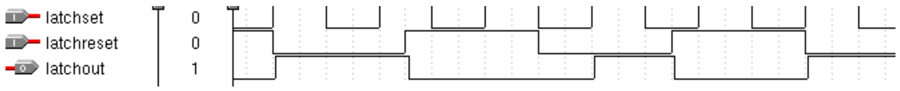


Figure 11: Latch timing diagram



3.7 SRAM

The memory used in this system is on a MCM6264C SRAM chip external to the FPGA. Only the two lowest-order address bits on the chip are used; these two are connected to the FSM's A_0 and A_1 outputs, and the other address lines on the 6264 are all tied to ground. The IO_1 through IO_4 lines of the SRAM are connected to the D_0 through D_3 lines of the FPGA, as well as to a hex LED for displaying their value. All other IO lines are left unconnected

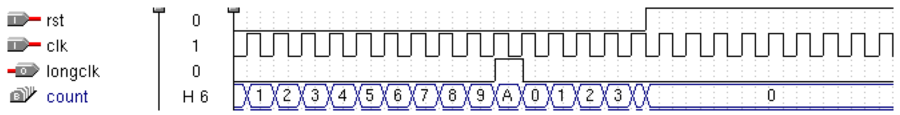
The chip's mode is controlled by its $\overline{\mathrm{WE}}$ write enable and $\overline{\mathrm{CS}_1}$ chip select inputs. The CS_2 chip select and $\overline{\mathrm{OE}}$ output enable lines are permanently tied to their active states (high and low respectively). The $\overline{\mathrm{WE}}$ input is connected to the FSM's $\overline{\mathrm{WE}}$ output, and the $\overline{\mathrm{CS}_1}$ input is connected to the clock. By connecting the active low chip select input to the clock, we ensure that the SRAM chip is only active during the low half of the clock cycle. Since the FSM outputs only change in response to the rising edge of the clock, this allows the FSM outputs time to stabilize before the RAM processes them. Thus glitches on the write-enable or address lines will not cause memory locations to be undesirably overwritten.

3.8 Divider

The purpose of the divider is to generate a 1 Hz "long clock" signal from the 1.8432 MHz clock signal. The output of the divider is a signal that is high for one pulse of the system clock, once per second. This is done by maintaining a counter that is incremented each clock pulse. (A 24-bit counter was used because this module was originally designed to work with a 10 MHz clock; a 21-bit counter would have been adequate for this purpose.) When the counter reaches 1,843,199, the output is set high. On the next clock pulse (when the counter is 1,843,200), the counter is reset to zero and the output is set low again.

The divider also has a reset signal that allows the counter to be reset to zero. This is connected to the FSM's STARTTIMER signal. By resetting the divider whenever the timer starts counting, we ensure that the timer does not begin counting with a fraction of a second.

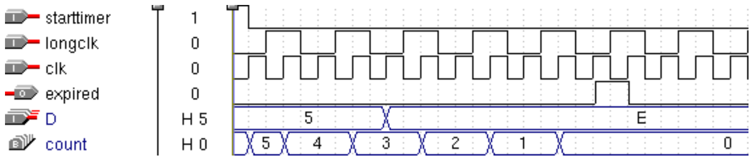
Figure 12: Divider timing diagram (ratio changed from 1,843,200:1 to 10:1)



3.9 Timer

The timer uses the output of the divider to measure a time interval specified by the data bus. The timer maintains an internal 4-bit counter. When the timer's START input is asserted on a rising edge of the clock, the buffer is initialized to the contents of the data bus. Thereafter, whenever the LONGCLK signal is high on a rising edge of the clock, the counter is decremented by 1. When the clock is zero, the timer generates a one clock cycle long pulse on the EXPIRED output signal, and stops counting. A RESET input is provided; if this input is high on a rising edge of the clock, the timer stops counting. This ensures that the timer is not counting when the FSM is not expecting input from the timer.

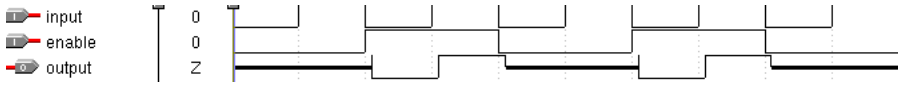
Figure 13: Timer timing diagram



3.10 Tristate Buffer

The tristate buffer module used in this input takes one input and an enable signal, and generates one output. The output is equal to the input when the enable signal is high, and in a tri-state (high-impedance) state when the enable signal is low. In this design, four identical tristate buffers are used to connect the C_i inputs from the switches to the D_i data bus lines. The enable signal to each of these tristate buffers is the logical NOR of the write-enable and clock signals. Thus, the C switches only control the data bus when the write-enable signal is low (active) and the clock is also low. Since the SRAM module is in write mode for this combination of signals, this ensures that no contention can occur on the data bus.

Figure 14: Tristate buffer timing diagram



4 Testing

All modules of the system were tested first in simulation using MAX+PLUS II software, then in hardware once the FPGA was programmed.

To test the synchronizer, level to pulse unit, latch, and tri-state buffer, waveforms were generated that tested all possible combinations of inputs and outputs. After the simulation was run, all outputs were verified to be correct.

To simplify testing the divider, it was modified slightly so that it generated a pulse every 10 cycles of the clock rather than every 1,843,200. This made the behavior easier to visualize and validate in simulation. A clock signal was applied to the input, and the output and state of the internal counter were verified. The reset signal was also tested.

The timer was tested in simulation using a waveform that tested the normal operation of the counter for several different values of D as well as operating the RESET input. The output of the counter was correct.

The FSM was tested in simulation using a waveform that tested the possible states and transitions. The FSM was first reset, then placed in each of the READ, WRITE, and BLINK modes using the GOSYNC and F signals. The correct outputs and internal state variables were observed for each state. Next, the traffic light operation of the FSM was tested by placing it in traffic light mode.

The inputs from the RAM and timer were simulated. By using both values of the SENSSYNC and WREQLATCH signals, all possible states and all possible state transitions were verified. Correct outputs were verified.

After simulation testing was completed, the FPGA was programmed and tested in hardware. For testing purposes, an interface was provided for several intermediate signals and they were connected to hex LEDs: the FSM's state variable, the counter's count variable, and the RAM address lines. The RAM was programmed with the nominal values for each timing parameter. As in simulation, the controller was operated in READ, WRITE, BLINK, and RUN modes, and all possible states and transitions were observed. The debugging outputs (state, count, and address lines) all had the correct outputs, and the traffic signal outputs were correct.

All testing was completed successfully.

5 Conclusion

A traffic light controller meeting the desired specifications can be built using a synchronous finite state machine implemented using a FPGA. The controller was successfully implemented and tested.

A Appendix: VHDL Source

A.1 top.vhd

```
-- top.vhd: top-level structure for traffic light controller
-- Dan R. K. Ports   <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity top is

  port (
    clk, sens, wreq, go, reset, c0, c1, c2, c3 : in    std_logic;
    l0, l1, f0, f1                            : in    std_logic;
    D                                         : inout std_logic_vector(3 downto 0);
    d_gosync, d_senssync, d_resetsync, d_latchreset, d_wreqlatch,
    d_starttimer, d_resettimer, d_longclk, d_expire: out std_logic;
    countout, stateout                        : out std_logic_vector(3 downto 0);
    a0, a1, n_weout, mr, my, mg, sr, sy, sg : out   std_logic);

end top;

architecture behavioral of top is

  component synchronizer
    port (
      clk, syncin : in  std_logic;
      syncout     : out std_logic);
  end component;

  component divider
    port (
      clk, rst : in  std_logic;
      longclk  : out std_logic);
  end component;

  component fsm
    port (
      reset, l0, l1, f0, f1                      : in  std_logic;
      gosync, senssync, wreqlatch, expire, clk   : in  std_logic;
      mr, my, mg, sr, sy, sg                     : out std_logic;
      stateout                           : out std_logic_vector (3 downto 0);
      a0, a1, n_we, latchreset, starttimer, resettimer : out std_logic);
  end component;

  component wrlatch
    port (
      latchset, latchreset : in  std_logic;
```

```
      latchout             : out std_logic);
  end component;

  component leveltopulse
    port (
      level : in  std_logic;
      clk   : in  std_logic;
      pulse : out std_logic);
  end component;

  component timer
    port (
      D                        : in  std_logic_vector(3 downto 0);
      clk, longclk, starttimer : in  std_logic;
      resettimer               : in  std_logic;
      countout                 : out std_logic_vector(3 downto 0);
      expired                  : out std_logic);
  end component;

  component tribuffer
    port (
      input  : in  std_logic;
      enable : in  std_logic;
      output : out std_logic);
  end component;

  signal gosync, senssync, resetsync, latchreset, wreqlatch : std_logic;
  signal starttimer, resettimer, longclk, expire : std_logic;
  signal n_we, we, tbe : std_logic;
begin  -- behavioral

  GOLTP : leveltopulse port map (
    level => go,
    clk   => clk,
    pulse => gosync);

  SENSSY : synchronizer port map (
    clk     => clk,
    syncin  => sens,
    syncout => senssync);

  RESETLTP : leveltopulse port map (
    level => reset,
    clk   => clk,
    pulse => resetsync);

  WLATCH : wrlatch port map (
    latchset   => wreq,
    latchreset => latchreset,
    latchout   => wreqlatch);
```

```
DIV : divider port map (
  clk     => clk,
  rst   => starttimer,
  longclk => longclk);

TIM : timer port map (
  D          => D,
  clk        => clk,
  longclk    => longclk,
  starttimer => starttimer,
  resettimer => resettimer,
  countout   => countout,
  expired    => expire);

 STATEMACHINE : fsm port map (
   reset      => resetsync,
   l0         => l0,
   l1         => l1,
   f0         => f0,
   f1         => f1,
   gosync     => gosync,
   senssync   => senssync,
   wreqlatch  => wreqlatch,
   expire     => expire,
   clk        => clk,
   mr         => mr,
   my         => my,
   mg         => mg,
   sr         => sr,
   sy         => sy,
   sg         => sg,
   a0         => a0,
   a1         => a1,
   n_we       => n_we,
   stateout   => stateout,
   latchreset => latchreset,
   starttimer => starttimer,
   resettimer => resettimer);

T0 : tribuffer port map (
  input  => C0,
  output => D(0),
  enable => tbe);
T1 : tribuffer port map (
  input  => C1,
  output => D(1),
  enable => tbe);
T2 : tribuffer port map (
  input  => C2,
  output => D(2),
  enable => tbe);
```

```
  T3 : tribuffer port map (
    input  => C3,
    output => D(3),
    enable => tbe);

  we <= not n_we;
  n_weout <= n_we;
  tbe <= we and not clk;


  d_gosync <= gosync;
  d_senssync <= senssync;
  d_resetsync <= resetsync;
  d_latchreset <= latchreset;
  d_wreqlatch <= wreqlatch;
  d_starttimer <= starttimer;
  d_resettimer <= resettimer;
  d_longclk <= longclk;
  d_expire <= expire;

end behavioral;
```

A.2 divider.vhd

```
-- divider.vhd: 1.8432 MHz to 1 Hz clock divider
-- Dan R. K. Ports   <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity divider is

  port (
    clk, rst : in  std_logic;
    longclk  : out std_logic);

end divider;

architecture behavioral of divider is

  signal longclkstate : std_logic := '0';
  signal count : std_logic_vector(24 downto 0);

begin  -- behavioral

  -- purpose: generate output pulse when counter reaches 1000000
  -- type   : sequential
  -- inputs : clk, rst
  -- outputs: longclk
  process (clk, rst)
  begin  -- process
    if rst = '1' then                   -- asynchronous reset (active high)
      count <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      count <= count + 1;
    if count = 1843199 then
      longclkstate <= '1';
    elsif count = 1843200 then
      count <= (others => '0');
      longclkstate <= '0';
    end if;
    end if;


  end process;

  longclk <= longclkstate;

end behavioral;
```

A.3 fsm.vhd

```
-- fsm.vhd: finite state machine controller for traffic signal
-- Dan R. K. Ports    <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity fsm is

  port (
    reset, l0, l1, f0, f1                       : in  std_logic;
    gosync, senssync, wreqlatch, expire, clk : in  std_logic;
    mr, my, mg, sr, sy, sg                    : out  std_logic;  -- light outputs
    stateout                                  : out std_logic_vector (3 downto 0);
    a0, a1, n_we, latchreset, starttimer, resettimer     : out std_logic );

end fsm;

architecture behavioral of fsm is

  type StateType is (s_idle, s_read, s_write, s_maing1, s_maing2,
                     s_mainy, s_walk, s_sideg, s_sideext, s_sidey,
                     s_blinkon, s_blinkoff);

  signal curState, nextState : StateType := s_idle;
  signal nexta0, nexta1 : std_logic := '0';

begin  -- behavioral

  -- purpose: determine next state and timer address bits
  -- type   : combinational
  -- inputs : curState, gosync, senssync, wreqlatch, expire, f0, f1
  -- outputs: nextState, nexta0, nexta1
  fsm: process (curState, gosync, senssync, wreqlatch, expire, f0, f1)
  begin  -- process fsm

    if reset = '1' then
      nextState <= s_idle;
    else
      if gosync = '1' then
        if (f0 = '0') and (f1 = '0') then
          nextState <= s_read;
          nexta0 <= l0;
          nexta1 <= l1;
        elsif (f0 = '1') and (f1 = '0') then
          nextState <= s_write;
          nexta0 <= l0;
          nexta1 <= l1;
```

```
    elsif (f0 = '0') and (f1 = '1') then
      nextState <= s_maing1;
      nexta0 <= '1';
      nexta1 <= '0';
    else
      nextState <= s_blinkon;
      nexta0 <= '1';
      nexta1 <= '1';
    end if;
  else
    case curState is
      when s_idle => nextState <= s_idle;
                     nexta0 <= '0';
                     nexta1 <= '0';
      when s_read => nextState <= s_read;
                     nexta0 <= l0;
                     nexta1 <= l1;
      when s_write => nextState <= s_read;
      when s_maing1 => nextState <= s_maing2;
                       nexta0 <= '0';
                       nexta1 <= '1';
      when s_maing2 => nextState <= s_mainy;
                       nexta0 <= '0';
                       nexta1 <= '0';
      when s_mainy => if wreqlatch = '1' then
         nextState <= s_walk;
         nexta1 <= '1';
         nexta0 <= '0';
       else
         nextState <= s_sideg;
         nexta1 <= '0';
         nexta0 <= '1';
       end if;
      when s_walk => nextState <= s_sideg;
                     nexta1 <= '0';
                     nexta0 <= '1';
      when s_sideg => if senssync = '1' then
         nextState <= s_sideext;
         nexta1 <= '0';
         nexta0 <= '1';
       else
         nextState <= s_sidey;
         nexta1 <= '0';
         nexta0 <= '0';
       end if;
      when s_sideext => nextState <= s_sidey;
                        nexta1 <= '0';
                        nexta0 <= '0';
      when s_sidey => nextState <= s_maing1;
                      nexta1 <= '0';
                      nexta0 <= '1';
```

```
        when s_blinkon => nextState <= s_blinkoff;
                          nexta1 <= '1';
                          nexta0 <= '1';
        when s_blinkoff => nextState <= s_blinkon;
                           nexta1 <= '1';
                           nexta0 <= '1';
        when others => nextState <= s_blinkon;
                       nexta1 <= '1';
                       nexta0 <= '1';
      end case;
    end if;
  end if;

end process fsm;

-- purpose: transition states on rising edge and start timer when expired
-- type   : combinational
-- inputs : clk, exp
-- outputs: curState, starttimer
state_clocked: process (clk)
begin  -- process state_clocked
  if rising_edge(clk) then

    if expire = '1' or gosync = '1' or reset = '1' or curState = s_idle
                or curState = s_read or curstate = s_write then
      curState <= nextState;
      a0 <= nexta0;
      a1 <= nexta1;

      if nextState = s_idle or nextState = s_read or nextState = s_write then
        resettimer <= '1';
        starttimer <= '0';
      else
        starttimer <= '1';
        resettimer <= '0';
      end if;
    else
      starttimer <= '0';
    end if;

  end if;
end process state_clocked;


-- purpose: set outputs for current state
-- type   : combinational
-- inputs : curState
-- outputs: mr, my, mg, sr, sy, sg, n_we, latchreset
set_outputs: process (curState)
begin  -- process set_outputs
  case curState is
```

```
when s_idle => mr <= '0'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '1';
               stateout <= "0000";
when s_read => mr <= '0'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '1';
               stateout <= "0001";
when s_write => mr <= '0'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '0';
               n_we <= '0'; latchreset <= '1';
               stateout <= "0010";
when s_maing1 => mr <= '0'; my <= '0'; mg <= '1';
               sr <= '1'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "0011";
when s_maing2 => mr <= '0'; my <= '0'; mg <= '1';
               sr <= '1'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "0100";
when s_mainy => mr <= '0'; my <= '1'; mg <= '0';
               sr <= '1'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "0101";
when s_walk => mr <= '1'; my <= '1'; mg <= '0';
               sr <= '1'; sy <= '1'; sg <= '0';
               n_we <= '1'; latchreset <= '1';
               stateout <= "0110";
when s_sideg => mr <= '1'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '1';
               n_we <= '1'; latchreset <= '0';
               stateout <= "0111";
when s_sideext => mr <= '1'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '1';
               n_we <= '1'; latchreset <= '0';
               stateout <= "1000";
when s_sidey => mr <= '1'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '1'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "1001";
when s_blinkon => mr <= '0'; my <= '1'; mg <= '0';
               sr <= '1'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "1010";
when s_blinkoff => mr <= '0'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
               stateout <= "1011";
when others => mr <= '0'; my <= '0'; mg <= '0';
               sr <= '0'; sy <= '0'; sg <= '0';
               n_we <= '1'; latchreset <= '0';
```

```
                     stateout <= "1100";
    end case;
  end process set_outputs;

end behavioral;
```

A.4 leveltopulse.vhd

```
-- leveltopulse.vhd: level to pulse unit
-- Dan R. K. Ports    <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;

entity leveltopulse is

  port (
    level : in  std_logic;
    clk   : in  std_logic;
    pulse : out std_logic);

end leveltopulse;

architecture behavioral of leveltopulse is

  signal x, y, z : std_logic := '0';

begin  -- behavioral

  process (clk)
  begin  -- process
    if rising_edge(clk) then
      x <= level;
      y <= x;
      z <= y;
    end if;
  end process;

  pulse <= (y and (not z));

end behavioral;
```

A.5 synchronizer.vhd

```
-- synchronizer.vhd: 1-input synchronizer
-- Dan R. K. Ports   <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  port (
    clk, syncin : in  std_logic;
    syncout     : out std_logic);
end synchronizer;

architecture synchronizer of synchronizer is

begin  -- synchronizer

  -- purpose: synchronize output on rising edge of clock
  -- type   : combinational
  -- inputs : clk
  -- outputs: syncout
  sync: process (clk)
  begin  -- process sync
    if rising_edge(clk) then
      syncout <= syncin;
    end if;
  end process sync;

end synchronizer;
```

A.6 timer.vhd

```
-- timer.vhd: programmable count-down timer
-- Dan R. K. Ports   <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity timer is

  port (
    D                         : in  std_logic_vector(3 downto 0);
    clk, longclk, starttimer : in  std_logic;
    resettimer                : in  std_logic;
    countout                  : out std_logic_vector(3 downto 0);
    expired                   : out std_logic);

end timer;

architecture behavioral of timer is

  signal count : std_logic_vector(3 downto 0);
  signal countzero : std_logic := '1';
  signal counting : std_logic := '0';

begin  -- behavioral

  -- purpose: count down on clock pulses
  -- type   : sequential
  -- inputs : clk, starttimer, longclk
  -- outputs: expired
  process (clk, starttimer)
  begin  -- process

    if rising_edge(clk) then
      if resettimer = '1' then
        count <= D;
        counting <= '0';
        countzero <= '0';
      elsif starttimer = '1' then
        count <= D;
        counting <= '1';
        countzero <= '0';
      elsif count = 0 then
        if countzero = '1' then
          counting <= '0';
  countzero <= '1';
        else
          countzero <= '1';
```

```
        end if;
      elsif longclk = '1' then
        count <= count - 1;
      else
count <= count;
      end if;
    end if;

  end process;

  expired <= countzero and counting;
  countout <= count;

end behavioral;
```

A.7 tribuffer.vhd

```
-- tribuffer.vhd: one-signal tristate buffer
-- Dan R. K. Ports    <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;

entity tribuffer is

  port (
    input  : in    std_logic;
    enable : in    std_logic;
    output : out   std_logic);

end tribuffer;

architecture behavioral of tribuffer is

begin  -- behavioral

  -- purpose: output input when enabled
  -- type   : combinational
  -- inputs : enable, input
  -- outputs: output
  process (enable, input)
  begin  -- process

    if enable = '1' then
      output <= input;
    else
      output <= 'Z';
    end if;

  end process;


end behavioral;
```

A.8 wrlatch.vhd

```
-- wrlatch.vhd: asynchronous SR latch for walk request
-- Dan R. K. Ports   <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01

library ieee;
use ieee.std_logic_1164.all;

entity wrlatch is

  port (
    latchset, latchreset : in  std_logic;
    latchout             : out std_logic);

end wrlatch;

architecture behavioral of wrlatch is
  signal latchreg : std_logic := '0';
begin  -- behavioral

  latchreg <= (latchreg or latchset) and not (latchreset);
  latchout <= latchreg;

end behavioral;
```