

Introducing Subroutines

- The delay code introduced as part of the traffic lights program is an example of a self-contained set of instructions that perform some specific and useful subtask – supporting the main task of turning on and off the traffic light.
- These subtask programs are called subroutines and correspond to functions, methods and procedures of high-level language programs.
- Delaying is an action that is required at multiple points in the traffic lights program. As such, it would be efficient if we only had to write the code once but could use it multiple times, as required.
- In addition to efficiency; this facility would also introduce a degree of structure to our program by formalizing the way in which we access and leave subroutines.
- This structuring, however, requires the addition of additional hardware and instructions at the Instruction Set Architecture Level and additional mnemonics at the Assembly Language Level.

151

Formally Introducing the Stack

Exploring the Stack in Samphire

- A Stack is a way of structuring information and accessing that information in a first-in-last-out manner. This contrasts with a queue, for example, which is accessed in a first-in-first-out manner.
- You will come across many other ways of organizing data during your CS studies.
- The Samphire Stack begins at address BF – this is known as the “bottom of the stack”.

IP	00000000	00	+000
SP	10111111	BF	-065
SR	00000000	00	+000
ISOZ			

152

Formally Introducing the Stack

- The structured way of accessing information in a stack is via the operations: `push` and `pop`.
- `push` places information on the stack and `pop` takes it off
- Contrast this with the operations for accessing a queue. These are `join` – to join the queue and `serve` – to leave the queue.

153

Formally Introducing the Stack

- The address of the next free location on the stack is kept in the stack pointer (SP) register.
- In the RAM view of the Samphire Simulator, this location is marked by a blue cursor.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

154

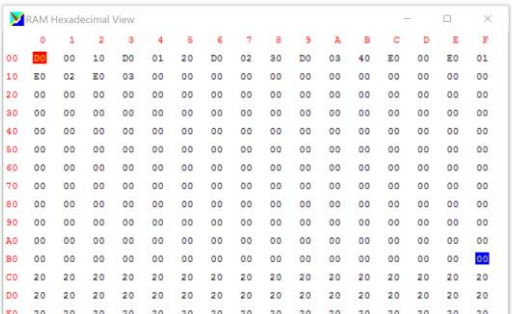
Formally Introducing the Stack

- The stack 'grows' into lower memory addresses as values are pushed onto the stack.
- Consider a program to push a 4 values onto the stack:

```

mov al, 10
mov bl, 20
mov cl, 30
mov dl, 40
push al
push bl
push cl
push dl
end

```



155

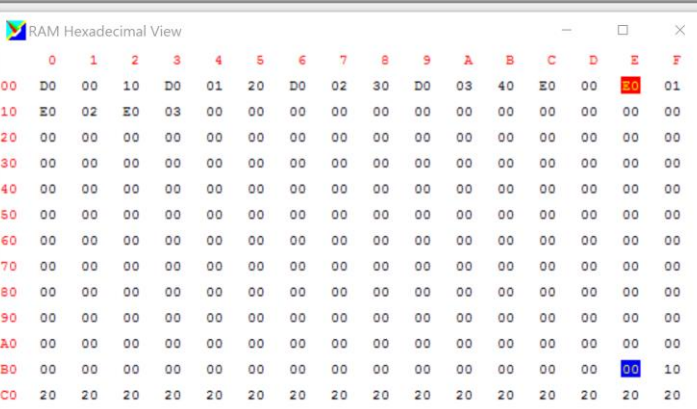
Formally Introducing the Stack

- After pushing the first value, the Stack Pointer will have the value BE:

```

mov al, 10
mov bl, 20
mov cl, 30
mov dl, 40
push al
push bl
push cl
push dl
end

```



156

Formally Introducing the Stack

- After pushing all 4 values, the SP will have the value BB:

The screenshot shows a debugger interface. On the left, assembly code is listed:


```

    AL 00010000 10 +016 IF 00010100 14 +020
    BL 00100000 20 +032 SF 10111011 BB -069
    CL 00110000 30 +048 SR 00000000 00 +000
    DL 01000000 40 +064 ISQZ
    end

    mov al, 10
    mov bl, 20
    mov cl, 30
    mov dl, 40
    push al
    push bl
    push cl
    push dl
    end
    
```

 The right pane shows the 'RAM Hexadecimal View' window. It displays a memory dump with columns for addresses (00 to FF) and hex values. The value at address BB (40) is highlighted in blue, showing the hex value 40.

157

Formally Introducing the Stack

- The address of the next free location in the stack is always given by the value in the SP.
- The address of the most recent value placed on the stack is at address SP+1. This address is called the top of the stack.
- If the value of the next free location in the stack is equal to the bottom of the stack (BF, in the case of Samphire), the stack is said to be empty.
- Trying to `pop` an empty stack will be assessed as an illegal operation.

158

Formally Introducing the Stack

- The `push` instruction
 1. Copies the value in its operand register into memory at the address specified in the SP.
 2. Subtracts 1 from the SP (Causing the stack to 'grow' into lower memory addresses).

159

Formally Introducing the Stack

- Values should only be removed from the stack by using the `pop` instruction
- The `pop` instruction
 1. Adds 1 to the SP
 2. Copies the value from memory at the address specified in the SP to its register operand.
Popping an empty stack will result in error.

160

Formally Introducing the Stack

The screenshot shows an assembly IDE with the following assembly code:

```

AL 00010000 10 +016 IP 00010100 14 +020
BL 00100000 20 +032 SP 10111011 BB -069
CL 00110000 30 +048 SR 00000000 00 +000
DL 01000000 40 +064      ISOZ

```

The assembly code is:

```

mov al, 10
mov bl, 20
mov cl, 30
mov dl, 40
push al
push bl
push cl
push dl
pop al
pop bl
pop cl
pop dl
end

```

The RAM Hexadecimal View window shows the stack contents:

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	D0	00	10	D0	01	20	D0	02	30	D0	03	40	E0	00	E0	01
10	E0	02	E0	03	E1	00	E1	01	E1	02	E1	03	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
D0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20

161

Formally Introducing the Stack

Note that this program, when run, results in the order of the values in al, bl, cl, and dl being reversed.

The screenshot shows an assembly IDE with the following assembly code:

```

AL 01000000 40 +064 IP 00011100 1C +028
BL 00110000 30 +048 SP 10111111 BF -065
CL 00100000 20 +032 SR 00000000 00 +000
DL 00010000 10 +016      ISOZ

```

The assembly code is:

```

mov al, 10
mov bl, 20
mov cl, 30
mov dl, 40
push al
push bl
push cl
push dl
pop al
pop bl
pop cl
pop dl
end

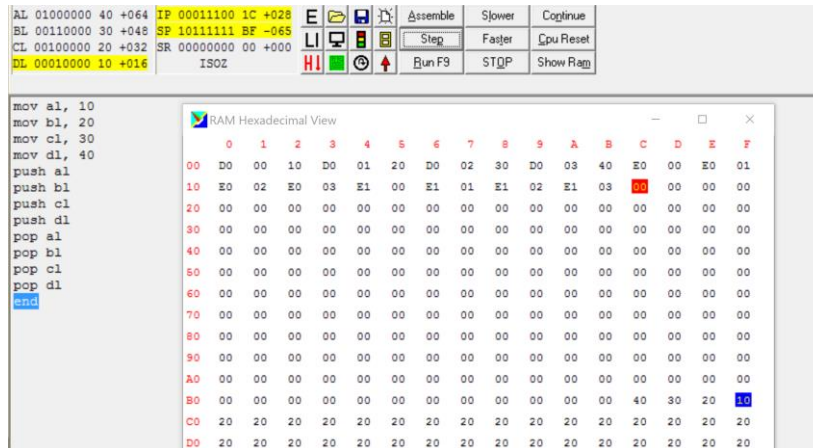
```

The RAM Hexadecimal View window shows the stack contents:

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	D0	00	10	D0	01	20	D0	02	30	D0	03	40	E0	00	E0	01
10	E0	02	E0	03	E1	00	E1	01	E1	02	E1	03	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
D0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20

162

Note also that the values that were on the stack remain in memory. They will be overwritten by subsequent pushes



- Call-Return provides low-level support for the implementation of subroutines.
- The `call` Instruction
 1. Transfers control to the called subroutine by placing the subroutine address into the Instruction Pointer (Program Counter).
 2. Pushes the address of the instruction that follows the `call` instruction (this is called the return address) onto the stack.

Introducing a Subroutine Call-Return Mechanism

- In Samphire, the `call` instruction has the following format:

`call <address>`

Where `<address>` is the memory address of the start of the subroutine.

- To help to determine the start address of a subroutine, Samphire uses the *Assembler Directive*: `org`

(A label is another example of an Assembler Directive. It helps the assembler to do its job by providing some helpful information. You should not confuse an assembler directive with an assembly language instruction. The latter, translates directly into a machine-level instruction and is executed by the hardware.)

165

Introducing a Subroutine Call-Return Mechanism

- `org` tells the assembler to place the following instructions/data into memory starting at a specific address.

`org <address>`

When writing a subroutine, this instruction will be used to place the subroutine code into memory at address `<address>`.

(We will use `org` later to put a collection of data into memory at a specific location)

166

Introducing a Subroutine Call-Return Mechanism

Example

```
mov al, 30
call 70      ;Subroutine Call
mov [c0], al
```

```
org 70
add al, 1 } Code of the
ret       } Subroutine.
```

Every subroutine ends with a ret (return) instruction.

This instruction pops the return address from the stack into the Instruction Pointer. Thus passing control back to the instruction following the call instruction.

167

Introducing a Subroutine Call-Return Mechanism

Call

```
mov al, 30
call 70
mov [c0], al

org 70
add al, 1
ret
end
```

Subroutine Start

ret

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	B0	00	30	CA	70	D2	C0	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	B0	00	01	CB	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20

168

Introducing a Subroutine Call-Return Mechanism

Address	Instruction	Stack	IP
03	call 70		03
05	mov [c0], al	05	70
70	add al, 1		
73	ret	05	73
			05

169

Introducing a Subroutine Call-Return Mechanism

```

;Traffic Lights with a fixed duration delay using subroutines

start:
  mov al, 84      ; 84 corresponds to Red-Green on the Traffic Lights
  out 01          ; Write to Traffic Lights Port
  call 70         ;
  mov al, 48      ; 48 corresponds to Amber-Amber on the Traffic Lights
  out 01          ; Write to Traffic Lights Port
  call 70
  mov al, 30      ; 30 corresponds to Green-Red on the Traffic Lights
  out 01          ; Write to Traffic Lights Port
  call 70
  mov al, 48      ; 48 corresponds to Amber-Amber on the Traffic Lights
  out 01          ; Write to Traffic Lights Port
  call 70
  jmp start

org 70 ;
  mov bl, fa      ; initialize bl with a value representing the delay
loop:
  inc bl
  cmp bl, 00      ; check to see if bl has overflowed
  jnz loop        ; if not continue incrementing and checking
  ret
end

```

170

Introducing a Subroutine Call-Return Mechanism

- Note:
 1. that this code is much more structured, easier to read, and shorter than the non subroutine version.
 2. The length of the delay in the subroutine is determined by the value in the bl register.

This register is initialized to a particular value and is incremented until its value returns to zero. We can act on this situation by jumping conditionally on the value of the zero flag.

The greater the initial value of bl, the faster it will return to zero – the shorter the delay.