Unit3

Data and Control transfer instructions

Data Movement instructions

movx src, dest

xchg

- It can exchange data values between two general purpose registers, or between a register and a memory location.

- Format of the instruction is as follows:

  xchg operand1(src), operand2(dest)

- Either operand1 or operand2 can be a general-purpose register or a memory location (but both cannot be a memory location).

- Two operands must be the same size. i.e. 8-, 16-, or 32-bit

xchg

- None of the operand can be specified using immediate addressing.

bswap

- The BSWAP instruction reverses the order of the bytes in a register.

- Bits 0 through 7 are swapped with bits 24 through 31, while bits 8 through 15 are swapped with bits 16 through 23.

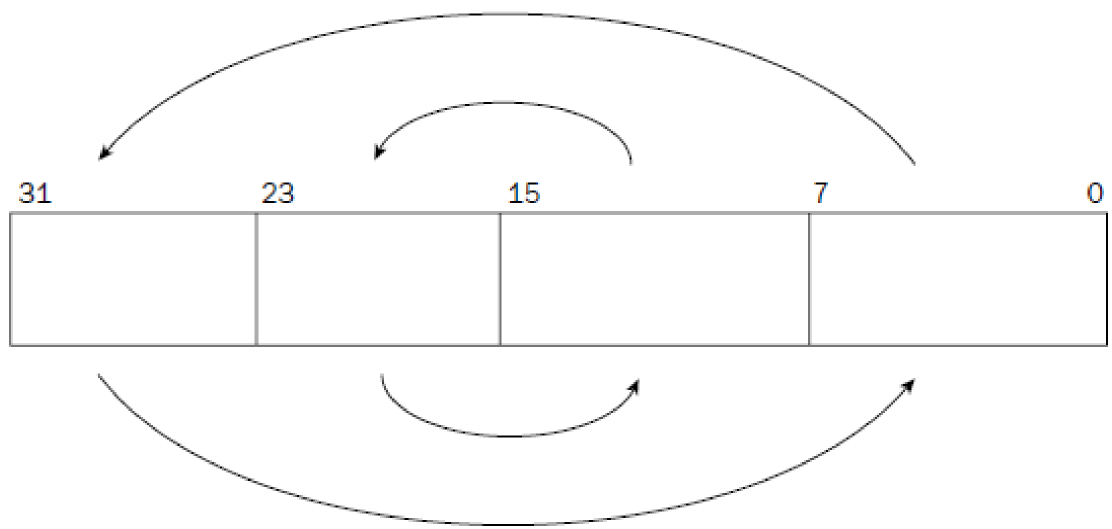


Figure 5-3

bswap

- bits are not reversed; but rather, the individual bytes contained within the register are reversed.

```
movl $0x12345678, %ebx
bswap %ebx
```

XADD

- xadd is used to exchange the values between two registers, or a memory location and a register, add the values, and then store them in the destination location (either a register or a memory location).

- format of the XADD instruction is

  xadd source, destination

CMPXCHG

- The CMPXCHG instruction compares the destination operand with the value in the EAX, AX, or AL registers.
- If the values are equal, the value of the source operand value is loaded into the destination operand.
- If the values are not equal, the destination operand value is loaded into the EAX, AX, or AL registers.
- The
- format of the CMPXCHG instruction is

cmpxchg source, destination

example of the cmpxchg instruction

```
.section .data
    data:
        .int 10
.section .text
    .globl _start
        _start:
        nop
        movl $10, %eax
        movl $5, %ebx
        cmpxchg %ebx, data
        movl $1, %eax
        int $0x80
```

Result:

```
11 cmpxchg %ebx, data
(gdb) x/d &data
0x8049090 <data>: 10
(gdb) s
12 movl $1, %eax
(gdb) x/d &data
0x8049090 <data>: 5
(gdb)
```

```
movl $105, %ebx
movl $235, %eax
cmp %ebx, %eax            ------------eax-ebx
cmova %eax, %ebx
```

```
15   cmp %ebx, %eax
    (gdb) print $eax
$1 = 235
     (gdb) print $ebx
$2 = 105
(gdb) s
16 cmova %eax, %ebx
(gdb) s
(gdb) print $ebx
$3 = 235
(gdb)
```

Data movement with conditional move instruction

cmovx/cc src , dest

Instruction Pair	Description	EFLAGS Condition
CMOVA/CMOVNBE	Above/not below or equal	(CF or ZF) = 0
CMOVAE/CMOVNB	Above or equal/not below	CF=0
CMOVNC	Not carry	CF=0
CMOVB/CMOVNAE	Below/not above or equal	CF=1
CMOVC	Carry	CF=1
CMOVBE/CMOVNA	Below or equal/not above	(CF or ZF) = 1
CMOVE/CMOVZ	Equal/zero	ZF=1
CMOVNE/CMOVNZ	Not equal/not zero	ZF=0
CMOVP/CMOVPE	Parity/parity even	PF=1
CMOVNP/CMOVPO	Not parity/parity odd	PF=0

Fig unsigned conditional move instructions

Data movement with conditional move instruction

Instruction Pair	Description	EFLAGS Condition
CMOVGE/CMOVNL	Greater or equal/not less	(SF,OF)=0
CMOVL/CMOVNGE	Less/not greater or equal	(SF,OF)=1
CMOVLE/CMOVNG	Less or equal/not greater	SF,OF or ZF =1
CMOVO	Overflow	OF=1
CMOVNO	Not overflow	OF=0
CMOVS	Sign (negative)	SF=1
CMOVNS	Not sign (non-negative)	SF=0
cmovg/cmovnle	Greater/not less or equal	ZF=0 and SF=OF

Fig Signed conditional move instructions

```
.section .data
        value:
                .int 5
.section .text
        .globl _start
        _start:
        nop
        movl  $3, %ebx
        movl  value, %ecx

        cmp  %ebx, %ecx
        cmova  %ecx, %ebx

        movl  $7,%edx
        cmp  %edx, %ecx
        cmova  %ecx, %edx

        movl  $0, %ebx
        movl  $1, %eax
        int  $0x80
```

```
.section .data
    data:
        .int  5
.section .text
        .globl  _start
            _start:
            nop
            movl  $0, %eax
            movl  $-1, %ebx

            cmpl  $2, data
            cmova  %ebx,  %eax

            cmpl  $7 , data
            cmovb  %ebx, %eax

            movl  $0, %ebx
            movl  $1, %eax
            int  $0x80
```

- **EBX: The integer file descriptor**
- **ECX: The pointer (memory address) of the string to display**
- **EDX: The size of the string to display**

- **0 (STDIN): The standard input for the terminal device (normally the keyboard)**
- **1 (STDOUT): The standard output for the terminal device (normally the terminal screen)**
- **2 (STDERR): The standard error output for the terminal device (normally the terminal screen)**

Machine stack:

- Stack is a data structure in which data items are added and removed in the LIFO order.

e.g.

esp=0x000011C0

eax=0x13579BDF

PUSH eax

Machine stack:

- Machine stack can be used in several ways:
  - To keep the return address for the function calls
  - To temporarily save the contents of registers and recover them.
  - Local variables of high level language function
  - Execution control for recursive functions is implemented using stack.

```
.section .data
data:
      .int 125
      .section .text
.globl _start
            _start:
            Nop
            movl $24420, %ecx
            movw $350, %bx
            movb $100, %eax
            pushl %ecx
            pushw %bx
            pushl %eax
            pushl data
            pushl $data              #puts the 32-bit memory address referenced by the data label
            popl %eax
            popl %eax
            popl %eax
            popw %ax
            popl %eax
            movl $0, %ebx
            movl $1, %eax
            int $0x80
```

```
.section .data
    data:
    .int  125
.section .text
.globl _start
        _start:
        nop
        movw  $0x1122, %ax
        movl  $0xaabbccdd, %edx
        movb  $0x88, %bl
        pushw  %ax
        pushl  %edx
        inc  %esp
        popl  %ebx
        dec  %esp
        popw  %cx
        movl $0,  %ebx
        movl $1,  %eax
        int  $0x80
```

Control transfer instructions:

- Target address can be specified by --

1. **Immediate constant**
   - Relative addressing

     JMP swapbyte(label)

     JMP 0x100 ----target address= a+0x100

   Where a = address of next instruction immediately after jmp.

   0x100= offset from the next instruction of jmp

2. Register addressing:
   - Target address of control instruction may be specified using the register.

     JMP %eax

3. Memory addressing:

- Address of target instruction is read from memory.

  JMP (memvar)

- 32 bit memory address is stored in **memvar.**
- During execution 32 bit value is read from memory location **memvar** and copied to **eip register** and transfer the control

- Unconditional control transfer instruction:

  JMP target

- Conditional control transfer instruction:

  J<u>CC</u> target

Control transfer instructions:

Instruction	Description	EFLAGS
JA	Jump if above	CF=0 and ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX register is 0	
JECXZ	Jump if ECX register is 0	
JE	Jump if equal	ZF=1
JG	Jump if greater	ZF=0 and SF=OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 or SF<>OF
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=0 and ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater	ZF=1 or SF<>OF

Control transfer instructions continued….:

JNGE	Jump if not greater or equal	SF<>OF
JNL	Jump if not less	SF=OF
JNLE	Jump if not less or equal	ZF=0 and SF=OF
JNO	Jump if not overflow	OF=0
JNP	Jump if not parity	PF=0
JNS	Jump if not sign	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if parity odd	PF=0
JS	Jump if sign	SF=1
JZ	Jump if zero	ZF=1

#Example of using the CMP and JGE instructions

```
.section .text
.globl _start
_start:
    nop
    movl $15, %eax
    movl $20, %ebx
    cmp %eax, %ebx
    jge  next
    movl $1, %eax
    int $0x80
    next:
    movl $10, %ebx
    movl $1, %eax
    int $0x80
```

Building loops in program:

- Loop sections in the program that are iterated over again and again.
- Loops have 3 main parts-
  - Initialization code
  - Condition testing code
  - Body

- WAP to perform sum of all integers from 1 to n.

The loop instructions:

- **Loop:**
  - Loops are another way of altering the instruction path within the program.
  - Loops enable us to code repetitive tasks with a single loop function.
  - The loop operations are performed repeatedly until a specific condition is met.

Instruction	Description
LOOP	Loop until the ECX register is zero
LOOPE/LOOPZ	Loop until either the ECX register is zero / ZF is SET
LOOPNE/LOOPNZ	Loop until either the ECX register is zero / ZF is NOT SET

Using indexed memory locations

values:

.int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

- When referencing data in the array, we must use an index system to determine which value we are accessing.

- The memory location is determined by the following:
  – A base address
  – An offset address to add to the base address
  – The size of the data element
  – An index to determine which data element to select

The format of the expression is-

base_address(offset_address, index, size)

- To reference the value 20 from the values array shown, use----

```
movl $2, %edi
movl values(, %edi, 4), %eax
```

- This instruction loads the **second index value (3rd value)** of 4 bytes from the values label to the EAX register

- ***Using indirect addressing with registers:***
  - The memory location address of the data value by placing a **dollar sign ($)** in front of the label in the instruction.

```
movl $values, %edi
```

  - Is used to move the memory address the values label references to the EDI register.

```
movl %ebx, (%edi)
```

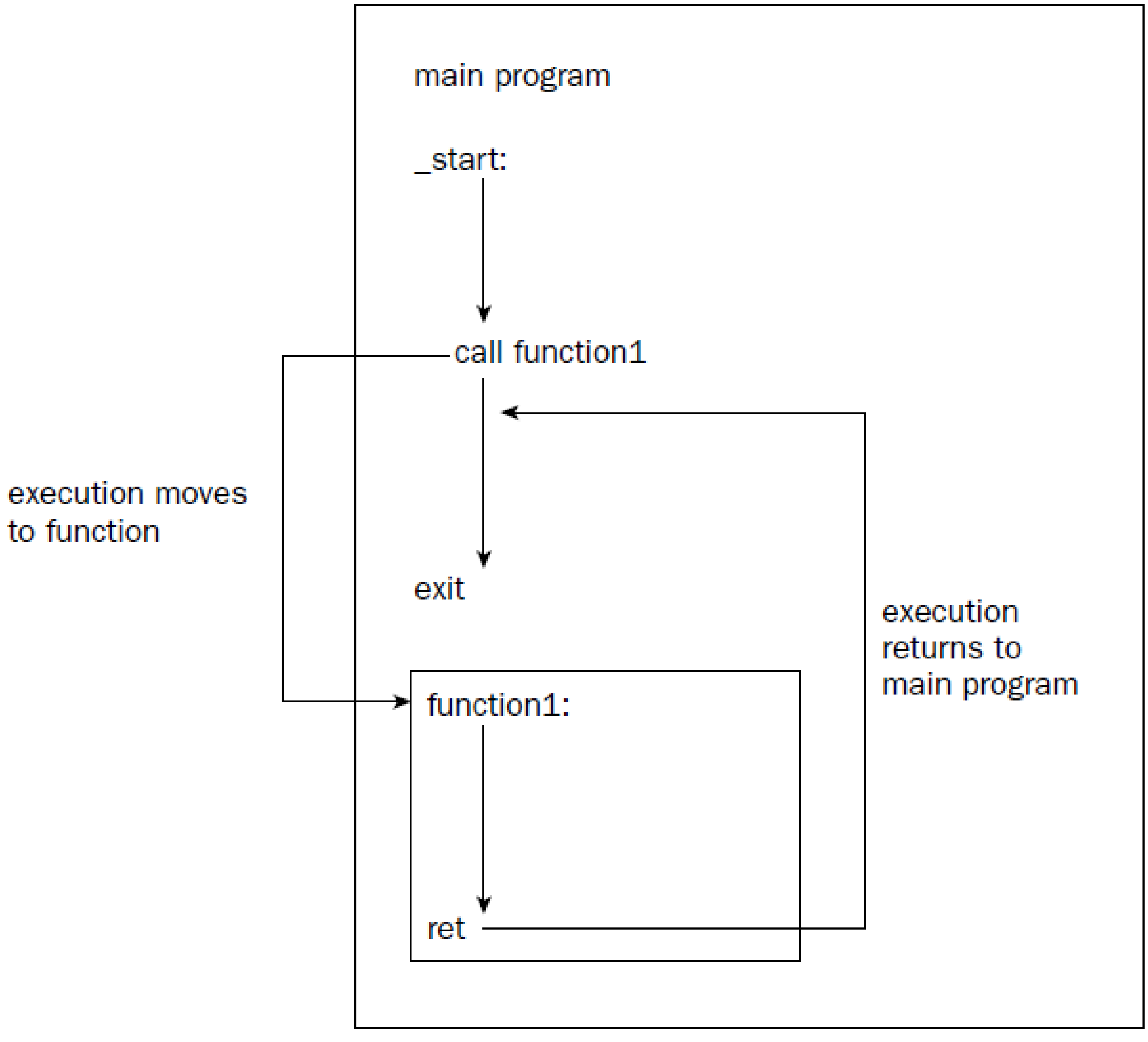
  - moves the value in the EBX register to the memory location contained in the EDI register.

movl %edx, 4(%edi)

– This instruction places the value contained in the EDX register in the memory location 4 bytes **after** the location pointed to by the EDI register.

movl %edx, -4(&edi)

– This instruction places the value in the memory location 4 bytes before the location pointed to by the EDI register.


main program
_start:
call function1
exit
execution moves
to function
function1:
ret
execution
returns to
main program

Function call standard template:

```
function_label:
pushl %ebp
movl %esp, %ebp
        .
        .
        .
        .
        .
movl %ebp, %esp
popl %ebp
ret
```

#4 Assembly language program to Demonstrate Function calls and return.

```
.section .data
output:
    .asciz "This is section %d\n"

.section .text
.globl _start
    _start:
    pushl $1
    pushl $output
    call printf
    add $8, %esp                    # should clear up stack
    call overhere
    pushl $3
    pushl $output
    call printf
```

```
add $8, %esp                    # should clear up stack
pushl $0
call exit
overhere:
      pushl %ebp
      movl %esp, %ebp
      pushl $2
      pushl $output
call printf
add $8, %esp                    # should clear up stack
movl %ebp, %esp
popl %ebp
ret
```

```
as -gstabs –o  test1.o test1.s
ld -dynamic-linker /lib/ld-linux.so.2 -lc -o test1 test1.o
./test1
```

Output:

This is section 1

This is section 2

This is section 3

- ***Passing function parameters on the stack***
  - Before a function call is made, the main program places the required input parameters for the function on the **top of the stack**.

  - When the CALL instruction is executed, it places the **return address from the calling program onto the top of the stack** as well, so the function knows where to return.

Passing parameters:

- Passing parameters through registers:

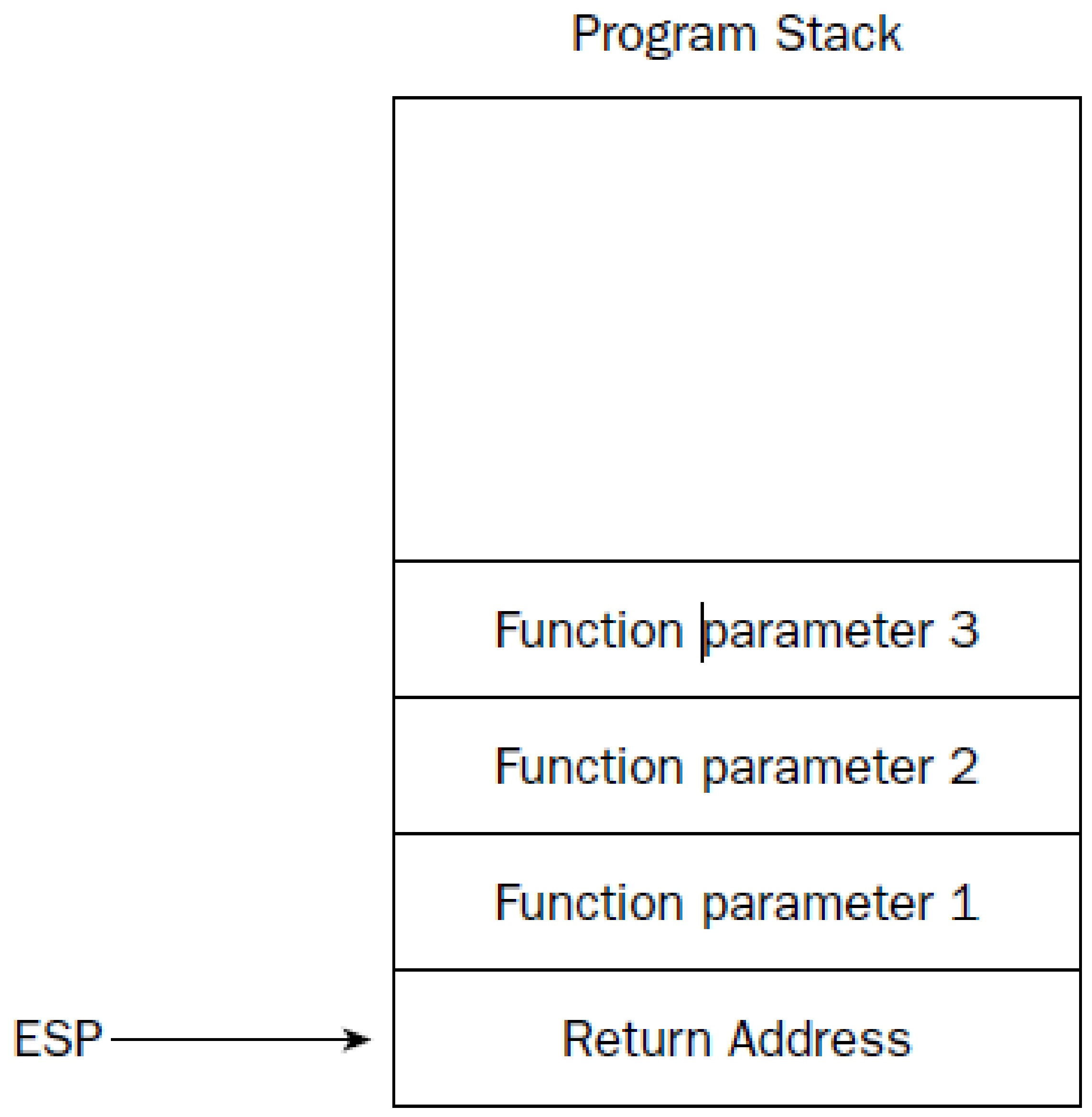
```
#function that converts an 8 bit signed no to 32 bit signed number

Convert:
            movsx  %al ,%ebx
            ret
```

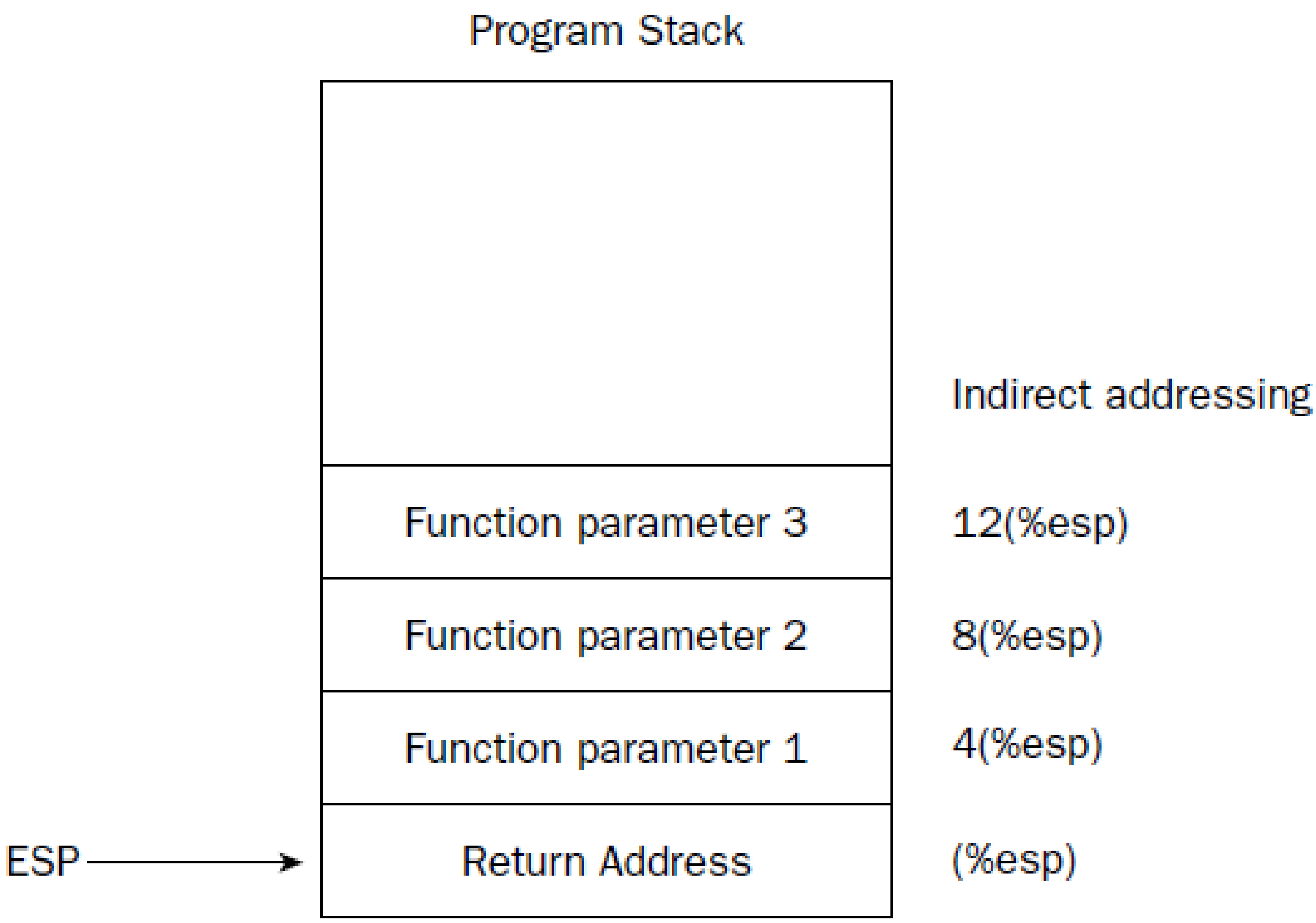
```
#use the function with parameter passing using registers

            mov  memvar8  ,%al
            call convert
            mov  %ebx, memvar32
```

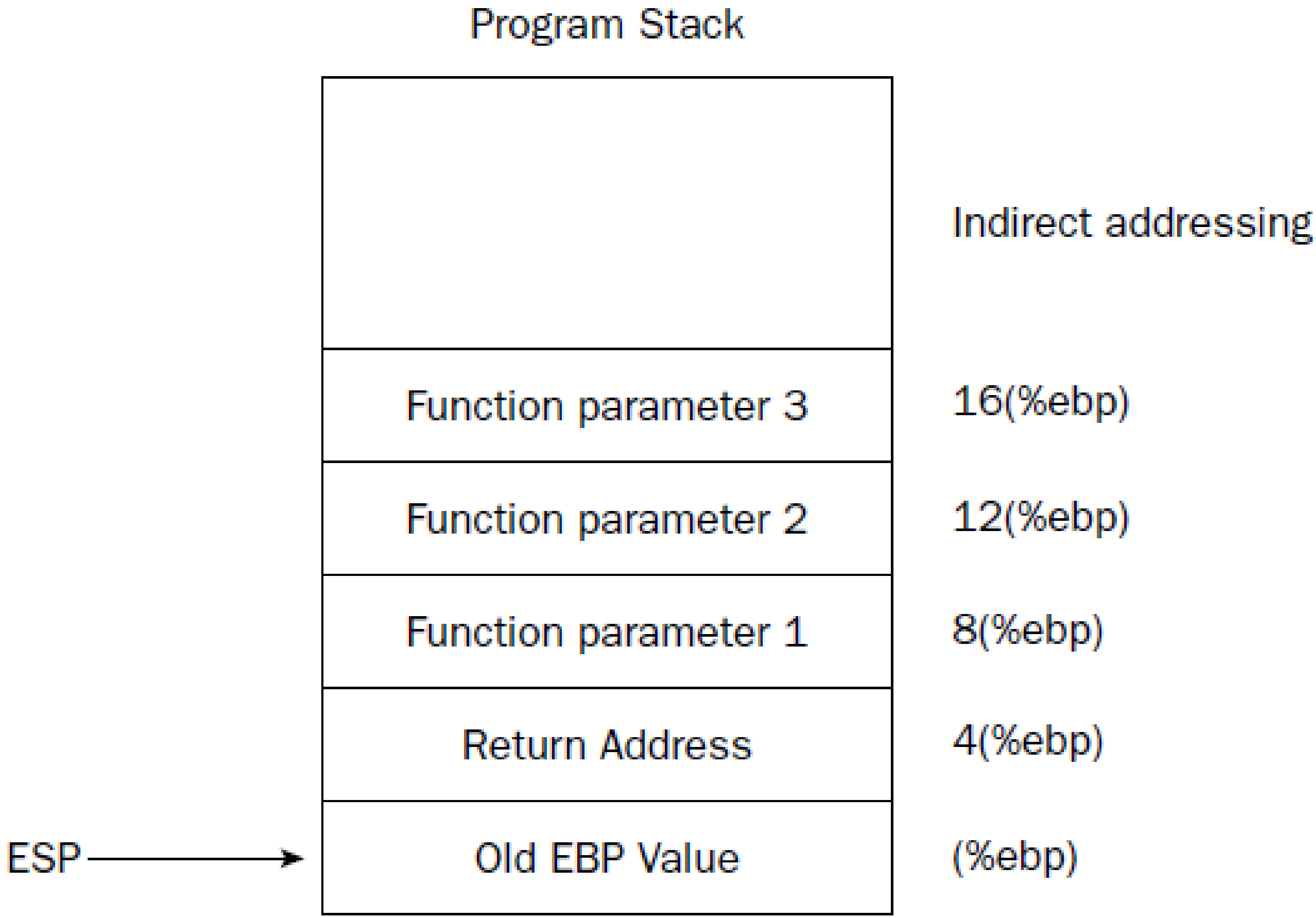
Parameter Passing conventions:

- Parameters passing by value:
- Parameters passing by address or reference:

- Parameters are passed to the called function by copying their values to registers, memory or stack.
- If the called function changes the value of the parameter , the original value remains unchanged and only the copy is changed.
- In such a case parameters are said to be **passed by value**


Program Stack
Function parameter 3
Function parameter 2
Function parameter 1
Return Address
ESP

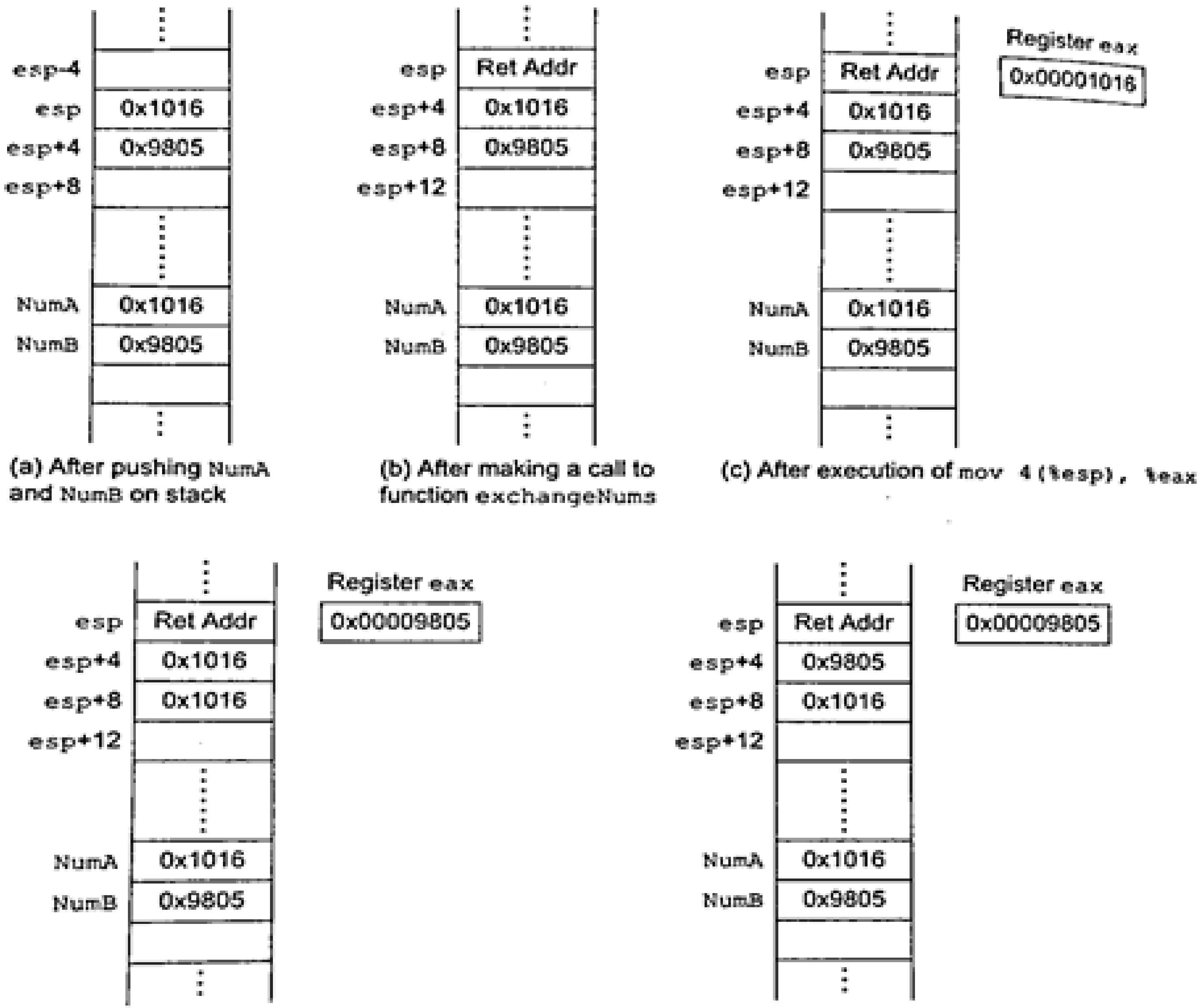

Program Stack
Indirect addressing
Function parameter 3
12(%esp)
Function parameter 2
8(%esp)
Function parameter 1
4(%esp)
ESP
Return Address
(%esp)



EBP register is often used as a base pointer to the stack

```
        .
        .
Pushl NumB
Pushl NumA
Call  exchangeNums
Addl $8, %esp
        .
        .
```

```
exchangeNums:
        mov  4(%esp), %eax
        xchg  %eax, 8(%esp)
        mov  %eax, 4(%esp)
        ret
```



(a) After pushing NumA and NumB on stack

(b) After making a call to function exchangeNums

(c) After execution of `mov 4(%esp), %eax`

(d) After execution of `xchg %eax, 8(%esp)`

(e) After execution of `mov %eax, 4(%esp)`

Figure : Memory contents while executing exchangeNums.

Parameter passing by address or reference:

```
.
.
Pushl  $NumB
Pushl  $NumA
Call  exchangeNums
Addl $8, %esp
.
.
```

```
exchangeNums:
        mov  8(%esp), %eax      #addresses of two 32 bit nos as parameters
        mov  12(%esp), %ebx
        xchg  (%eax), %ecx
        xchg  (%ebx), %ecx
        xchg  (%eax), %ecx
        ret
```

#program for exchanging 2, 32 bit numbers through parameter passing by address /reference

```
.section .data
numA:
        .int 0xAABBCCDD
numB:
        .int 0x11223344
output:
        .asciz "The exchanged values are numA=%d ,numB=%d\n"
.section .text.
globl _start
_start:
        nop
        pushl $numB
        pushl $numA
        call exchnum
        addl $8,%esp
        movl $0, %ebx
        movl $1, %eax
        int $0x80
```

```
exchnum:
        pushl %ebp
        mov %esp, %ebp
        movl 8(%esp),%eax
        movl 12(%esp),%ebx
        xchg (%eax),%ecx
        xchg (%ebx),%ecx
        xchg (%eax),%ecx
        pushl (%ebx)
        pushl (%eax)
        pushl $output
        call printf
        add $12,%esp
        movl %ebp,%esp
        pop %ebp
        ret
```

Interfacing with GNU C compilers;

- ALP can be interfaced to C functions

- Programs written in AL functions can call C functions and vice versa.

- Functions maintain a frame on stack that contains parameters, return address, reference to older frame and local operands.

- Parameters and return addresses are put on stack by the **caller** (by using push and call instructions)

Interfacing with GNU C compilers;

- reference to older frame and space for local variables are maintained by the called function.
- Use frame pointer instead of stack pointer
- ebp register as the frame pointer
- In ALP, the **function prologue** is a few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
- **function epilogue** appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.

Interfacing with GNU C compilers;

- Prologue code for functions:

```
Pushl  %ebp
movl %esp, %ebp
        .
        .
```

- ebp was set to stack pointer before creating space for local variables.

- Old frame pointer can be restored by popping off the stack.

Interfacing with GNU C compilers;

- Epilogue code for functions:

```
movl  %ebp, %esp
Popl  %ebp
ret
```

Thank You