Unit2: Data and Control transfer operations

Sandip J.Murchite

Data transfer instructions

- 1) Bswap R32
- Example: bswap %eax
- Description : In computer network, the data may arrive in big endian order while the IA 32 processor may need to operate on the data in little endian order.

Open Save

```
1  .section .data
2
3
4  value:
5     .int 0xffddeecc
6
7
8
9
10 .section .text
11
12 .globl _start
13 _start: nop
14
15         movl value,%eax
16         bswap %eax
17
18
19    movl $1,%eax
20    movl $0,%ebx
21    int $0x80
22
```

```
student@student-OptiPlex-3020: ~
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

student@student-OptiPlex-3020:~$ as -gstabs -o abc.o abc.s
student@student-OptiPlex-3020:~$ ld -o abc abc.o
student@student-OptiPlex-3020:~$ gdb -q abc
Reading symbols from abc...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file abc.s, line 15.
(gdb) run
Starting program: /home/student/abc

Breakpoint 1, _start () at abc.s:15
15              movl value,%eax
(gdb) s
16              bswap %eax
(gdb) print/x $eax
$1 = 0xffddeecc
(gdb) s
19          movl $1,%eax
(gdb) print/x $eax
$2 = 0xcceeddff
(gdb)
```

Data transfer instructions

- 2) XCHG SRC,DEST
  Values stored in source register and destination register get exchanged.
- Example: xchg %eax,%ebx
- Before execution : %eax=0xff and %ebx=0xaa
- After execution : %eax=0xaa and %ebx =0xff
- 3) Cmovcc src,dest
- Example: cmovae src,dest
- Description: it move content from source to destination only when destination operand is above or equal than the source.
- Destination operand>= Source operand.
- Source operand can be register or memory, but destination operand can only be register.
- This instruction is used immediately after compare instruction.

Open Save

```
1  .section .data
2
3
4  value:
5     .int 60
6  value1:
7     .int 12
8
9
10
11
12 .section .text
13
14 .globl _start
15 _start: nop
16
17         movl value,%eax
18         movl value1,%ebx
19         cmpl %ebx,%eax
20         cmovae %eax,%ebx
21         xchg %eax,value1
22
23
24    movl $1,%eax
25    movl $0,%ebx
26    int $0x80
27
```


student@student-OptiPlex-3020: ~

```
(gdb) run
Starting program: /home/student/abc

Breakpoint 1, _start () at abc.s:17
17              movl value,%eax
(gdb) s
18              movl value1,%ebx
(gdb) s
19              cmpl %ebx,%eax
(gdb) s
20              cmovae %eax,%ebx
(gdb) s
21              xchg %eax,value1
(gdb) print/d $ebx
$1 = 60
(gdb) x/d &value1
0x804909b:      12
(gdb) s
24          movl $1,%eax
(gdb) x/d &value1
0x804909b:      60
(gdb) print/d $eax
$2 = 12
(gdb)
```

Data transfer instructions

- Machine stack
- Pushl src
- Description: The source operand of push instruction can be an immediate data,register or memory variable.
- Example: Pushl %eax ,pushw %ax
- Popl dest
- Description: the Destination operand of pop instruction can only register or memory variable.
- Example: Popl,%eax , popw %ax

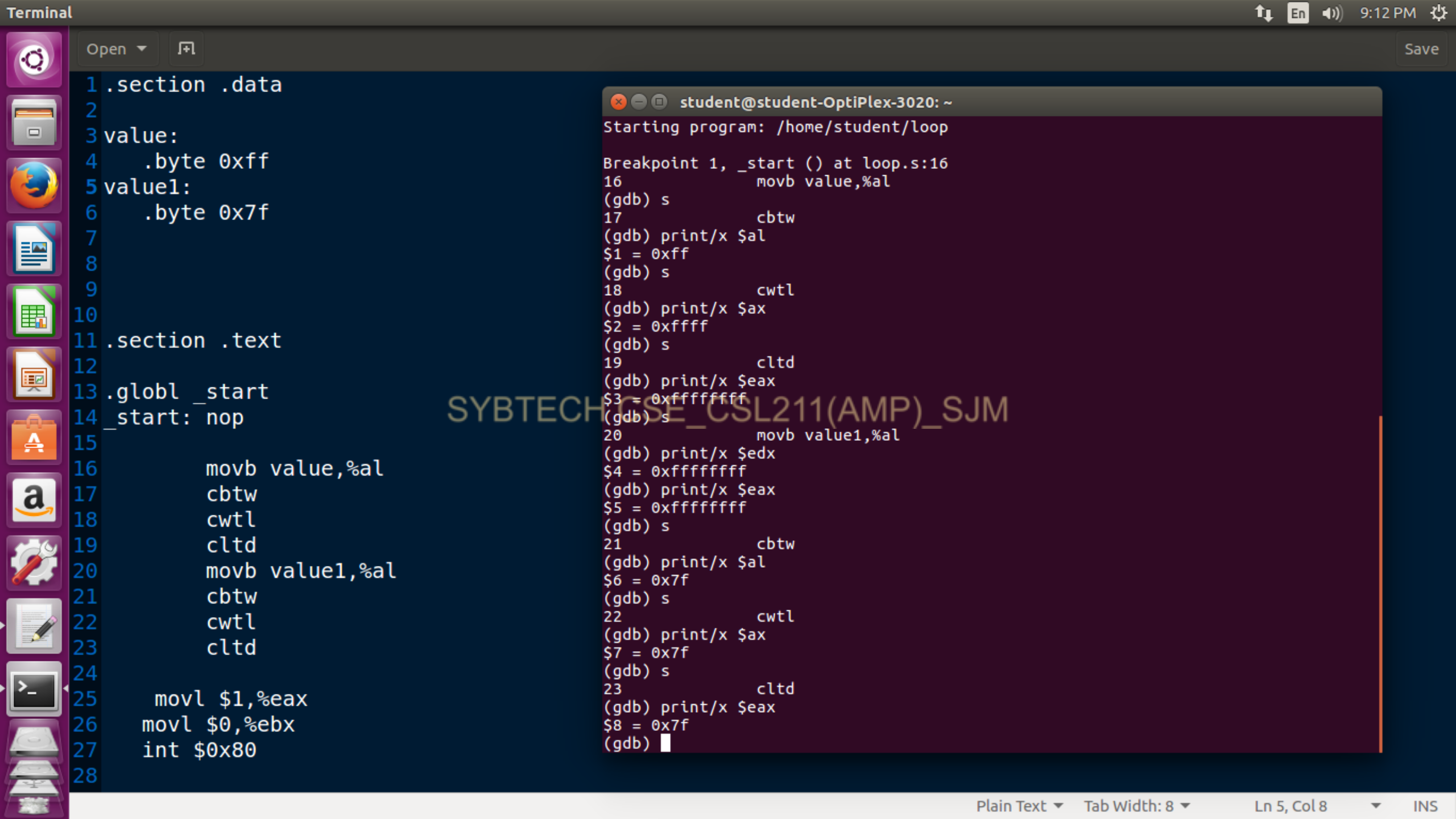
Open Save

```
1  .section .data
2
3
4
5  value:
6      .int 0x11224499
7
8
9  .section .text
10
11 .globl _start
12 _start: nop
13     movl $0xffddeecc,%eax
14     movl $0xaabbccdd,%ebx
15     movl value,%ecx
16     pushl %ebx
17     pushw %cx
18     inc %esp
19     popw %bx
20     popl %edx
21
22
23
24      movl $1,%eax
25      movl $0,%ebx
26      int $0x80
```

```
student@student-OptiPlex-3020: ~
Breakpoint 1, _start () at stack.s:13
13          movl $0xffddeecc,%eax
(gdb) s
14          movl $0xaabbccdd,%ebx
(gdb) s
15          movl value,%ecx
(gdb) s
16          pushl %ebx
(gdb) print/x $esp
$1 = 0xbffff0e0
(gdb) s
17          pushw %cx
(gdb) print/x $esp
$2 = 0xbffff0dc
(gdb) s
18          inc %esp
(gdb) print/x $esp
$3 = 0xbffff0da
(gdb) s
19          popw %bx
(gdb) print/x $esp
$4 = 0xbffff0db
(gdb) s
20          popl %edx
(gdb) print/x $esp
$5 = 0xbffff0dd
(gdb) s
24           movl $1,%eax
(gdb) print/x $esp
$6 = 0xbffff0e1
(gdb)
```

Data transfer instructions

- Handling signed and unsigned numbers
- Double the size of the source operand by means of sign extension
- cbtw
  - The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register.
- Cwtl
  - The CWDE (convert word to double word) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.
- Cltd
  - Input 32 bit number in register eax and the resultant 64 bit number is made available in registers edx and eax.


Terminal
9:12 PM
Open
Save
1 .section .data
2
3 value:
4 .byte 0xff
5 value1:
6 .byte 0x7f
7
8
9
10
11 .section .text
12
13 .globl _start
14 _start: nop
15
16 movb value,%al
17 cbtw
18 cwtl
19 cltd
20 movb value1,%al
21 cbtw
22 cwtl
23 cltd
24
25 movl $1,%eax
26 movl $0,%ebx
27 int $0x80
28
student@student-OptiPlex-3020: ~
Starting program: /home/student/loop
Breakpoint 1, _start () at loop.s:16
16 movb value,%al
(gdb) s
17 cbtw
(gdb) print/x $al
$1 = 0xff
(gdb) s
18 cwtl
(gdb) print/x $ax
$2 = 0xffff
(gdb) s
19 cltd
(gdb) print/x $eax
$3 = 0xffffffff
(gdb) s
20 movb value1,%al
(gdb) print/x $edx
$4 = 0xffffffff
(gdb) print/x $eax
$5 = 0xffffffff
(gdb) s
21 cbtw
(gdb) print/x $al
$6 = 0x7f
(gdb) s
22 cwtl
(gdb) print/x $ax
$7 = 0x7f
(gdb) s
23 cltd
(gdb) print/x $eax
$8 = 0x7f
(gdb)
SYBTECH_BSE_CSL211(AMP)_SJM
Plain Text
Tab Width: 8
Ln 5, Col 8
INS

Data Transfer instructions

- Direct addressing mode
- base address (offset address, index, size)

- values:
- .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
- Example: To reference the value 20 from the values array shown, you would use the following
- instructions:
- Movl $2,%edi
- Movl values(,%edi,4)

Data transfer instruction

- indirect addressing with registers
- you can get the memory location address of the data value by placing a dollar sign ($) in front of the label in the instruction.
- movl $values, %esi

- This instruction places the value contained in the memory location 8 bytes after the location pointed to by the esi register to register eax register.
- Movl 8(%esi),%eax

Open Save

```
1  .section .data
2
3
4
5   value:
6     .int 110,11,33,44,550
7
8  .section .text
9
10 .globl _start
11 _start: nop
12
13     movl $1,%edi
14 loop: movl value(,%edi,4),%ebx
15     movl $value,%esi
16     movl 8(%esi),%eax
17     movl $0,16(%esi)
18
19    movl $1,%eax
20    movl $0,%ebx
21    int $0x80
22
```


student@student-OptiPlex-3020: ~

```
Breakpoint 1 at 0x8048075: file large.s, line 13.
(gdb) run
Starting program: /home/student/large

Breakpoint 1, _start () at large.s:13
13          movl $1,%edi
(gdb) s
14      loop: movl value(,%edi,4),%ebx
(gdb) s
15          movl $value,%esi
(gdb) print/d $ebx
$1 = 11
(gdb) s
16          movl 8(%esi),%eax
(gdb) s
17          movl $0,16(%esi)
(gdb) print/d $eax
$2 = 33
(gdb) s
19          movl $1,%eax
(gdb) x/5d &value
0x804909c:      110     11      33      44
0x80490ac:      0
(gdb)
```

Part II: Control Transfer Instructions

- IA 32 processors, the target address in a control transfer instruction can be specified in one of the following ways.
  1. Immediate constant
     - Jmp swapbyte
     - Jmp 0x100
     - Target address = A+0x100H where A= address of the next instruction.
     - Absolute address: jmp *0x100
     - Target address=0x100h

Part II: Control Transfer Instructions

- IA 32 processors, the target address in a control transfer instruction can be specified in one of the following ways.

2. ## Register addressing

   - Jmp %eax

   (Target address may be specified using a register)

2. ## Memory addressing mode:

   the address of the target instruction is read from memory.

   Jmp (%eax)

- Address of target instruction is stored in memory location whose address is available in register eax

Open
Save

```
1
2
3
4
5
6
7
8
9  .section .text
10
11 .globl _start
12 _start: nop
13
14         movl $1,%eax
15         jmp end
16         movl $30,%ebx
17          int $0x80
18
19
20   end:  movl $20,%ebx
21         int $0x80
22
```

```
student@student-OptiPlex-3020: ~
student@student-OptiPlex-3020:~$ as -gstabs -o abc.o abc.s
student@student-OptiPlex-3020:~$ ld -o abc abc.o
student@student-OptiPlex-3020:~$ ./abc
student@student-OptiPlex-3020:~$ echo $?
20
student@student-OptiPlex-3020:~$ objdump -D abc

abc:     file format elf32-i386


Disassembly of section .text:

08048054 <_start>:
 8048054:       90                      nop
 8048055:       b8 01 00 00 00          mov    $0x1,%eax
 804805a:       eb 07                   jmp    8048063 <end>
 804805c:       bb 1e 00 00 00          mov    $0x1e,%ebx
 8048061:       cd 80                   int    $0x80

08048063 <end>:
 8048063:       bb 14 00 00 00          mov    $0x14,%ebx
 8048068:       cd 80                   int    $0x80

Disassembly of section .stab:
```

Conditional Control transfer instructions

- IA32 architectures support conditional control transfer instruction
- jcc target
- Examples: je, jz,jne,ja,jnbe,jnc,jc

Consider a scenario where we need to compute fuction f(x)

F(x)=(X+X) for X>2

(X+3) for X<=2

- Assembly code for Function f(X)

Assume value of X=5 which is loaded in register %ebx

```
Movl $5,%ebx                    %ebx=5
Movl $3,%eax                    %eax=3
Cmpl $2,%ebx                    %ebx-2
Jle next
Movl %ebx,%eax
Next: addl %ebx,%eax
```

80386 ALP to evaluates the sum of first n integer by iteration

Open | Save

```
1  .section .data
2
3
4  value:
5     .int 4
6
7
8
9
10 .section .text
11
12 .globl _start
13 _start: nop
14         movl $0,%eax
15         movl value,%ecx
16
17   back: addl %ecx,%eax
18         loop back
19
20
21      movl $1,%eax
22     movl $0,%ebx
23     int $0x80
24
```

```
student@student-OptiPlex-3020: ~
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file loop.s, line 14.
(gdb) run
Starting program: /home/student/loop

Breakpoint 1, _start () at loop.s:14
14              movl $0,%eax
(gdb) s
15              movl value,%ecx
(gdb) s
17          back: addl %ecx,%eax
(gdb) s
18              loop back
(gdb) s
17          back: addl %ecx,%eax
(gdb) s
18              loop back
(gdb) s
17          back: addl %ecx,%eax
(gdb) s
18              loop back
(gdb) s
17          back: addl %ecx,%eax
(gdb) s
18              loop back
(gdb) s
21            movl $1,%eax
(gdb) print/d $eax
$1 = 10
(gdb)
```

Function call and Return

- Functions are subprograms to which the control of execution is passed temporarily. After execution of the called function is completed , the control is transferred back to the calling program.
- The most important reason is to reuse the code
- Example: if in a program several strings are to be displayed on the screen at different time , it is better to write a function to display one string. Each time a string is to be displayed , this function can be called.
- Writing function in a program is to make it readable and understandable.

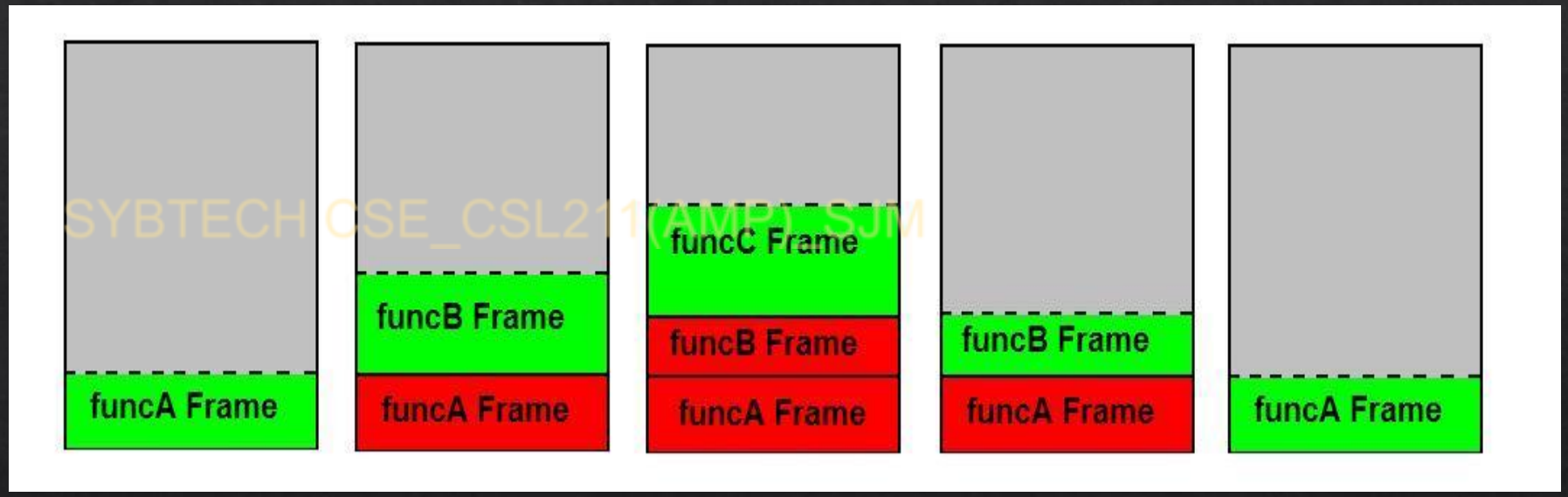
```
        CALL FUNCTION
0X100C  PUSH %EAX
0X1100  FUNCTION:
```


- During execution of CALL instruction ,address of next instruction is saved on the stack. Stack pointer decremented by 4

- The register EIP changed to 0x00001100,thereby control transfer to function **FUNCTION**
- IA32 architectures provide an instruction called RET to return control to the caller after execution of the called function is completed.
- The RET instruction pops a 32 bit offset from top of the stack into EIP register, thereby causing a control transfer.

Stack frame

- Each function call results in a creation of a stack frame.



Function prologue and epilogue

```
Function:
pushl %ebp
movl %esp, %ebp
.
.
.
.

movl %ebp, %esp
popl %ebp
ret
```

- The first two instructions at the top of the function code

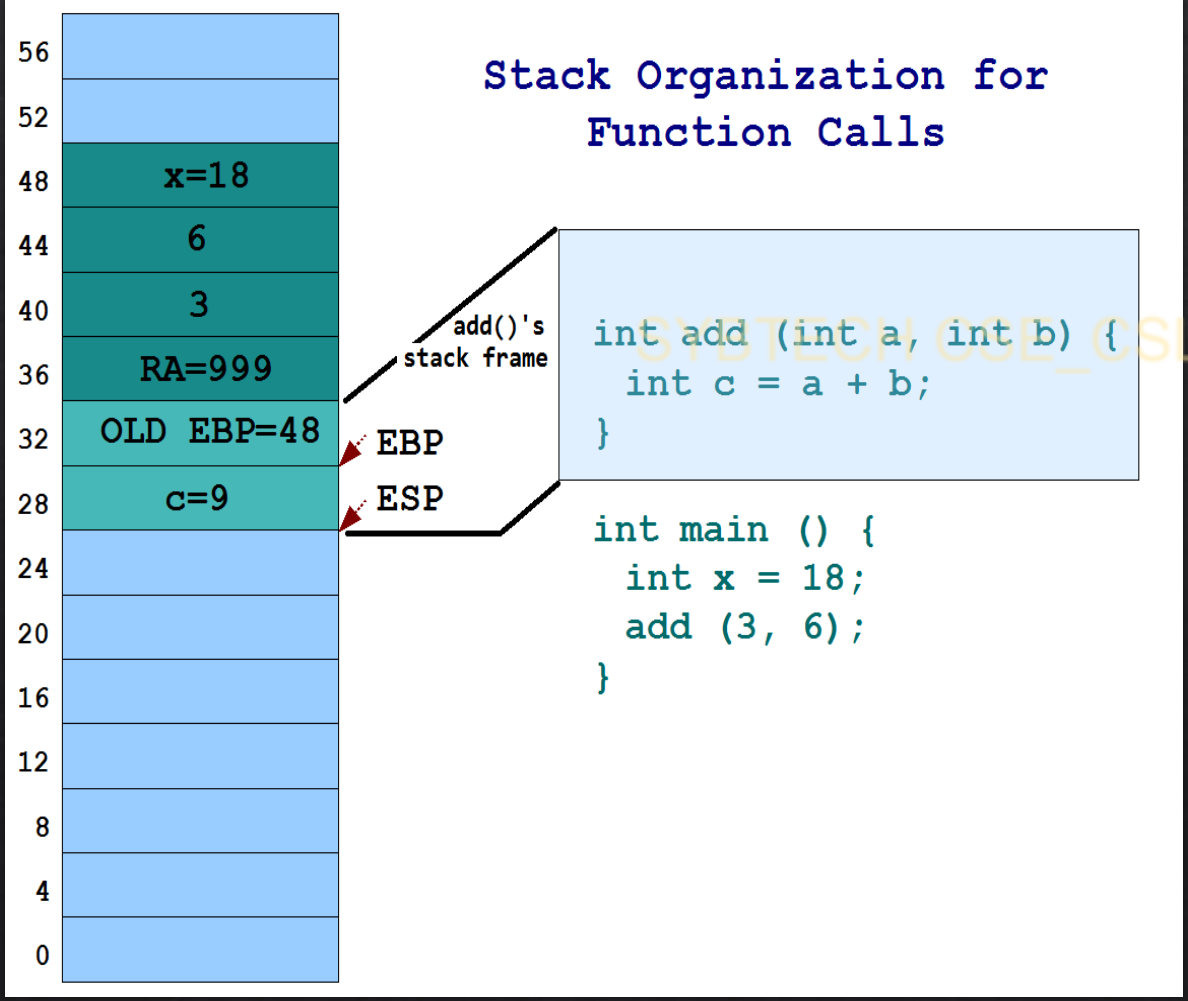
1) save the original value of EBP to the top of the stack
2) copy the current ESP stack pointer to the EBP register.

- After the function processing completes

- the last two instructions in the function

1. Retrieve the original value in the ESP register that was stored in the EBP register.
2. Restore the original EBP register value.

Stack organization for function call



Function: Pushl %ebp
movl %esp,%ebp
....

.....

Movl %ebp,%esp
Pop %ebp
Ret

In GNU environment, ebp is used to store address of current frame.
When function is called, it must save frame pointer and must restore it before returning the control to the caller.

Assembly language program on Function call

```
.section .data
Output:
    .asciz "This is section %d\n"
.section .text
.globl _start
_start: pushl $1
          pushl $output
          call printf
          addl $8,%esp
         call function
          pushl $3
          pushl $output
          call printf
          addl $8,%esp
         movl $1,%eax
         movl $0,%ebx
         int $0x80
```

```
Function:  pushl %ebp
           movl %esp,%ebp
```

Entry Code

```
pushl $2
pushl $output
call printf
addl $8,%esp
```

```
Movl %ebp,%esp
Popl %ebp
   ret
```

Exit Code

```
This is section 1
This is section 2
This is section 3
```

Open Save

```
4     .asciz "This is section %d\n"
5
6
7  .section .text
8
9  .globl _start
10 _start: nop
11         pushl $1
12         pushl $output
13         call printf
14         addl $8,%esp
15         call overhere
16         pushl $3
17         pushl $output
18         call printf
19         addl $8,%esp
20         movl $1,%eax
21         movl $0,%ebx
22         int $0x80
23 overhere: pushl %ebp
24          movl %esp,%ebp
25         pushl $2
26         pushl $output
27         call printf
28         addl $8,%esp
29         movl %ebp,%esp
30         popl %ebp
31         ret
```

```
student@student-OptiPlex-3020: ~
student@student-OptiPlex-3020:~$ clear
student@student-OptiPlex-3020:~$ as -gstabs -o loop.o loop.sstudent@student-OptiPlex-3020:~$ ld -dynamic
-linker /lib/ld-linux.so.2 -lc -o loop loop.o
student@student-OptiPlex-3020:~$ ./loop
This is section 1
This is section 2
This is section 3
student@student-OptiPlex-3020:~$
```

Parameter Passing Conventions

- There are two parameter passing conventions that are used by most of programming languages.
- Parameter passing by value
- **Drawback**: If called function changes the value of the parameter, the original value remains unchanged and only copy on stack is changed.
- Parameter passing by address or reference
- It provide mechanism to pass addresses of the parameters (instead of their values) to the called function.

Parameter passing by Address OR Reference