

Toll Booth Spec.

The toll booth has 2 external sensors C and I. C indicates whether a car is in the booth(car = C = 1, lcar = C = 0). I indicates whether a coin has been placed in the collection basket (no coin = I = 00, 5p = I = 01, 10p = I = 10, 20p = I = 11), no other coins are valid.

There are two output lights and an output alarm. When a car pulls in, a red light is illuminated. It remains lit until at least 35p has been deposited when the red light goes out and the green light is lit. The green light is lit until the car leaves the booth. If a car exits without paying the full toll then the red light remains lit and the alarm is activated. The alarm remains activated until another car enters the booth.

A possible list of states is

State	Condition	R	G	A
S _{nocar}	no car in booth	1	0	0
S ₀	Car in booth, 0p paid	1	0	0
S ₅	Car in booth, 5p paid	1	0	0
S ₁₀	Car in booth, 10p paid	1	0	0
S ₁₅	Car in booth, 15p paid	1	0	0
S ₂₀	Car in booth, 20p paid	1	0	0
S ₂₅	Car in booth, 25p paid	1	0	0
S ₃₀	Car in booth, 30p paid	1	0	0
S _{paid}	Car in booth, toll paid	0	1	0
S _{cheat}	Car left booth, toll not paid	1	0	1

An arbitrary list of state values

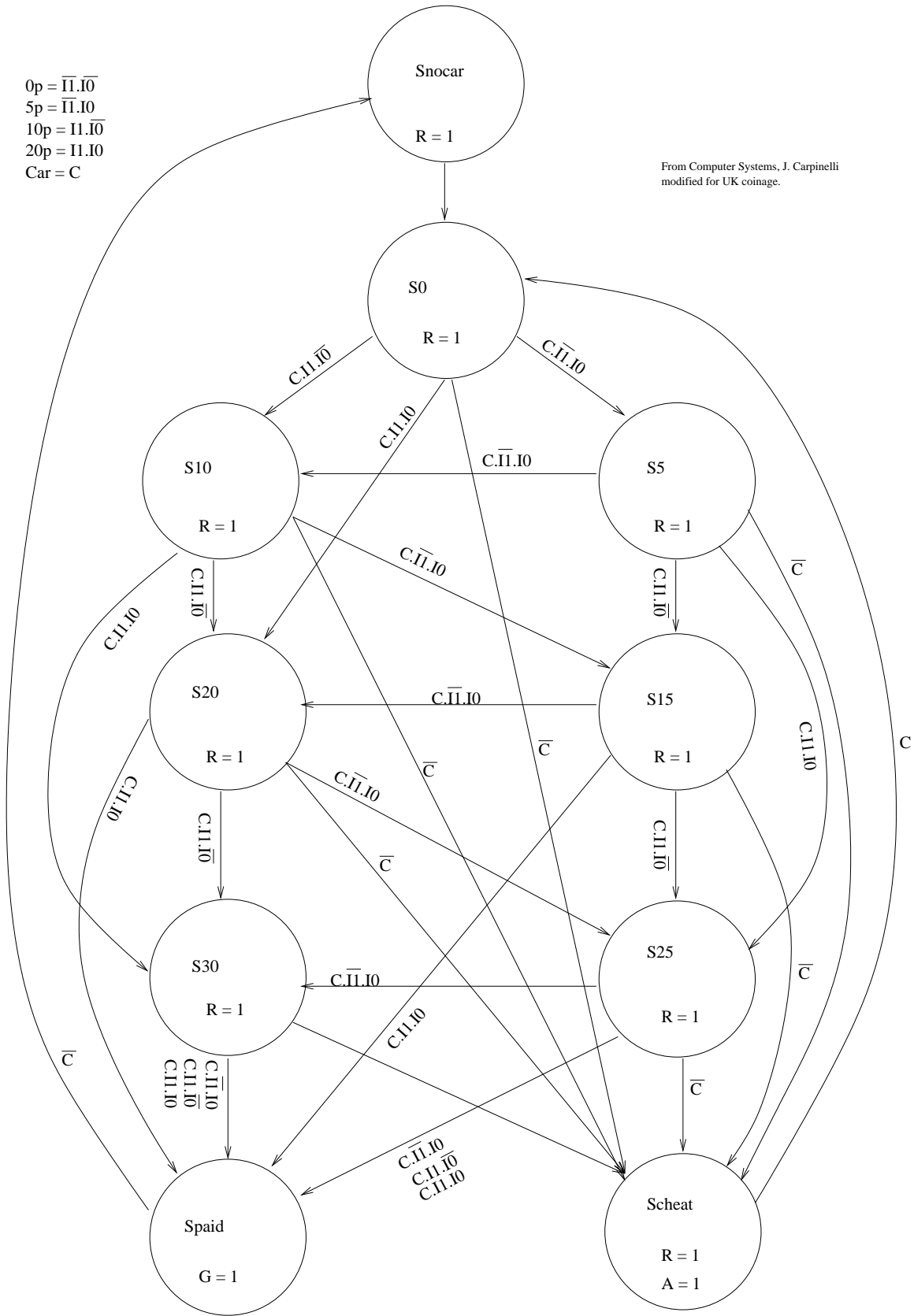
State	Value
S _{nocar}	0000
S ₀	1000
S ₅	1001
S ₁₀	1011
S ₁₅	1010
S ₂₀	1110
S ₂₅	1111
S ₃₀	1101
S _{paid}	1100
S _{cheat}	0100

and transitions

Present State	C	I ₁ I ₀	Next State	R	G	A
S _{nocar}	1	XX	S ₀	1	0	0
S _{paid}	0	XX	S _{nocar}	1	0	0
S _{cheat}	1	XX	S ₀	1	0	0
S ₀	0	XX	S _{cheat}	1	0	1
S ₀	1	01	S ₅	1	0	0
S ₀	1	10	S ₁₀	1	0	0
S ₀	1	11	S ₂₀	1	0	0
S ₅	0	XX	S _{cheat}	1	0	1
S ₅	1	01	S ₁₀	1	0	0
S ₅	1	10	S ₁₅	1	0	0
S ₅	1	11	S ₂₅	1	0	0
S ₁₀	0	XX	S _{cheat}	1	0	0
S ₁₀	1	01	S ₁₅	1	0	0
S ₁₀	1	10	S ₂₀	1	0	0
S ₁₀	1	11	S ₃₀	1	0	0
S ₁₅	0	XX	S _{cheat}	1	0	1
S ₁₅	1	01	S ₂₀	1	0	0
S ₁₅	1	10	S ₂₅	1	0	0
S ₁₅	1	11	S _{paid}	0	1	0
S ₂₀	0	XX	S _{cheat}	1	0	1
S ₂₀	1	01	S ₂₅	1	0	0
S ₂₀	1	10	S ₃₀	1	0	0
S ₂₀	1	11	S _{paid}	0	1	0
S ₂₅	0	XX	S _{cheat}	1	0	1
S ₂₅	1	01	S ₃₀	1	0	0
S ₂₅	1	10	S _{paid}	0	1	0
S ₂₅	1	11	S _{paid}	0	1	0
S ₃₀	0	XX	S _{cheat}	1	0	1
S ₃₀	1	01	S _{paid}	0	1	0
S ₃₀	1	10	S _{paid}	0	1	0
S ₃₀	1	11	S _{paid}	0	1	0

$0p = \bar{1}1.\bar{1}0$
 $5p = \bar{1}1.10$
 $10p = 11.\bar{1}0$
 $20p = 11.10$
 $Car = C$

From Computer Systems, J. Carpinelli
modified for UK coinage.



Values have to be assigned to each state and there are many ways of coding or mapping a state to a value. A good general rule is to give the initial, or starting state the value zero as this may simplify initialisation of the hardware (be sure that you understand why.), except when using one-hot encoding. See the link given on the main page for a detailed discussion of encodings.

Hand crafting an encoding can be time consuming for anything other than simple designs. For designs that cycle through a fixed sequence of states, eg traffic light controllers, a Gray code will provide a minimum transition distance between states. Although this may not be the most efficient in terms of the logic used it is simple. However there is no consideration of the input/output values in determining the next state when using this technique. The list of state values shown earlier is a modified Gray code and the example is worked using this coding.

An alternative approach is to use the following guidelines when assigning state values. These are based on next state and input/output values and are taken from the link referred to above. Worked examples of these techniques can be found at the same place (<http://www.redbrick.dcu.ie/academic/CLD/chapter9/chapter09.doc3.html>)

Highest Priority

States that have the same next state for a given input transition should be given adjacent assignments in the state map.

Medium Priority

States that have a common ancestor should be given adjacent assignments in the state map.

Lowest Priority

States with the same output for a given input should be given given adjacent assignments in the state map.

The state map is like a Karnaugh map with states occupying cells within the map, the whole map having sufficient rows and columns to represent all possible states.

It is not always possible to adhere to these guidelines when placing states onto the state map, eg applying them to the tollbooth example could provide the following sets:

• Highest priority

Most of the states share the state S_{cheat} as a common destination when the input $C = 0$. The list is

$$S_0, S_5, S_{10}, S_{15}, S_{20}, S_{25}, S_{30}$$

It is not possible to assign all of these states adjacent to each other so a compromise will have to be made as to which states should be placed adjacent. One could consider human nature and decide that drivers will either cheat immediately, or if they have nearly enough change, or modify the placement with respect to the medium priority list. *i.e.* states that share a common destination and a common ancestor should be placed adjacent to each other. It could also be argued that the likelihood of a driver cheating is much lower than that of a driver paying a toll so the focus should be on the medium priority groupings as they are more likely to occur.

• Medium Priority

There are many groups of states that share a common ancestor, namely

$$(S_0 \rightarrow S_5, S_{10}, S_{20}, S_{cheat}), (S_5 \rightarrow S_{10}, S_{15}, S_{25}, S_{cheat}), (S_{10} \rightarrow S_{15}, S_{20}, S_{30}, S_{cheat}), (S_{15} \rightarrow S_{20}, S_{25}, S_{paid}, S_{cheat}), (S_{20} \rightarrow S_{25}, S_{30}, S_{paid}, S_{cheat}), (S_{25} \rightarrow S_{30}, S_{paid}, S_{cheat}), (S_{30} \rightarrow S_{paid}, S_{cheat})$$

Low Priority

All states except S_{paid} set the red lamp on.

These lists are then used to develop a mapping which minimises the the bit changes needed to transition from one state to another state as well as considering the inputs and outputs of the system.


```

)
)
or (p3 and p2 and not c and ((p1 and not p0)
                             or(p0)
                             )
)
or (p3 and p2 and c)
or (not p3 and p2 and not c and not p1 and not p0)
);

n1 <= reset and ((p3 and not p2 and c and ((not p1 and i(1))
                                           or(i(1) and not i(0))
                                           or(not p1 and p0 and i(0))
                                           or (p1 and not i(1))
                                           )
)
or (p3 and p2 and c and ((p1 and not i(1) and not i(0))
                          or (p1 and not p0 and not i(0))
                          )
)
);

n0 <= reset and ((p3 and not p2 and c and ((p0 and not i(1) and not i(0))
                                           or(not p1 and not p0 and i(0))
                                           or(p0 and i(1) and i(0))
                                           or(not p0 and i(1) and not i(0))
                                           )
)
or (p3 and p2 and c and ((p0 and not i(1) and not i(0))
                          or(p1 and not i(1) and i(0))
                          or (p1 and not p0 and i(1) and not i(0))
                          )
)
);

p3 <= current_state(3);
p2 <= current_state(2);
p1 <= current_state(1);
p0 <= current_state(0);

-- purpose: set new current state
update0 : block ((not (clk'stable) and clk) = '1')
begin -- block update0
    current_state(0) <= guarded n0;
end block update0;

-- purpose: new current state
update1 : block ((not (clk'stable) and clk) = '1')
begin -- block update1
    current_state(1) <= guarded n1;
end block update1;

-- purpose: new current state
update2 : block ((not (clk'stable) and clk) = '1')
```

```
begin -- block update2
  current_state(2) <= guarded n2;
end block update2;

-- purpose: new current state
update3 : block ((not (clk'stable) and clk) = '1')
begin -- block update3
  current_state(3) <= guarded n3;
end block update3;

-- purpose: latch red light
redlight : block ((not (clk'stable) and clk) = '1')
begin -- block redlight
  regout_2 <= guarded not(p3 and p2 and not p1 and not p0);
end block redlight;

-- purpose: latch green light
greenlight : block ((not (clk'stable) and clk) = '1')
begin -- block greenlight
  regout_1 <= guarded (p3 and p2 and not p1 and not p0);
end block greenlight;

-- purpose: latch alarm
alarm : block ((not (clk'stable) and clk) = '1')
begin -- block alarm
  regout_0 <= guarded (not p3 and p2 and not p1 and not p0);
end block alarm;

r <= regout_2;
g <= regout_1;
a <= regout_0;
end dataflow;
```

This model can be synthesised and compared to the following state-machine model for both area and performance. Use `syf` to convert the `.fsm` model to `.vbe` and `boog` to create the structural model. Compare the results of the different state encoding algorithms provide by the state machine synthesis tool. The algorithms all generate different results for the structural model.

```
-----  
-- Title       : Toll Booth second model  
-- Project     :  
-----  
-- File        : tollbooth.fsm  
-- Author      : <ngunton@ptolome.cems.uwe.ac.uk>  
-- Company     : FoCEMS, UWE  
-- Last update : 2002/12/04  
-- Platform    : Alliance 5.0 on GNU/Linux  
-----  
-- Description : state machine description. Based  
-- on the model given by Carpinelli and modified for UK coinage  
-----  
-- Revisions  :  
-- Date       Version Author Description  
-- 2002/11/27 1.0      ngunton Created  
-----
```

entity tollbooth is

```
port (  
  c    : in  bit;           -- car in booth  
  i    : in  bit_vector(1 downto 0); -- coins. 01 5p, 10 10p, 11 20p  
  clk  : in  bit;  
  r    : out bit;           -- red light  
  g    : out bit;           -- green light  
  a    : out bit;           -- alarm  
  vdd  : in  bit;  
  vss  : in  bit;  
  reset : in  bit);
```

end tollbooth;

architecture machine of tollbooth is

```
  type state_type is(Snocar, S0, S5, S10, S15, S20, S25, S30, Spaid, Scheat);
```

```
-- Pragma CLOCK clk  
-- Pragma CURRENT_STATE current_state  
-- Pragma NEXT_STATE next_state
```

```
  signal current_state, next_state : state_type;  
begin
```

```
  -- purpose: state comb  
  -- type    : combinational  
  -- inputs  : c, r, i  
  -- outputs : r, g, a  
  comb      : process (c, reset, i, current_state)  
begin  -- process comb  
  if reset = '0' then  
    next_state <= Snocar;  
    r          <= '1';  
    g          <= '0';  
    a          <= '0';  
  else  
    case current_state is
```



```
else
    next_state    <= Scheat;
end if;
when S15 => r      <= '1';
          g      <= '0';
          a      <= '0';
if c = '1' then
    if i = B"01" then
        next_state <= S20;
    elsif i = B"10" then
        next_state <= S25;
    elsif i = B"11" then
        next_state <= Spaid;
    elsif i = B"00" then
        next_state <= S15;
    end if;
else
    next_state    <= Scheat;
end if;
when S20 => r      <= '1';
          g      <= '0';
          a      <= '0';
if c = '1' then
    if i = B"01" then
        next_state <= S25;
    elsif i = B"10" then
        next_state <= S30;
    elsif i = B"11" then
        next_state <= Spaid;
    elsif i = B"00" then
        next_state <= S20;
    end if;
else
    next_state    <= Scheat;
end if;
when S25 => r      <= '1';
          g      <= '0';
          a      <= '0';
if c = '1' then
    if i = B"01" then
        next_state <= S30;
    elsif i = B"10" then
        next_state <= Spaid;
    elsif i = B"11" then
        next_state <= Spaid;
    elsif i = B"00" then
        next_state <= S25;
    end if;
else
    next_state    <= Scheat;
end if;
when S30 => r      <= '1';
          g      <= '0';
          a      <= '0';
```

```
        if c = '1' then
            if (i = B"01" or i = B"10" or i = B"11") then
                next_state <= Spaid;
            elsif i = B"00" then
                next_state <= S0;
            end if;
        else
            next_state <= Scheat;
        end if;
    when Spaid => r <= '0';
                g <= '1';
                a <= '0';
                if c = '1' then
                    next_state <= Spaid;
                else
                    next_state <= Snocar;
                end if;
    when Scheat => r <= '1';
                g <= '0';
                a <= '1';
                if c = '0' then
                    next_state <= S0;
                else
                    next_state <= Scheat;
                end if;
    when others => r <= '1';
                g <= '0';
                a <= '0';
                next_state <= Snocar;
    end case;
end if;
end process comb;

-- purpose: update
-- type    : sequential
-- inputs  : clk
-- outputs:
ticker : process (clk)
begin -- process ticker
    if clk'event and clk = '1' then -- rising clock edge
        current_state <= next_state;
    end if;
end process ticker;
end machine;
```