Table implementation for Moore FSM

<table>
<thead>
<tr>
<th>Input</th>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,10</td>
<td>goN</td>
<td>30</td>
</tr>
<tr>
<td>01,11</td>
<td>waitN</td>
<td>5</td>
</tr>
<tr>
<td>10,11</td>
<td>goE</td>
<td>30</td>
</tr>
<tr>
<td>01,11</td>
<td>waitE</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8.20. This Moore FSM controls traffic.

```
; Table structure
const struct State {
  unsigned char Out;
  unsigned short Time;
  unsigned char Next[4];};
typedef const struct State STyp;
#define goN   0
#define waitN 1
#define goE   2
#define waitE 3
STyp fsm[4]={
```

```
; Linked structure
const struct State {
  unsigned char Out;
  unsigned short Time;
  const struct State *Next[4];};
typedef const struct State sTyp;
#define goN   &fsm[0]
#define waitN &fsm[1]
#define goE   &fsm[2]
#define waitE &fsm[3]
STyp fsm[4]={
```
void main(void)
{
unsigned char n; // state number
unsigned char Input;
    DDRM = 0x3F;  // outputs
    ATDDIEN=0x03; // digital PAD1-0
    DDRAD= 0x00;  // inputs
    n = goN;
    while(1)
    {
        PTM = fsm[n].Out;
        Wait1sec(fsm[n].Time);
        Input = PTAD&0x03;
        n = fsm[n].Next[Input];
    }
}

Out   equ  0   8 bit data to output for this state
Time  equ  1   16-bit time to wait for this state
Next  equ  3   4 next indices given inputs 0,1,2,3
Size  equ  7
goN   equ  0
waitN equ  1
goE   equ  2
waitE equ  3

org  $4000

********************************

// state diagram
{0x21,30,{goN,waitN,goN,waitN}},
{0x22, 5,{goE,goE,goE,goE}},
{0x0C,30,{goE,goE,waitE,waitE}},
{0x14, 5,{goN,goN,goN,goN}}};
void main(void)
{
STyp *Pt; // state pointer
unsigned char Input;
    DDRM = 0x3F;
    ATDDIEN=0x03; // digital PAD1-0
    DDRAD= 0x00;  // inputs
    Pt = goN;
    while(1)
    {
        PTM = Pt->Out;
        Wait1sec(Pt->Time);
        Input = PTAD&0x03;
        Pt = Pt->Next[Input];
    }
}
fcb  goN,goN,goN,goN   *
*******************************
n     set  0    state number
Input  set  1
Main  lds  #$4000
  bsr  Timer_Init
  leas -2,s
  movb #$3F,DDRM      PM all outputs
  movb #$03,ATDDIEN   PAD1,PAD0 digital
  movb #$00,DDRAD       PAD1,PAD0 inputs
  movb #goN,n,sp     n = goN;
loop  ldaa n,s
     ldab #Size
     mul                 RegD=7*n
     ldx  #fsm
     leax D,x              RegX=fsm+7*n
     movb Out,x,PTM     PORTM = fsm[n].Out;
     ldd  Time,x        RegD= fsm[n].Time
     bsr  Timer_Wait     Wait1sec(fsm[n].Time);
     ldaa PTAD
     anda #$03          PTAD&0x03
     staa Input,s       Input = PTAD&0x03;
     leax a,x            RegX=fsm+7*n+Input
     movb Next,x,n,s     n = fsm[n].Next[Input];
     bra  loop

ICC12
first input parameter in RegD
rest of the input parameters on the stack
stack frame using RegX
return parameter in RegD

GCC12
first input parameter in RegD
rest of the input parameters on the stack
stack frame using memory location $0800
return parameter in RegD

Metrowerks
last input parameter in RegD
rest of the input parameters on the stack
stack frame using SP
return parameter in RegD
highly optimized

You must implement binding

Using a stack frame for parameters and local variables
Advantage: you can use the stack for other temporary
Disadvantage: ties up the use of either register X or Y

Using the stack pointer for parameters and local variables
Advantage: you can use both RegX and RegY
Disadvantage: can’t push more stuff on stack

It is possible to have both, simply by converting temporary
push/pull into local variables.

<table>
<thead>
<tr>
<th>* start with two LV</th>
<th>* but need temporary</th>
<th>* convert temporaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>* also need both X,Y</td>
<td>* use of stack</td>
<td>* to local variables</td>
</tr>
<tr>
<td>* 1) binding</td>
<td>* 1) binding</td>
<td>* 1) binding</td>
</tr>
<tr>
<td>LV1 set 0</td>
<td>LV1 set 0</td>
<td>LV1 set 0</td>
</tr>
<tr>
<td>LV2 set 2</td>
<td>LV2 set 2</td>
<td>LV2 set 2</td>
</tr>
<tr>
<td>subroutine</td>
<td>subroutine</td>
<td>subroutine</td>
</tr>
<tr>
<td>* 2) allocation</td>
<td>* 2) allocation</td>
<td>* 2) allocation</td>
</tr>
<tr>
<td>leas –4,sp</td>
<td>leas –4,sp</td>
<td>leas –7,sp</td>
</tr>
<tr>
<td>* 3) access</td>
<td>* 3) access</td>
<td>* 3) access</td>
</tr>
<tr>
<td>std LV1,sp</td>
<td>pshx</td>
<td>stx LV3,sp</td>
</tr>
<tr>
<td>ldx LV1,sp</td>
<td>psha</td>
<td>staa LV4,sp</td>
</tr>
<tr>
<td></td>
<td>std LV1,sp</td>
<td>std LV1,sp</td>
</tr>
<tr>
<td></td>
<td>pulb</td>
<td>ldb LV4,sp</td>
</tr>
<tr>
<td></td>
<td>ldx LV1,sp</td>
<td>ldx LV1,sp</td>
</tr>
<tr>
<td></td>
<td>puly</td>
<td>ldy LV3,sp</td>
</tr>
</tbody>
</table>

Jonathan W. Valvano
8.5. Expression evaluation

Polish Notation or Reverse Polish Notation
- numbers are pushed on the stack,
- values of the variables are pushed on the stack,
- unary function: input popped and result pushed,
- binary function: both inputs popped and result pushed.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Polish Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*M+N</td>
<td>3 M * N +</td>
</tr>
<tr>
<td>~(M</td>
<td>(N&amp;P))</td>
</tr>
<tr>
<td>M*(5+P)-N/10</td>
<td>M 5 P + * N 10 / -</td>
</tr>
<tr>
<td>w-x+y+z-4</td>
<td>w x – y + z + 4 -</td>
</tr>
</tbody>
</table>

Table 8.4. Examples of Polish Notation.

\[ P = (M+2)*(5+P)+3*N \quad \text{and} \quad M = 2 + 5 P + * 3 N * + \]
### 8.6. OS calls using software interrupts

The difference between a trap and a subroutine

- subroutine address binding occurs during assembly
- trap handler address stored at $FFFF6 and $FFFF7
Figure 8.15. A 6812 swi instruction can be used to call OS functions. On the 6811, the SP points to just above the top of the stack.
OS uses traps

- OS is a set of functions that facilitate use of computer
- OS is implemented as a set of trap handlers
- OS can be assembled and loaded into memory
- OS also sets the addresses in the trap vector table
- application program can be independently assembled
- when application requires OS function it executes a trap

<table>
<thead>
<tr>
<th>org $4000</th>
<th>org $4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>In equ 0</td>
<td>RDRF equ $20</td>
</tr>
<tr>
<td>Out equ 1</td>
<td>InChar ldaa SC0SR1</td>
</tr>
<tr>
<td>RegA set 2</td>
<td>bita #RDRF</td>
</tr>
<tr>
<td>SWIhan cmpb #In</td>
<td>beq InChar</td>
</tr>
<tr>
<td></td>
<td>ldab SC0SR1</td>
</tr>
<tr>
<td></td>
<td>andb #TDRE</td>
</tr>
<tr>
<td></td>
<td>beq OutChar</td>
</tr>
<tr>
<td></td>
<td>staa SC0DRL</td>
</tr>
<tr>
<td></td>
<td>rts</td>
</tr>
<tr>
<td>done rti</td>
<td></td>
</tr>
<tr>
<td>notIC cmpb #Out</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne done</td>
</tr>
<tr>
<td></td>
<td>bsr OutChar</td>
</tr>
<tr>
<td>main lds #$0C00</td>
<td>main lds #$0C00</td>
</tr>
<tr>
<td></td>
<td>ldaa #13</td>
</tr>
</tbody>
</table>
Program 8.47. 6812 assembly program showing the use of SWI to implement an OS function.

<table>
<thead>
<tr>
<th></th>
<th>loop</th>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldab #Out</td>
<td>bsr OutChar</td>
</tr>
<tr>
<td></td>
<td>swi</td>
<td>bsr InChar</td>
</tr>
<tr>
<td></td>
<td>ldab #In</td>
<td></td>
</tr>
<tr>
<td></td>
<td>swi</td>
<td>bsr OutChar</td>
</tr>
<tr>
<td></td>
<td>ldab #Out</td>
<td>bsr</td>
</tr>
<tr>
<td></td>
<td>swi</td>
<td>InChar</td>
</tr>
<tr>
<td></td>
<td>bra loop</td>
<td>bra loop</td>
</tr>
<tr>
<td>org</td>
<td>$FF6</td>
<td>org $FFFE</td>
</tr>
<tr>
<td>fdb</td>
<td>SWIhan</td>
<td>fdb main</td>
</tr>
<tr>
<td>org</td>
<td>$FFFE</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>main</td>
<td></td>
</tr>
</tbody>
</table>