Subroutines II

Parameter Passing, Local Variables

Outline

- Subroutines contd.
- Parameter passing through stack
- Local variables

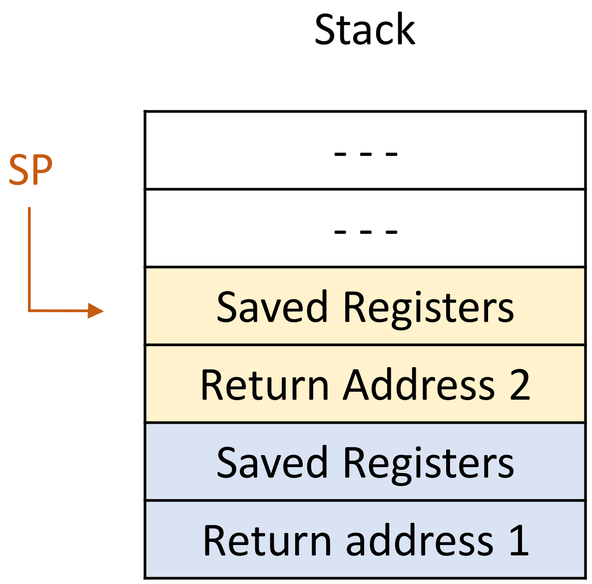
Recap

- Use of stack with subroutines

```
subrotuine1: push ax
             push bx
             ...
             call subroutine2
             ...
             pop bx
             pop ax
             ret

subrotuine2: push cx
             ...
             pop cx
             ret

start:       call subroutine1
```



Parameter passing

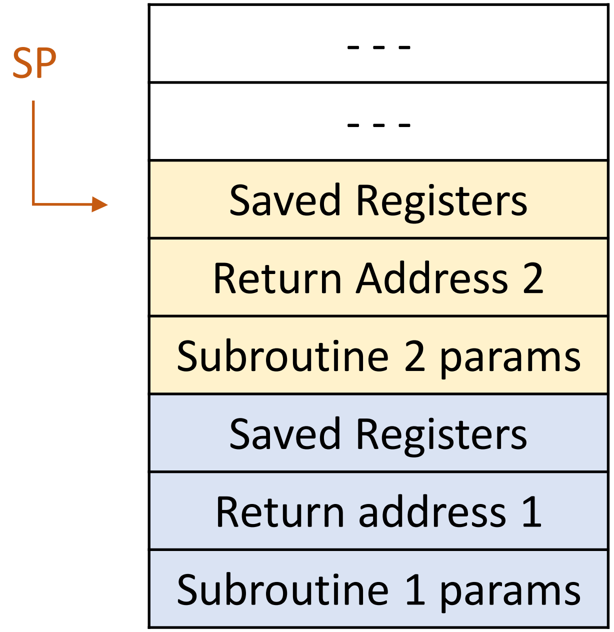
- So far we have passed parameters to subroutines via **registers**
- Parameter passing by registers is constrained in two ways.
  - The maximum parameters a subroutine can receive are seven when all the general registers are used.
  - Also, subroutines are themselves limited in their use of registers, and this limitation increases when the subroutine has to make a nested call thereby using certain registers as its parameters.
- Hence, parameter passing by registers is not expandable and generalizable.

Parameter passing through Stack

- Considering stack as an alternate, we observe that whatever data is placed there, it stays there, and across function calls as well.
- For example the bubble sort subroutine needs an array address and the count of elements.
- If we place both of these on the stack, and call the subroutine afterwards, it will stay there.
- The subroutine is invoked with its return address on top of the stack and its parameters beneath it.

Parameter passing through Stack

- This is how stack would like once two nested calls have been made



Accessing arguments

- To access the arguments from the stack, the immediate idea that strikes is to pop them off the stack.
- And this is the only possibility using the given set of information.
- However the first thing popped off the stack would be the return address and not the arguments.
- This is because the arguments were first pushed on the stack and the subroutine was called afterwards.
- The arguments cannot be popped without first popping the return address.

Accessing arguments

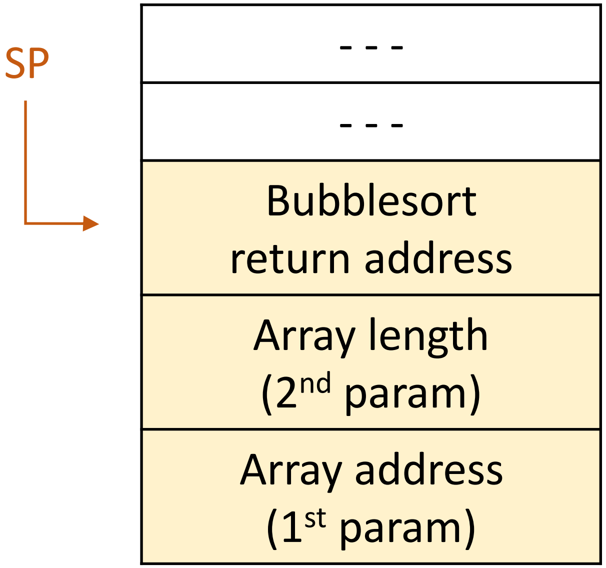
- To handle this using PUSH and POP, we must first pop the return address in a register, then pop the operands, and push the return address back on the stack so that RET will function normally.
- However so much effort doesn’t seem to pay back the price.
- Processor designers should have provided a logical and neat way to perform this operation.
- They did provided a way and in fact we will do this without introducing any new instruction.

Accessing arguments: BP register

- Recall that the default segment association of the BP register is the stack segment.
  - The reason is to peek inside the stack using the BP register and read the parameters without removing them and without touching the stack pointer.
- The stack pointer could not be used for this purpose, as it cannot be used in an effective address.
  - Something like mov ax, [SP] is not supported
- The base pointer is provided as a replacement of the stack pointer so that we can peek inside the stack without modifying the structure of the stack.

Accessing arguments: BP register

- When the bubble sort subroutine is called, the stack pointer is pointing to the return address.
- Two bytes below it is the second parameter and four bytes below is the first parameter.
- The stack pointer is a reference point to these parameters.



Accessing arguments: BP register

- If the value of SP is captured in BP, then the return address is located at [bp+0], the second parameter is at [bp+2], and the first parameter is at [bp+4].
- This is because SP and BP both had the same value and they both defaulted to the same segment, the stack segment.
- This copying of SP into BP is like taking a snapshot or like freezing the stack at that moment.
- Even if more pushes are made on the stack decrementing the stack pointer, our reference point will not change.
- The parameters will still be accessible at the same offsets from the base pointer.

Preserving original BP value

- However we have destroyed the original value of BP in the process, and this will cause problems in nested calls where both the outer and the inner subroutines need to access their own parameters.
- The outer subroutine will have its base pointer destroyed after the call to inner subroutine and will be unable to access its parameters.

Preserving original BP value

- To solve both of these problems, we reach at the standard way of accessing parameters on the stack.
- The first two instructions of any subroutines accessing its parameters from the stack are given below:
  - push bp
  - mov bp, sp
- As a result our datum point has shifted by a word.
- Now the old value of BP will be contained in [bp] and the return address will be at [bp+2].
- The second parameters will be [bp+4] while the first one will be at [bp+6].

Example 5.5

```
; bubble sort subroutine taking parameters from stack
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag:     db   0

bubblesort:   push bp                     ; save old value of bp
              mov  bp, sp                 ; make bp our reference point
              push ax                     ; save old value of ax
              push bx                     ; save old value of bx
              push cx                     ; save old value of cx
              push si                     ; save old value of si

              mov  bx, [bp+6]             ; load start of array in bx
              mov  cx, [bp+4]             ; load count of elements in cx
              dec  cx                     ; last element not compared
              shl  cx, 1                  ; turn into byte count

mainloop:     mov  si, 0                  ; initialize array index to zero
              mov  byte [swapflag], 0 ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]            ; load number in ax
              cmp  ax, [bx+si+2]          ; compare with next number
              jbe  noswap                 ; no swap if already in order

              xchg ax, [bx+si+2]          ; exchange ax with second number
              mov  [bx+si], ax            ; store second number in first
              mov  byte [swapflag], 1 ; flag that a swap has been done

noswap:       add  si, 2                  ; advance si to next index
              cmp  si, cx                 ; are we at last index
              jne  innerloop              ; if not compare next two
```

```
37                  cmp  byte [swapflag], 1 ; check if a swap has been done
38                  je   mainloop           ; if yes make another pass
39
40                  pop  si                 ; restore old value of si
41                  pop  cx                 ; restore old value of cx
42                  pop  bx                 ; restore old value of bx
43                  pop  ax                 ; restore old value of ax
44                  pop  bp                 ; restore old value of bp
45                  ret  4                  ; go back and remove two params
46
47    start:        mov  ax, data
48                  push ax                 ; place start of array on stack
49                  mov  ax, 10
50                  push ax                 ; place element count on stack
51                  call bubblesort         ; call our subroutine
52
53                  mov  ax, data2
54                  push ax                 ; place start of array on stack
55                  mov  ax, 20
56                  push ax                 ; place element count on stack
57                  call bubblesort         ; call our subroutine again
58
59                  mov  ax, 0x4c00         ; terminate program
60                  int  0x21
```

11	The value of the stack pointer is captured in the base pointer. With further pushes SP will change but BP will not and therefore we will read parameters from bp+4 and bp+6.
45	The form of RET that takes an argument is used causing four to be added to SP after the return address has been popped in the instruction pointer. This will effectively discard the parameters that are still there on the stack.
47-50	We push the address of the array we want to sort followed by the count of elements. As immediate cannot be directly pushed in the 8088 architecture, we first load it in the AX register and then push the AX register on the stack.

Clearing parameters from stack

- After the subroutine has returned, its parameters left on the stack are a waste. They have to be cleared from the stack.
- Either of the caller and the callee can take the responsibility of clearing them from there.
- Stack clearing by the caller needs an extra instruction on behalf of the caller after every call made to the subroutine, unnecessarily increasing instructions in the program.
    - If there are thousand calls to a subroutine the code to clear the stack is repeated a thousand times.

Clearing parameters from stack

- If the callee has to clear the stack it can do this easily with **RET n** instruction, where n is the number of *bytes* to remove from stack.
- Its operation is
  1) IP ← [SP]
  2) SP ← SP + 2 + n
- Note the order of above operations.
- Without the +n part, it is a regular RET. Adding n to SP effectively discards n bytes at the top of stack.

Subroutine local variables

- Another important role of the stack is in the creation of local variables that are only needed while the subroutine is in execution and not afterwards.
- They should not take permanent space like global variables.
- Local variables should be created when the subroutine is called and discarded afterwards.
  - So that the spaced used by them can be reused for the local variables of another subroutine.
- They only have meaning inside the subroutine and no meaning outside it.

Local variables on stack

- The most convenient place to store these variables is the stack.
- We need some special manipulation of the stack for this task. i.e we need to produce a gap in the stack for our variables.
- This is explained with the help of the swapflag in the bubble sort example.
- Previously we declared swapflag as global variable, permanently occupying space in memory. It is only needed by the bubble sort subroutine and should be a local variable.

Local variables on stack

- The stack pointer will be decremented by an extra two bytes thereby producing a gap in which a word can reside.
- This gap will be used for our temporary, local, or automatic variable; however we name it.
- We can decrement it as much as we want producing the desired space, however the decrement must be by an even number, as the unit of stack operation is a word.

Accessing local variables

- The most convenient time for creating this gap is immediately after saving the value of SP in BP.
- So that the same base pointer can be used to access the local variables as well; this time using negative offsets.
- The standard way to start a subroutine which needs to access parameters and has local variables is as under.
    - push bp
    - mov bp, sp
    - sub sp, 2
- Gap of any size can be created in a single instruction with subtraction.

Local variables

- The parameters can still be accessed at bp+4 and bp+6 and the swapflag can be accessed at bp-2.
- The subtraction in SP was after taking the snapshot; therefore BP is above the parameters but below the local variables.
- The parameters are therefore accessed using positive offsets from BP and the local variables are accessed using negative offsets.

Example 5.6

```
; bubble sort subroutine using a local variable
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5

bubblesort:   push bp                   ; save old value of bp
              mov  bp, sp               ; make bp our reference point
              sub sp, 2                 ; make two byte space on stack
              push ax                   ; save old value of ax
              push bx                   ; save old value of bx
              push cx                   ; save old value of cx
              push si                   ; save old value of si

              mov  bx, [bp+6]           ; load start of array in bx
              mov  cx, [bp+4]           ; load count of elements in cx
              dec  cx                   ; last element not compared
              shl  cx, 1                ; turn into byte count

mainloop:     mov  si, 0                ; initialize array index to zero
              mov  word [bp-2], 0       ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]          ; load number in ax
              cmp  ax, [bx+si+2]        ; compare with next number
              jbe  noswap               ; no swap if already in order

              xchg ax, [bx+si+2]        ; exchange ax with second number
              mov  [bx+si], ax          ; store second number in first
              mov  word [bp-2], 1       ; flag that a swap has been done
```

```
33  noswap:       add  si, 2               ; advance si to next index
34                cmp  si, cx              ; are we at last index
35                jne  innerloop           ; if not compare next two
36
37                cmp  word [bp-2], 1      ; check if a swap has been done
38                je   mainloop            ; if yes make another pass
39
40                pop  si                  ; restore old value of si
41                pop  cx                  ; restore old value of cx
42                pop  bx                  ; restore old value of bx
43                pop  ax                  ; restore old value of ax
44                mov  sp, bp              ; remove space created on stack
45                pop  bp                  ; restore old value of bp
46                ret  4                   ; go back and remove two params
47
48  start:        mov  ax, data
49                push ax                  ; place start of array on stack
50                mov  ax, 10
51                push ax                  ; place element count on stack
52                call bubblesort          ; call our subroutine
53
54                mov  ax, data2
55                push ax                  ; place start of array on stack
56                mov  ax, 20
57                push ax                  ; place element count on stack
58                call bubblesort          ; call our subroutine again
59
60                mov  ax, 0x4c00          ; terminate program
61                int  0x21
```

11	A word gap has been created for swap flag. This is equivalent to a dummy push. The registers are pushed above this gap.
23	The swapflag is accessed with [bp-2]. The parameters are accessed in the same manner as the last examples.
44	We are removing the hole that we created. The hole is removed by restoring the value of SP that it had at the time of snapshot or at the value it had before the local variable was created. This can be replaced with "add sp, 2" however the one used in the code is preferred since it does not require to remember how much space for local variables was allocated in the start. After this operation SP points to the old value of BP from where we can proceed as usual.

Summary: Caller's Responsibilities

1. Push Parameters (if required)
2. Call Subroutine

Summary: Callee's Responsibilities

1. Upon entering the subroutine
   i. Push BP, Copy SP to BP
   ii. Create space for locals (if required)
   iii. Save state (registers)
2. Perform the subroutine task
3. Before leaving the subroutine
   i. Restore state (registers)
   ii. Release space of local variables
   iii. Pop (restore) BP
   iv. Release space of parameters

Stack Frames

- Each subroutine call adds a set of information on stack. The complete set is removed from stack once subroutine is finished.
- For this set, we use the term ‘stack frame’, or ‘activation frame’ or ‘activation record’

DrawSquare subroutine calls DrawLine

Frame Pointer is another name of BP