Recap

Debugging
Intrusiveness
Monitors and dumps

Overview

Finite State Machines (Section 8.7)
State graph to assembly

Lab 4. Traffic Light Controller

This lab has these major objectives:
- The usage of linked list data structures;
- Create a segmented software system;
- an input-directed traffic light controller.

Limitations
- three switches
- six LEDs
- PP7 connected to red LED

See description of actual Lab 4 assignment

PT1=0, PT0=0 means no cars exist on either road
PT1=0, PT0=1 means there are cars on the East road
PT1=1, PT0=0 means there are cars on the North road
PT1=1, PT0=1 means there are cars on both roads

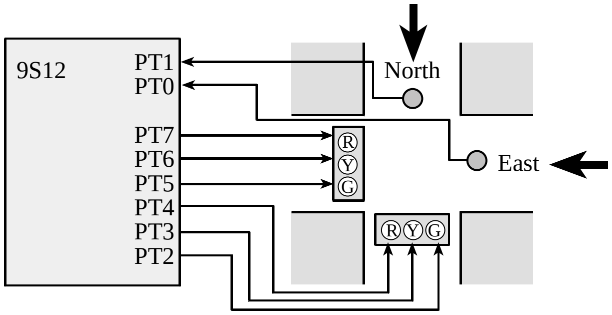


Figure 6.19. Traffic light interface.

goN, PT7-2 = 100001 makes it green on North and red on East
waitN, PT7-2 = 100010 makes it yellow on North and red on East
goE, PT7-2 = 001100 makes it red on North and green on East
waitE, PT7-2 = 010100 makes it red on North and yellow on East

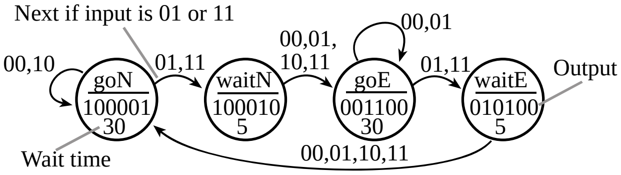


Figure 6.20. Graphical form of a Moore FSM that implements a traffic light.

State \ Input	00	01	10	11
goN (100001,30)	**goN**	**waitN**	**goN**	**waitN**
waitN (100010,5)	**goE**	**goE**	**goE**	**goE**
goE (001100,30)	**goE**	**goE**	**waitE**	**waitE**
waitE (010100,5)	**goN**	**goN**	**goN**	**goN**

Table 6.4. Tabular form of a Moore FSM that implements a traffic light.

```
      org $0800
Pt    rmb 2   ;state pointer
;Linked data structure
      org $4000 ;Put in ROM
OUT   equ 0    ;offset for output
WAIT  equ 1    ;offset for time
NEXT  equ 3    ;offset for next
goN   fcb $21  ;North green, East red
      fdb 3000 ;30sec
      fdb goN,waitN,goN,waitN
waitN fcb $22  ;North yellow, East red
      fdb 500  ;5sec
      fdb goE,goE,goE,goE
goE   fcb $0C  ;North red, East green
      fdb 3000 ;30 sec
      fdb goE,goE,waitE,waitE
waitE fcb $14  ;North red, East yellow
      fdb 500  ;5sec
      fdb goN,goN,goN,goN
main lds  #$4000    ;stack init
     bsr  Timer_Init ;enable TCNT
     ldaa #$FC      ;PT7-2 are lights
     staa DDRT      ;PT1-0 are sensors
     ldx  #goN      ;State pointer
     stx  Pt
FSM  ldx  Pt
     ldab OUT,x     ;Output value
     lslb
     lslb           ;line up with 7-2
     stab PTT       ;set lights
     ldy  WAIT,x    ;Time delay
     bsr  Timer_Wait10ms
     ldab PTT       ;Read input
     andb #$03      ;just bits 1,0
     lslb           ;2 bytes/address
     abx            ;add 0,2,4,6
     ldx  NEXT,x    ;Next state
     stx  Pt
     bra  FSM
     org  $FFFE
     fdb  main       ;reset vector
```

Program 6.22. Linked data structure implementation of the traffic light controller.

How do we prove to the judge it works?

Log all **`(input,time,output)`** *data* (like Lab 3*)*

Prove it works for a machine with a few states

then show the 1-1 mapping

Write in assembly

0) define the controller sequence **output, wait, input, next**

1) create the structure format (use **`equ`** *definitions)*

Where in memory should the state graph go?

How do we write assembly code to specify where?

How do we specify an arrow? (pointer or index)

2) show the 1-1 mapping from graph to assembly code

3) write assembly code

Start in the middle of the problem

How do we output?

How do we wait?

How do we input?

How do we go to next state?

What needs to be done once?

To add more complexity

(e.g., put a red/red state after each yellow state),

we simply increase the size of the **`fsm[]`** structure

define the **`Out`**, **`Time`**, and **`Next`** pointers

To add more output signals

(e.g., walk light),

use more of **`Out`** field.

could increase the precision of the **`Out`** field

To add two input lines

(e.g., walk button),

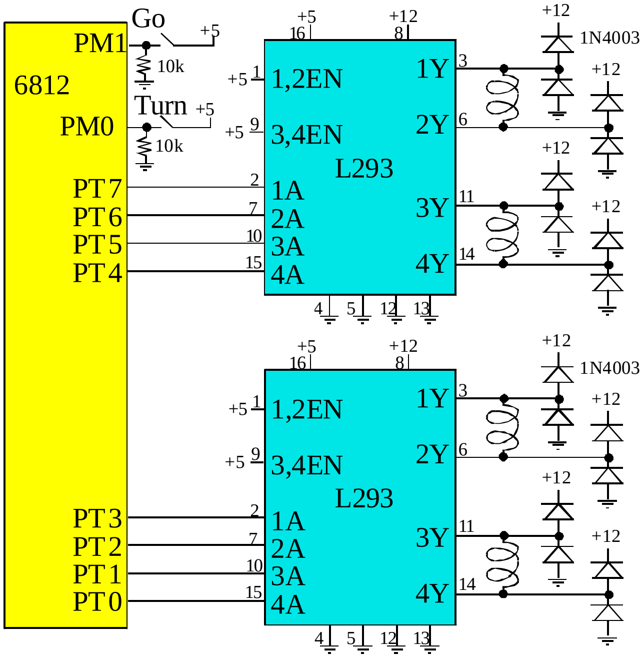
increase the size of **`Next[8]`**.

size = 2**(number of inputs)

Stepper motor controller

Inputs: Go and Turn

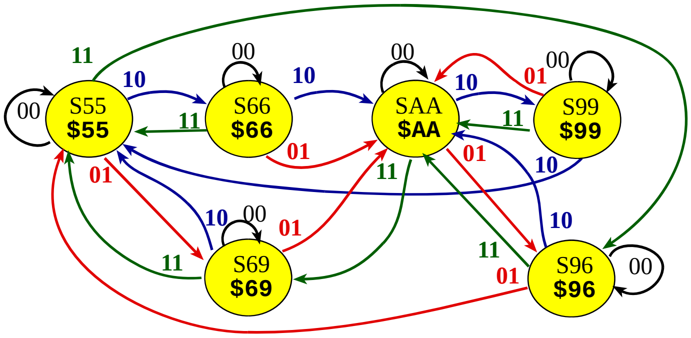
Outputs: two 4-wire bipolar stepper motors



Bipolar stepper motor interface using an L293 driver

```
// Port M bits 1-0 are inputs
// =00 Stop
// =10 Go    (55,66,AA,99)
// =01 RTurn(55,69,AA,96)
// =11 LTurn(55,96,AA,69)
// Port T bits 7-0 are outputs to steppers

const struct State {
  unsigned char out;       // command
  const struct State *next[4];};
typedef const struct State StateType;
StateType *Pt;
#define S55 &fsm[0]
#define S66 &fsm[1]
#define SAA &fsm[2]
#define S99 &fsm[3]
#define S69 &fsm[4]
#define S96 &fsm[5]
StateType fsm[6]={
  {0x55,{S55,S69,S66,S96}},  // S55
  {0x66,{S66,SAA,SAA,S55}},  // S66
  {0xAA,{SAA,S99,S99,S69}},  // SAA
  {0x99,{S99,SAA,S55,SAA}},  // S99
  {0x69,{S69,SAA,S55,S55}},  // S69
  {0x96,{S96,S55,SAA,SAA}}}; // S96
```



This stepper motor FSM has two input signals four outputs.

```
void main(void){
unsigned char Input;
  Timer_Init();
  DDRT = 0x0ff;
  DDRM = 0;
  Pt = S55;    // initial state
  while(1){    // never quit
    PTT = Pt->out;    // stepper drivers
    Timer_Wait(2000); // 0.25ms wait
    Input = PTM&0x03;
    Pt = Pt->next[Input];
  }
}
```

Write in assembly

The bottom line

FSM is good if:

1) the FSM is easy to understand,
2) the FSM is easy to change,
3) the state graph defines exactly what it does,
4) the state graph is 1-1 with the data structure,
5) each state has the same format.

In other words, if all you see is the state graph, there should be no ambiguity about what the machine does.