

Student: “Programming in assembly is a lot like digging a post hole with a teaspoon. Of course, you do have complete control over every piece of dirt”

Valvano: “Programming in a high level language is a lot like painting a masterpiece with a paint roller. Of course, you do get the job done quickly.”

Recap

Finite State Machines
Pointer implementation

Overview

Fixed-point: why, when, how
Local variables: scope and allocation
How these concepts apply to C
Binding, allocation, access, deallocation

10.1. Fixed-point numbers

Fixed point numbers

Why? (wish to represent non-integer values)
Next lab measures distance from 0 to 3 cm
E.g., 1.234 cm

When? (range is known, range is small)
Range is 0 to 3cm
Resolution is 0.003 cm

How? (value = Integer* Δ)
16-bit unsigned integer
 $\Delta = 10^{-3}$ decimal fixed-point
Range becomes 0.000 to 65.535

Output an integer.

Assume integer,
n, is between 0 and 9999.

not very pretty

OutChar(\$30+n/1000) ;thousand's digit

n = n%1000

OutChar(\$30+n/100) ;hundred's digit

n = n%100

OutChar(\$30+n/10) ;ten's digit

OutChar (\$30+n%10) ;one's digit

Output a fixed-point number.

Assume the integer part of the fixed point number,
n, is between 0 and 9999.

very pretty

```

OutChar($30+n/1000)           ;thousand's digit
n = n%1000
OutChar($2E)                   ;decimal point
OutChar($30+n/100)           ;hundred's digit
n = n%100
OutChar($30+n/10)             ;ten's digit
OutChar ($30+n%10)           ;one's digit

```

7.3. Local Variables

Introduction

scope => from where can this information be accessed

local means restricted to current program segment

global means any software can access it

allocation => when is it created, when is it destroyed

dynamic allocation using registers or stack

permanent allocation assigned a block of memory

A local variable (local scope, dynamic allocation)

temporary information

used only by one software module

allocated, used, then deallocated

not permanent

implement using the stack or registers

Reasons why we place local variables on the stack include

- dynamic allocation/release allows for reuse of memory
- limited scope of access provides for data protection
- only the program that created the local can access it
- the code is reentrant.
- the code is relocatable
- the number of variables is more than registers

Registers are local variables

Allocation: Register assigned to a task

Access: Register is used

Deallocation: Register free for other tasks

Line	Program	RegB	RegX	RegY
1	Main lds #\$4000			
2	bsr Timer_Init			
3	ldab #\$FC	\$FC		
4	stab DDRT	\$FC		
5	ldx #goN		Pt	
6	FSM ldab OUT,x	Output	Pt	
7	lslb	Output	Pt	
8	lslb	Output	Pt	
9	stab PTT	Output	Pt	

10	<code>ldy WAIT,x</code>		Pt	Wait
11	<code>bsr Timer_Wait10ms</code>		Pt	Wait
12	<code>ldab PTT</code>	Input	Pt	
13	<code>andb #\$03</code>	Input	Pt	
14	<code>lslb</code>	Input	Pt	
15	<code>abx</code>	Input	Pt	
16	<code>ldx NEXT,x</code>		Pt	
17	<code>bra FSM</code>		Pt	

Program 7.1. Register assignments in a finite state machine controller.

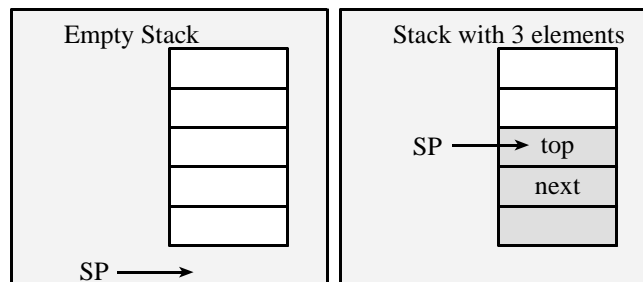


Figure 7.1. The 9S12 stack.

The `tsx` and `tsy` instructions do not modify the stack pointer.

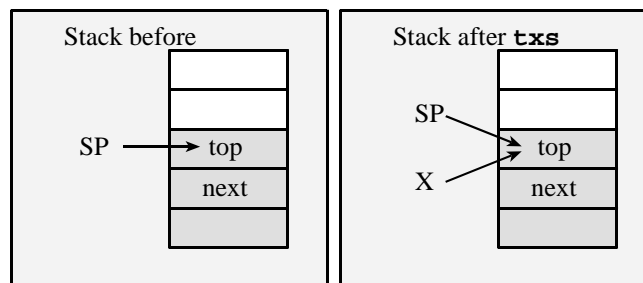


Figure 7.2. The `tsx` instruction creates a stack frame pointer.

The LIFO stack has a few rules:

1. Program segments should have an equal number of pushes and pulls;
2. Stack accesses (PUSH or PULL) should not be performed outside the allocated area;
3. Stack reads and writes should not be performed within the *free area*,
 PUSH should first decrement SP, then store the data,
 PULL should first read the data, then increment SP.

7.3. Local variables allocated on the stack

Stack implementation of local variables has four stages:

- binding
- allocation
- access, and
- deallocation.

1. **Binding** is the assignment of the address (not value) to a symbolic name.

```
sum set 0 ;16-bit local variable
```

2. **Allocation** is the generation of memory storage for the local variable.

```
pshx ;allocate sum
```

In this next example, the software allocates the local variable by decrementing the stack pointer. This local variable is also uninitialized.

```
des ;allocate sum
des
```

If you wished to allocate the 16-bit local and initialize it to zero, you could execute.

```
movw #0,2,-sp
```

This example allocates 20 bytes for the structure `big[20]`.

```
leas -20,sp ;allocate big[20]
```

3. The **access** to a local variable is a read or write operation that occurs during execution. In the next code fragment, the local variable `sum` is set to 0.

```
movw #0,sum,sp
```

In the next code fragment, the local variable `sum` is incremented.

```
ldd sum,sp
add #1
std sum,sp ;sum=sum+1
```

4. **Deallocation** is the release of memory storage for the location variable.

```
pulx ;deallocate sum
```

In this next example, the software deallocates the local variable by incrementing the stack pointer.

```
ins
ins ;deallocate sum
```

In this last example, the technique provides a mechanism for allocating large amounts of stack space.

```
leas 20,sp ;deallocate big[20]
```

Example of local variables on stack

```
org $4000
; calculate sum of numbers
; Input: RegD num
; Output: RegD Sum of 1,2,3,...,num
; Errors: may overflow
; 1) binding
num set 2 ;loop counter 1,2,3
sum set 0 ;running
calc
; 2) allocation
pswd ;allocate num
movw #0,2,-sp ;sum=0

; 3) access
loop ldd sum,sp
add num,sp
std sum,sp ;sum = sum+num
ldd num,sp
subd #1
std num,sp ;num = num-1
bne loop
ldd sum,sp ;result

; 4) deallocate
leas 4,sp
rts

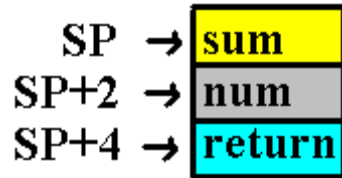
main lds #$4000
ldd #100
jsr calc
bra *
org $FFFE
fdb main
```

Draw a stack picture

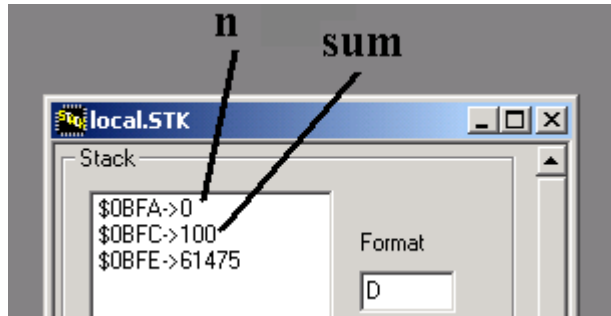
1) in text form

```
SP -> sum
SP+2 -> num
SP+4 -> return address
```

2) graphically



3) using TExaS



*****Run on TExaS*****

The bottom line

Scope specifies which module can access
 limiting scope reduces complexity
 Allocation specifies where data is located
 Temporary register,
 Permanent RAM (`data rmb 2`)
 Temporary RAM (`pt = malloc(100);`)
 Permanent ROM (`list fcb 5,6,10,9`)
 Temporary on stack
 Binding, allocation, access, deallocation