Assembly Implementation of Traffic Light Controller

org $800
OUT equ 0 ;offset for output
WAIT equ 1 ;offset for time (8 bits+OUT)
NEXT equ 3 ;offset for next state (16 bits+WAIT)
goN fcb $21 ;East red, north green
goe fcb $14 ;East green, north red
waitN fcb $22 ;East red, north yellow
waitE fcb $22 ;East red, north yellow
goE fcb $0C ;East green, north red

Assembly Implementation of Traffic Light Controller

Main lds #$4000 ;stack init
bsr Timer ;enable TCNT
movb #$FF,DDRB ;PORTB5-0 set to output to lights
movb #$00,DDRA ;PORTA1-0 set to input from sensors
ldx #goN ;Initialize state pointer (register X)
FSM ldab OUT,x stab PORTB
ldy WAIT,x bsr Timer,Wait10ms
ldab PORTA
andb #$03 ;Keep the bottom two bits
lslb ;Multiply by two b/c addresses are 2 bytes
abx ;add 0,2,4,6
ldx NEXT,x
bra FSM

Code Execution

ldx #$1000 ;Initialize state pointer (register X)
FSM ldab OUT,x
...

Memory Map

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td>WAIT</td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td>goE</td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td>waitN</td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td>goE</td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td>goN</td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td>waitN</td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td>waits</td>
<td>0800</td>
<td>01 F4</td>
<td>waits</td>
</tr>
<tr>
<td>goE</td>
<td>080E</td>
<td>08 16</td>
<td>ns0</td>
</tr>
<tr>
<td>goE</td>
<td>0810</td>
<td>08 16</td>
<td>ns1</td>
</tr>
<tr>
<td>goE</td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td>goE</td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

RegX XX XX
RegY XX XX
AccB XX

State Address Value Comment

goN 0800 21 out

No cars Car E Car N Car N,E

goN goN | waitN goN | waitN
waitN goE | goE | goE | goE
waitE goN | goN | goN | goN

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td>goO</td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td>goE</td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td>waitN</td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td>goE</td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td>goN</td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td>waitN</td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td>waits</td>
<td>0800</td>
<td>01 F4</td>
<td>waits</td>
</tr>
<tr>
<td>goE</td>
<td>080E</td>
<td>08 16</td>
<td>ns0</td>
</tr>
<tr>
<td>goE</td>
<td>0810</td>
<td>08 16</td>
<td>ns1</td>
</tr>
<tr>
<td>goE</td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td>goE</td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

RegX XX XX
RegY XX XX
AccB XX

Code Execution

ldx #goN
FSM ldab OUT,x
stab PORTB
ldy WAIT,x bsr Timer,Wait10ms
ldab PORTA
andb #$03 ;Keep the bottom two bits
lslb ;Multiply by two b/c addresses are 2 bytes
abx ;add 0,2,4,6
ldx NEXT,x
bra FSM
Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 7/42

Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 8/42

Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 9/42

Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 10/42

Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 11/42

Scott R. Little Lecture 4 Supplemental Material ECE 5780/6780 12/42
Code Execution

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td></td>
<td>0800</td>
<td>21, out</td>
</tr>
<tr>
<td>FSM</td>
<td>ldab OUT,x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stab</td>
<td>PORTB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldy</td>
<td>WAIT,x</td>
<td>0803</td>
<td>00, ns0</td>
</tr>
<tr>
<td>bar</td>
<td>Timer,Wait10ms</td>
<td>0805</td>
<td>08, ns1</td>
</tr>
<tr>
<td>ldab</td>
<td>PORTA</td>
<td>0807</td>
<td>08, ns2</td>
</tr>
<tr>
<td>andb</td>
<td>#$03</td>
<td>0809</td>
<td>08, ns3</td>
</tr>
<tr>
<td>lslb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldx</td>
<td>NEXT,x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bra</td>
<td>FSM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RegX 08 02
RegY 0B B8
AccB 02

State Address Value Comment
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tax
leas -4,sp
movv #0,sum,x
movv #100,n,x

loop ldd n,x
addd sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
txs
pulx

Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tax
leas -4,sp
movv #0,sum,x
movv #100,n,x

loop ldd n,x
addd sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
txs
pulx

Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tax
leas -4,sp
movv #0,sum,x
movv #100,n,x

loop ldd n,x
addd sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
txs
pulx

Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tax
leas -4,sp
movv #0,sum,x
movv #100,n,x

loop ldd n,x
addd sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
txs
pulx
Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |

Local variable allocation/deallocation

<table>
<thead>
<tr>
<th>sum set -4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n set -2</td>
</tr>
<tr>
<td>calc pshx</td>
</tr>
<tr>
<td>txs</td>
</tr>
<tr>
<td>leas -4,sp</td>
</tr>
<tr>
<td>movv #0, sum, x</td>
</tr>
<tr>
<td>movv #100, n, x</td>
</tr>
</tbody>
</table>

Loop

| ldd n, x |
| addd sum, x |
| std sum, x |
| ldd n, x |
| std n, x |
| bne loop |
| txs |
| pulx |
Correct code: Who do you believe?

- pg. 128 of your textbook - "Recursive algorithms are often easy to prove correct."

Introduction

- Coding guidelines that cannot be checked by a tool are less effective.
- Too many coding guidelines aren’t effective because they are not remembered or enforceable.
- The cost of restrictive guidelines may pay off with code that is more correct.

Rule 1

*Rule:* Restrict all code to very simple control flow constructs – do not use goto statements, setjmp or longjmp constructs, and direct or indirect recursion.
- Simple control translates into easier code verification and often improved clarity.
- Without recursion the function call graph is acyclic which directly aids in proving boundedness of the code.
- This rule doesn’t require a single return point for a function although this often simplifies control flow.

Rule 2

*Rule:* All loops must have a fixed upper-bound. It must be trivially possible for a checking tool to prove statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.
- The absence of recursion and presence of loop bounds prevents runaway code.
- Functions intended to be nonterminating must be proved to not terminate.
- Some functions don’t have an obvious upper bound (i.e. traversing a linked list), so an artificial bound should be set and checked via an assert.

Rule 3

*Rule:* Do not use dynamic memory allocation after initialization.
- Memory allocation code is unpredictable from a time standpoint and therefore impractical for time critical code.
- Many errors are introduced by improper dynamic memory allocation.
- Without dynamic memory allocation the stack is used for dynamic structures and without recursion bounds can be proved on stack size.

Rule 4

*Rule:* No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than 60 lines of code per function.
- Long functions often indicate poor code structure.
Rule 5

Rule: The assertion density should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken.

- Use of assertions is recommended as part of a strong defensive coding strategy.
-Assertions can be used to check pre- and post-conditions of functions, parameter values, return values, and loop invariants.
-Assertions can be disabled in performance critical code because they are side-effect free.

Rule 6

Rule: Data objects must be declared at the smallest possible level of scope.

- Variable will not be modified in unexpected places if they are not in scope.
- It can be easier to debug a problem if the scope of the variable is smaller.

Rule 7

Rule: The return value of non-void functions must be checked by each calling function, and the validity of parameters must be checked in each function.

- If the response to the error would be no different to the response to the success then there is no point in checking the value.
- Useless checks can be indicated by casting the return value to (void).

Rule 8

Rule: The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists, and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives is often also dubious but cannot always be avoided. Each use of a conditional compilation directive should be flagged by a tool-based checker and justified in the code.

- Conditional compilation directives can result in an exponentially growing number of code versions.

Rule 9

Rule: The use of pointers should be restricted. Specifically, no more than one level of dereferencing is allowed. Pointer dereference operations may not be hidden in macro definitions or inside typedef declarations. Function pointers are not permitted.

- Pointers are easily misused even by experienced programmers.
- Function pointers can severely limit the utility of static code checkers.

Rule 10

Rule: All code must be compiled, from the first day of development, with all compiler warnings enabled at the compiler’s most pedantic setting. All code must compile with these settings without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static code analyzer and should pass the analyses with zero warnings.

- This rule should be followed even in the case when the warning is invalid.
- Code that confuses the compiler or checker enough to result in an invalid warning should be rewritten for clarity.
- Static checkers should be required for any serious coding project.