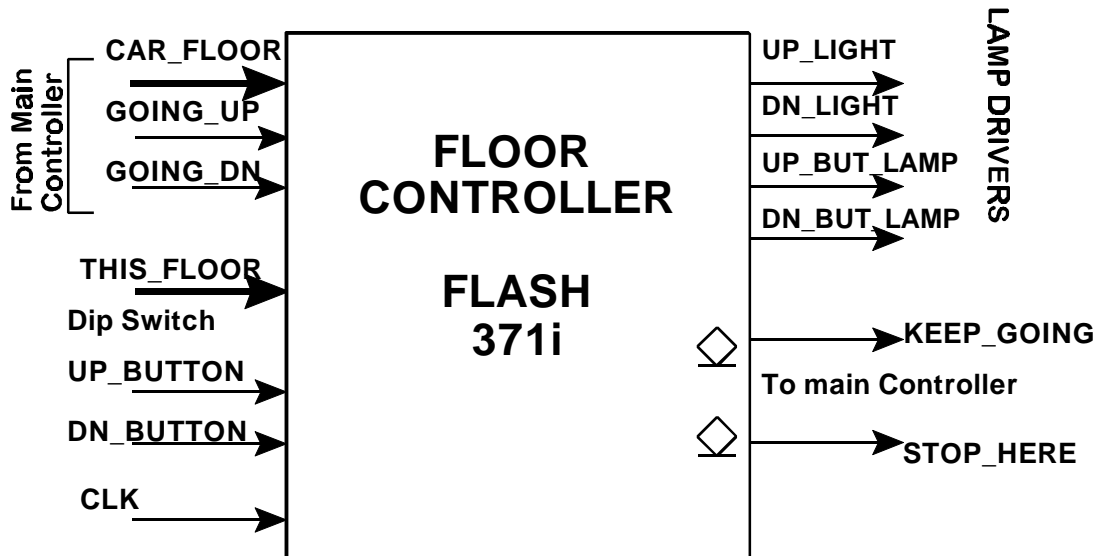


ELEVATOR PROBLEM FLOOR CONTROLLER



The inputs to the controller come from three sources:

- The Main Controller (CAR_FLOOR, GOING_UP, and GOING_DN)
- A six pole DIP SWITCH (THIS_FLOOR) with pullup resistors
- The two push buttons (UP_BUTTON, DN_BUTTON)

The outputs go to two places:

- The four lamps on the floor (UP_LIGHT, DOWN_LIGHT, UP_BUT_LAMP and DN_BUT_LSMP)
- Two open-collector outputs to the Main Controller (KEEP_GOING, STOP_HERE).

The interpretation of the two outputs is as follows:

KEEP_GOING	STOP_HERE	INTERPRETATION
0	0	Stop and pick up passenger(s) wishing to go in the opposite direction.
0	1	Stop here and let a passenger of. Then reverse directions (see note)
1	0	Keep on going to the next floor

1	1	Stop and either left of passenger(s) or take on passenger(s) going in the same direction. Then continue on.
Note: When the direction changes in the 1 case, the status signals may change as a result. Either the car will start down(10) or stop and remain at the floor (00).		

KEEP_GOING should be asserted (low) if:

- 1: A button on a high floor (going up) or a lower floor (going down) has been pressed.
- 2: A button in the car for a higher floor (going up) or for a lower floor (going down) has been pressed.

STOP_HERE should be asserted (low) if:

- 1: The UP button on this floor has been pressed (going up) or the DN button has been pressed (going down).
- 2: The button for this floor in the car has been pressed.

The UP_LIGHT and the DN_LIGHT simply display the state of the GOING_UP and GOING_DOWN signals.

The UP_BUT_LAMP should be turned on when the UP button is pressed and remain on until the elevator stops at the floor while going up. A similar situation exists for the DN_BUT_LAMP.

One might attempt to design this is a state machine but it isn't really a state machine. It is just as easy to write the VHDL by hand. The VHDL description for this is shown below.

```

library ieee;
use ieee.std_logic_1164.all;

entity floor_control is port(
  car_floor:          in    std_logic_vector(4 downto 0);
  this_floor:        in    std_logic_vector(4 downto 0);
  clk:               in    std_logic;
  going_up, going_dn: in    std_logic;
  up_button, dn_button: in    std_logic;
  up_but_lamp, dn_but_lamp: buffer std_logic;
  up_light, dn_light: out    std_logic;
  keep_going, stop_here: out    std_logic);
  end floor_control;

architecture behavioral of floor_control is
  signal higher, same, lower: std_logic;
  -- these are outputs of a comparator which compares this_floor
  -- with car_floor. higher means the car is higher than the floor.
  signal compare: std_logic_vector(4 downto 0);
  -- compare is used in the implementation of the comparator
  attribute synthesis_off of higher, same, lower: signal is true;
  attribute synthesis_off of compare: signal is true;
  -- All of the above are "unsynthesized" to minimize CPLD area.
  signal up_but_sync, dn_but_sync: std_logic;

```

```

-- These signals are used to synchronize the button inputs.
begin
-- First we will sample the push buttons
buttons: process(clk, up_button, dn_button)
begin
    if (clk'event and clk='1') then
        up_but_sync<=up_button;
        dn_but_sync<=dn_button;
    end if;
end process;
-- now we will turn the button lamps on
-- pressing the button turns the lamp on unless it should be off
-- It is turned off when the car reaches the floor and is going in
-- the same direction as the button.
lampbutn: process(clk, up_but_sync, dn_but_sync)
begin
    if (clk'event and clk='1') then
        if (up_but_sync='1')
            then up_but_lamp<='1';
            end if;
        if (same='1') and (going_up='1')
            then up_but_lamp<='0';
            end if;
        if (dn_but_sync='1')
            then dn_but_lamp<='1';
            end if;
        if (same='1') and (going_dn='1')
            then dn_but_lamp<='0';
            end if;
    end if;
end process;
-- the up and down lights are connected directly to the up and down status
up_light<=going_up;
dn_light<=going_dn;
-- now we will do the floor comparison. it would have been easier to
-- use three if statements (if this>car, if this=car, and if this<car)
-- but a tremendous amount of chip area would be used because of the
-- flat sop design which would result.
-- instead the compare bits are set if the car and this bits are
-- different. Then these bits are used to make the final comparison.
compare: process(this_floor, car_floor, compare)
begin
    same<='0'; higher<='0'; lower<='0';
    for i in 4 downto 0 loop
        compare(i)<=this_floor(i) xor car_floor(i);
    end loop;
    if compare(4)='0' then
        if this_floor(4)='1' then lower<='1'; else higher<='1'; end if;
    elsif compare(3)='0' then
        if this_floor(3)='1' then lower<='1'; else higher<='1'; end if;
    elsif compare(2)='0' then
        if this_floor(2)='1' then lower<='1'; else higher<='1'; end if;
    elsif compare(1)='0' then
        if this_floor(1)='1' then lower<='1'; else higher<='1'; end if;
    elsif compare(0)='0' then
        if this_floor(0)='1' then lower<='1'; else higher<='1'; end if;
    else same<='1'; end if;
end process;

-- finally we get to the two control signals. These must be synthesized
-- is a strange way. We have to use three-state outputs to look like
-- open collector outputs. This is accomplished by setting the output
-- permanently to 0 and then using the enable control to let it float
-- to the high state.

```

```

controls: process(same, higher, lower, up_but_lamp, dn_but_lamp, going_up, going_dn)
begin
  if (((up_but_lamp='1') or (dn_but_lamp='1'))
      and (lower='1') and (going_up='1'))
  or (((up_but_lamp='1') or (dn_but_lamp='1'))
      and (higher='1') and (going_dn='1'))
  then keep_going<='0'; else keep_going<='Z'; end if;

  if ((up_but_lamp='1') and (same='1') and (going_up='1'))
  or ((dn_but_lamp='1') and (same='1') and (going_dn='1'))
  then stop_here<='0'; else stop_here<='Z'; end if;
end process;
end behavioral;

```

Now, some excerpts from the report file.

```

stop_here = GND
stop_here.OE = stop_here.OE.CMB =
  going_dn * same.CMB * dn_but_lamp.Q
+ going_up * up_but_lamp.Q * same.CMB

keep_going = GND
keep_going.OE = keep_going.OE.CMB =
  going_up * dn_but_lamp.Q * lower.CMB
+ going_up * up_but_lamp.Q * lower.CMB
+ going_dn * dn_but_lamp.Q * higher.CMB
+ going_dn * up_but_lamp.Q * higher.CMB

```

Note that two macrocells are needed for each of these because the output enables are only single product terms.

```

dn_light = going_dn
up_light = going_up

```

We could have just used two pieces of wire except this provides some buffering.

```

/dn_but_lamp.D = /dn_but_lamp.Q * /dn_button.QI
+ going_dn * same.CMB

/up_but_lamp.D = going_up * same.CMB
+ /up_but_lamp.Q * /up_button.QI

```

Note that negative logic was used for these as it resulted in less logic. The clock and other inputs have not been shown.

```

lower =
  compare_4.CMB * compare_3.CMB * compare_2.CMB * compare_1.CMB *
  /compare_0.CMB * this_floor_0
+ compare_4.CMB * compare_3.CMB * compare_2.CMB * /compare_1.CMB *
  this_floor_1
+ compare_4.CMB * compare_3.CMB * /compare_2.CMB * this_floor_2
+ compare_4.CMB * /compare_3.CMB * this_floor_3
+ /compare_4.CMB * this_floor_4

```

```

higher =

```

```

compare_4.CMB * compare_3.CMB * compare_2.CMB * compare_1.CMB *
/compare_0.CMB * /this_floor_0
+ compare_4.CMB * compare_3.CMB * compare_2.CMB * /compare_1.CMB *
/this_floor_1
+ compare_4.CMB * compare_3.CMB * /compare_2.CMB * /this_floor_2
+ compare_4.CMB * /compare_3.CMB * /this_floor_3
+ /compare_4.CMB * /this_floor_4

```

Note that 5 product terms were needed for each of these, a total of ten terms. See below.

```

same =
compare_4.CMB * compare_3.CMB * compare_2.CMB *
compare_1.CMB * compare_0.CMB

```

```

compare_0 =
this_floor_0 * /car_floor_0
+ /this_floor_0 * car_floor_0

```

```

compare_1 =
this_floor_1 * /car_floor_1
+ /this_floor_1 * car_floor_1

```

```

compare_2 =
this_floor_2 * /car_floor_2
+ /this_floor_2 * car_floor_2

```

```

compare_3 =
this_floor_3 * /car_floor_3
+ /this_floor_3 * car_floor_3

```

```

compare_4 =
this_floor_4 * /car_floor_4
+ /this_floor_4 * car_floor_4

```

The five compare bits each take two product terms. The equal term requires only one. Hence the comparator requires a total of 21 product terms and 8 macrocells. Were the simple version of the description used, many many more product terms would have been needed!

```

dn_button.CI = clk
up_button.CI = clk

```

The fitter recognized that the dedicated input pins have synchronizing flip-flops available and thus made use of them. No output macrocells were used.

The pinout is as follows.

1	:	GND	7	:	Not Used
2	:	keep_going	8	:	Not Used
3	:	Not Used	9	:	Not Used
4	:	Not Used	10	:	dn_button
5	:	Not Used	11	:	VPP
6	:	Not Used	12	:	GND
			13	:	clk
			14	:	Not Used
			15	:	going_up

16	:	this_floor_4	31	:	dn_but_lamp
17	:	this_floor_2	32	:	up_button
18	:	this_floor_0	33	:	car_floor_4
19	:	car_floor_1	34	:	GND
20	:	car_floor_0	35	:	car_floor_3
21	:	stop_here	36	:	(keep_going_0E)
22	:	VCC			
23	:	GND	37	:	up_but_lamp
24	:	(higher)	38	:	(compare_0)
25	:	this_floor_1	39	:	(compare_2)
26	:	(same)	40	:	(compare_4)going_dn
27	:	dn_light	41	:	up_light
28	:	(compare_3)	42	:	car_floor_2
29	:	(compare_1)	43	:	(lower)this_floor_3
30	:	(stop_here_0E)	44	:	VCC

The device utilization is as follows:

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	3	3
Clock/Inputs	2	2
I/O Macrocells	24	32
29 /		37 = 78 %

	Required	Max (Available)
CLOCK/LATCH ENABLE signals	1	2
Input REG/LATCH signals	2	4
Input PIN signals	2	2
Input PINs using I/O cells	8	8
Output PIN signals	16	24
Total PIN signals	29	37
Macrocells Used	16	32
Unique Product Terms	34	160

At the speed at which an elevator moves, the worst case delay path is hardly a concern.