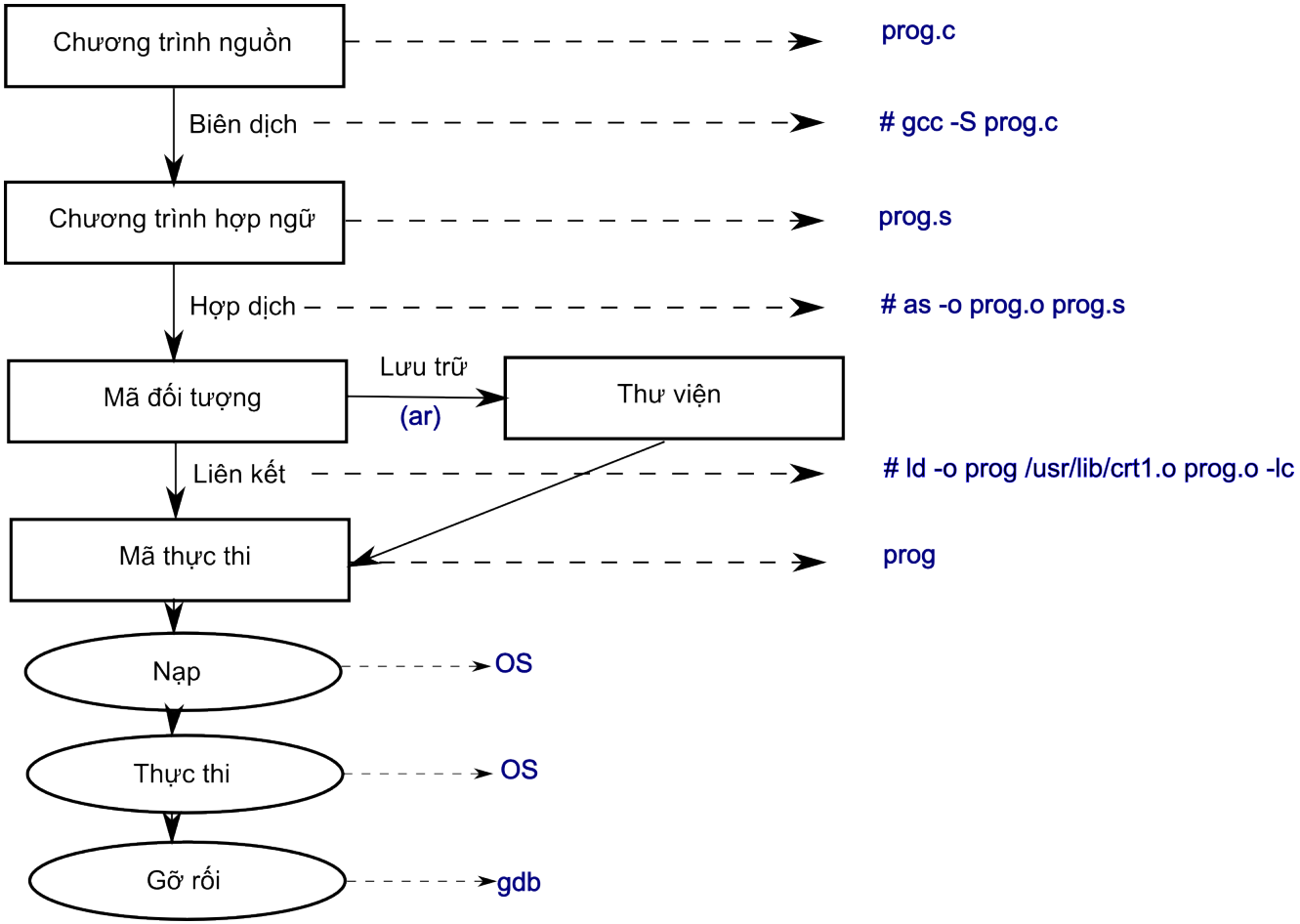
SYSTEM SOFTWARE
INTEL PROCESSOR AND GNU ASSEMBLER

Nguyen Huu Duc

School of Information and Communication Technology
Hanoi University of Science and Technology

Application software: Building and Execution



Objective and Content

- Objectives:
  - Learn the role of system software in building application software.
  - Learn the relationship between system softwares.
- Content
  - Architecture of Intel x86-32bit processor
  - Assembly language and GNU assembler.
  - Function calling conventions.

Hello world!

C source code- prog1.c

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("Hello world!\n");
6: }
```

- Where a program starts execution - function `main`
  - No parameter, value returned
- Function `printf` is not defined in the program
  - There is one parameter.
- No variable

Hello world!

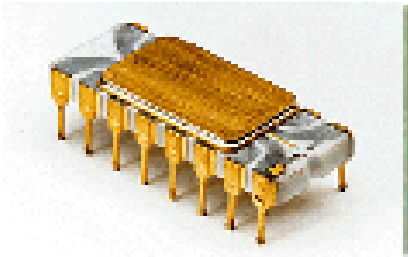
- Compiling to object file
  - `#gcc -mpreferred-stack-boundary=2 -c prog1.c`
- Compiling and linking to executable code
  - `#gcc -mpreferred-stack-boundary=2 -o prog1 prog1.c`
- Compiling to assembly language
  - `#gcc -mpreferred-stack-boundary=2 -S prog1.c`

Hello world!

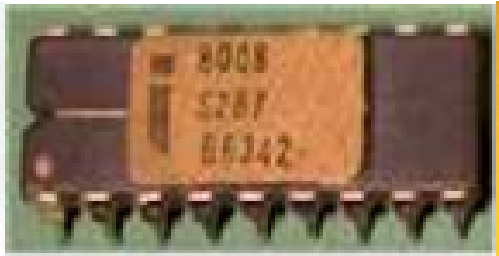
Assembly language - prog1.s

```
01: .file   "prog1.c"                       // Debug information
02:         .section        .rodata         // Start read-only data area
03: .LC0:                                   // Label for a constant of string
04:         .string "Hello world!\n"        // Allocate memory for a string
05:         .text                           // Start of program code
06:         .p2align 2,,3                   // Align memory
07: .globl main                             // Declare the global symbol for main
08:         .type   main, @function         // Declare type for main
09: main:                                   // main label
10:         pushl   %ebp                    // Prepare stack for function calling of
11:         movl    %esp, %ebp
12:         pushl   $.LC0                   // Pass argument to function printf
13:         call    printf                  // Call function printf
14:         addl    $4, %esp                // Release stackframe
15:         leave                           // End of function main
16:         ret
17:         .size   main, .-main            // Define size of funtion main
18:         .ident  "GCC: (GNU) 3.4.6 [FreeBSD] 20060305
```

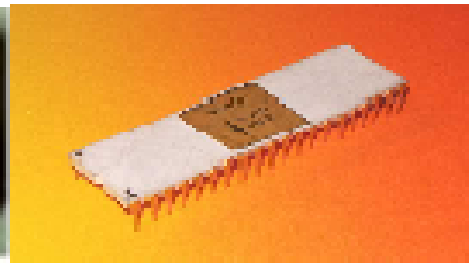
Intel processors(1)



4004

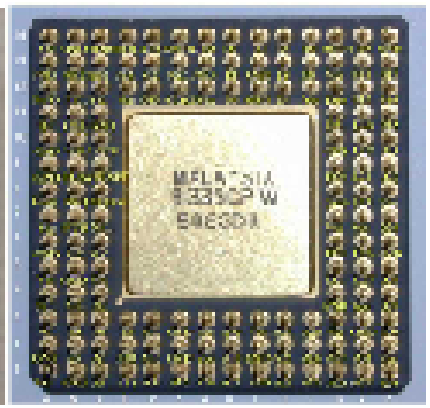


8008

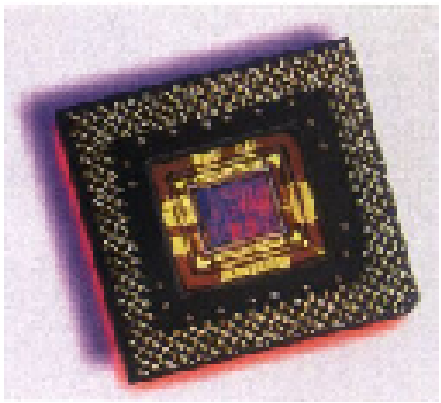


8080

80286



80386



Pentium

Intel processors (2)

- 1979: 8086 processor
  - All internal registers as well as internal and external data buses were 16 bits wide. 20-bit external address.
  - Real-address mode
  - 8087: coprocessor
  - 8088: cheaper than 8086
  - 80186: an advanced version of 8086
- 1982: 80286 processor
  - 24-bit address
  - Protected mode

Intel processors (3)

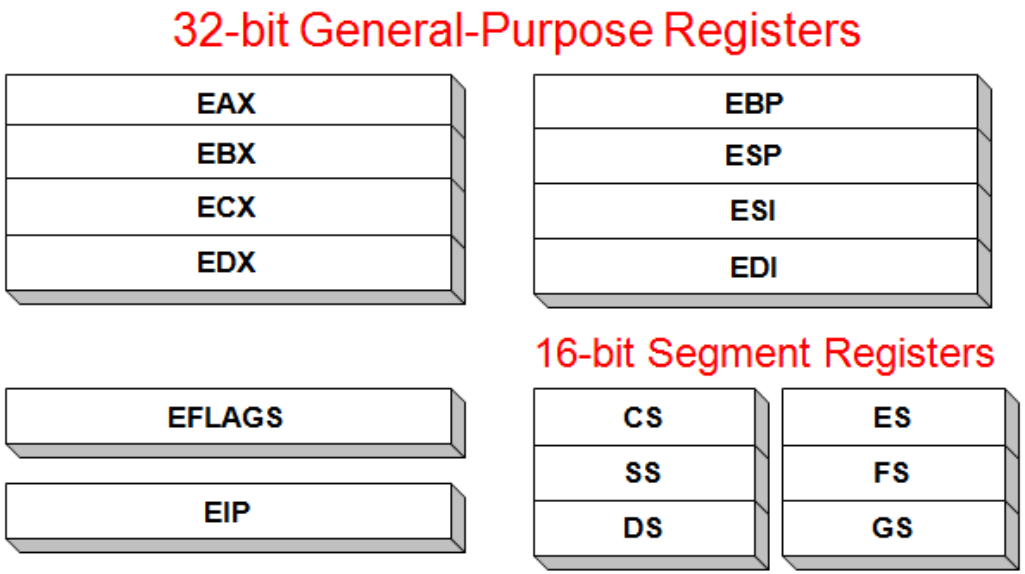
- 1985: Bộ xử lý 80386
  - First 32-bit processor
  - 32-bit data and 32-bit address
  - Paging mode, virtual address
  - 80486 (1989): An advanced version of 80386 with pipeline technique and integration of coprocessor and cache memory.
- 1993: 80586 (Pentium)
  - 64-bit data, 32-bit address
  - Super scalar - two instructions per clock cycle
  - Separation of code and data caches
  - MMX Technology (The later versions)
- 1995: Pentium Pro
- 1997: Pentium II - MMX
- 1999: Pentium III - SSE
- 2000: Pentium IV - SSE2
- 2003: Pentium M

IA32 architecture

- Start from 80386
- Register
- Instruction cycle
- Memory management

Register (1)

- 8 32-bit general-purpose registers
- 6 16-bit segment registers
- Flag register and instruction pointer



Register (2)

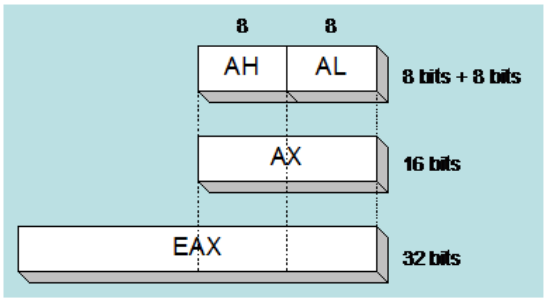
- EAX: Acummulater register
  - Automatically used by multiplication and division.
- ECX: Counter register
  - Automatically used by LOOP
- ESP: Stack Pointer register
  - PUSH, POP changes values of these registers.
- ESI,EDI: Source index and destination index
  - Used in operations with string
- EBP: Base address
  - Refer to local variable in function

Register (3)

- EAX, EBX, ECX, EDX are extended 32-bit registers.
  - AX is a register of 16 low bits of EAX
  - AH, AL is respectively 8 high bits and low bits of AX
- SI, DI, SP, BP is 16 low bits of ESI, EDI, ESP, EBP respectively

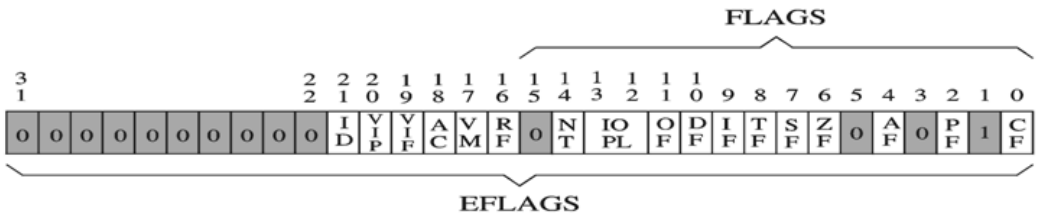
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP



Register (4)

- EFLAGS : this register contains a status flag.
- 8 80-bit registers of float ST(0)-ST(7)
- 8 64-bit registers for MMX
- 8 128-bit XMM registers (Used for SSE)



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

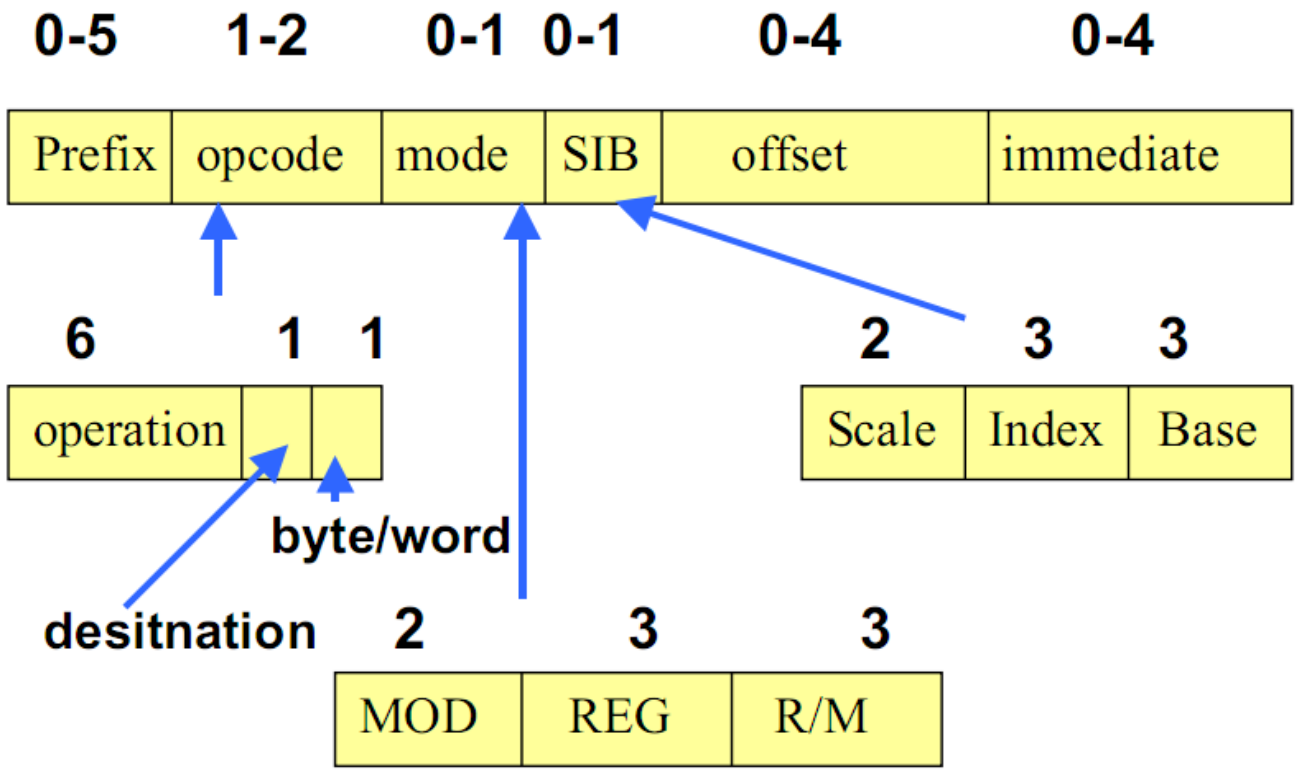
Control flags

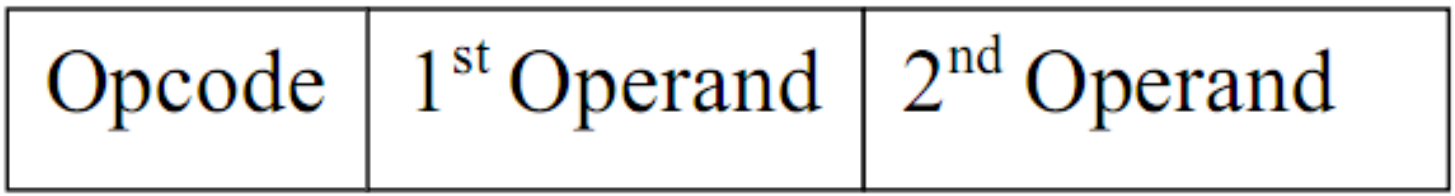
DF = Direction flag

System flags

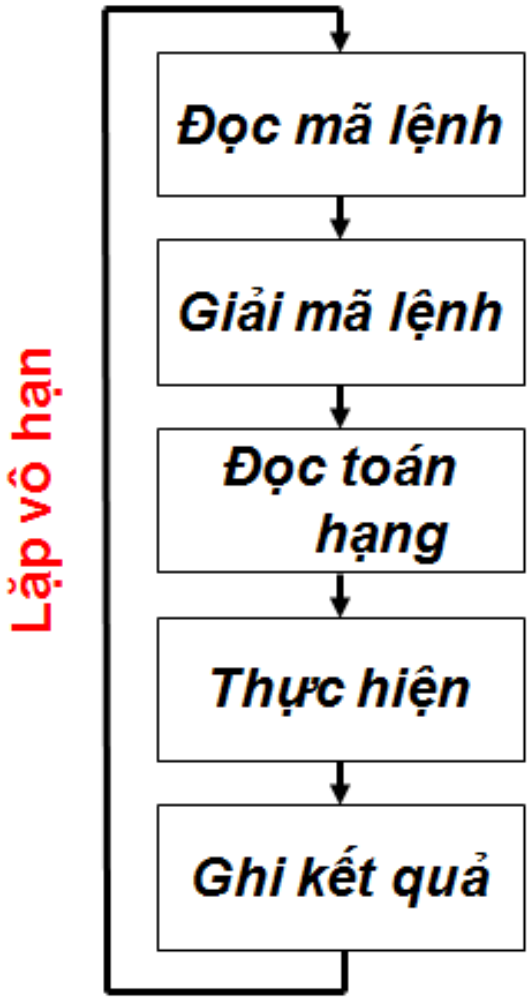
TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

Instruction





Instruction cycle



Lấy mã lệnh từ vùng chương trình

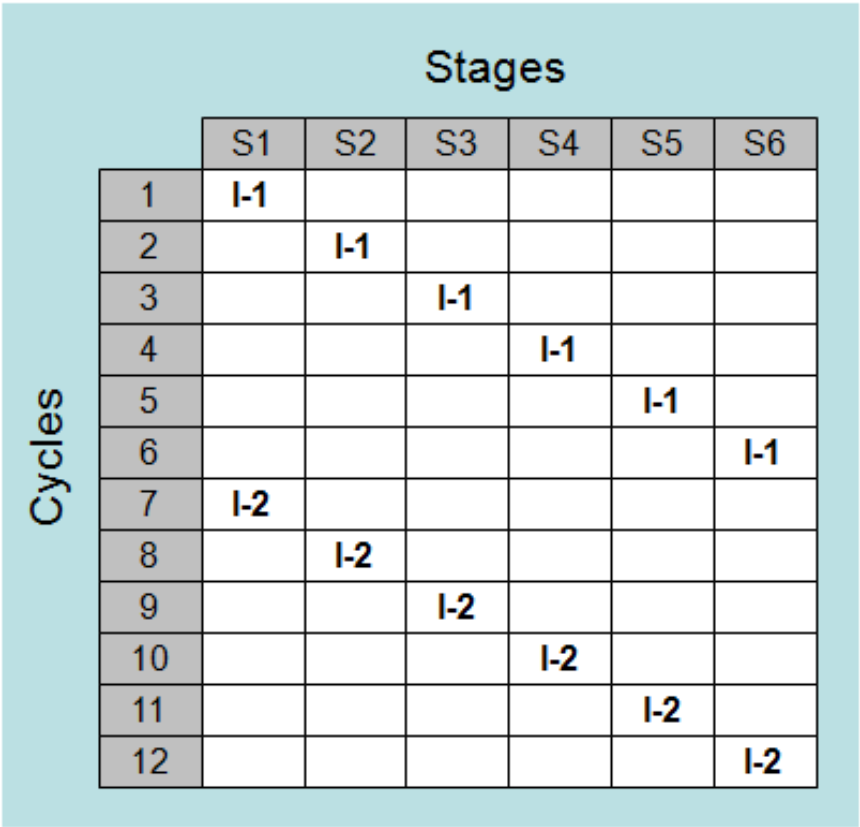
Xác định hành động và kích thước lệnh

Định vị và lấy toán hạng

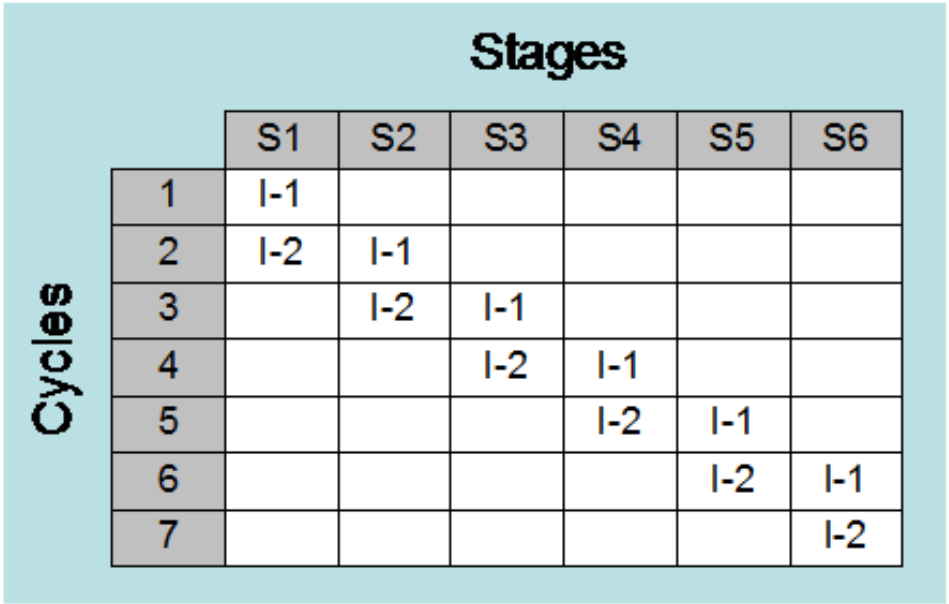
Tính toán giá trị và trạng thái

Lưu giá trị trả về vào bộ nhớ

Instruction pipeline



KHÔNG CÓ ĐƯỜNG ỐNG



CÓ ĐƯỜNG ỐNG

Some basic instructions (1)

```
MOV <source> <destination>
```

- <source> can be: register, immediate value, memory
- <destination> can be: register, memory
- Intel processor does not support direct memory-memory transfer
- Assembly language defines different representation for parameters.

Some basic instructions (2)

```
ADD <destination> <source>
SUB <destination> <source>
DIV <destination> <source>
MUL <destination> <source>
IDIV <destination> <source>
IMUL <destination> <source>
```

- DIV/MUL : unsigned multiplication/division
  - EDX stores high bytes of result
  - EAX stores low bytes of result
- IMUL has three parameters in some case.

Some basic instructions (3)

```
AND <destination> <source>
OR <destination> <source>
NOT <parameter>
NEG <parameter>
```

- NOT: the complement to 1 of the destiny operator
- NEG: the complement to 2 of the destiny operator

Some basic instructions (4)

```
CMP <first parameter> <second parameter>
TEST <first parameter> <second parameter>
```

- Compare and store results to EFLAGS
- TEST performs AND and then set a flag (PL/ZR,PE)

Some basic instructions (5)

```
JMP <label>
 JE <label>
 JL <label>
 JG <label>
JNE <label>
JGE <label>
JLE <label>
```

- Jump if flag is set
- Usually follow CMD/TEST

Some basic instructions (6)

```
LOOP <label>
```

- Decease ECX and jump to `label` if ECX is not equal 0

Some basic instructions (7)

```
LEA <destination> <source>
```

- Load address of `source` into `destination`

Some basic instructions (7)

```
LEA <destination> <source>
```

- Load address of `source` into `destination`

Some basic instructions (8)

```
PUSH <value>
POP <parameter>
PUSHAD
POPAD
```

- PUSH decreases value of ESP, POP increases value of ESP
- PUSHAD and POPAD are used to push/get value of general purpose registers onto the stack

Some basic instructions (9)

```
CALL <parameter>
```

- CALL transfers control to the address stored by parameter (register, memory, global offset value)
- Automatically push returned address (the instruction address of next call) onto the stack and decrease ESP
- Parameter passed to function is stored in the stack before CALL call.

Some basic instructions (10)

```
RET <optional parameter>
```

- Transfer the control to the instruction being the next of CALL through getting the return value on the stack
- Optional parameter is used to automatically retrieve parameters passed to function on the stack

Some basic instructions (11)

```
ENTER <size of frame> <mức độ lồng>
```

- ENTER prepares stack frame for a function/procedure call.
- Store EBP onto the stack, copy ESP to EBP, and increase ESP to the size of frame.

Some basic instructions (11)

```
LEAVE
```

- Release the current stack frame
- Assign value of EBP to ESP, then get value of EBP from the stack

Memory management modes (1)

- Real-address mode
  - Use 1MB of memory
  - A program can access anywhere on the memory
  - MS-DOS
- Protected mode
  - Start from 80386
  - Each program can use at most 4GB
  - OS decides memory area for each program
  - Programs can not access to mutual memory areas
  - Windows, Linux
- Virtual 8086 mode
  - Processor is in protected mode, it also makes a 8086 virtual machine with 1MB of memory

Real address mode

- A program can access 6 memory segments.
  - Code segment (CS)
  - Stack segment (SS)
  - Data segment (DS)
  - Three extent segments (ES, FS, GS)
- Each segment has at most 64KB of memory
- Logical address: [segment:offset]
- Physical address: (segment $<<$ 4 + offset)

Flat memory mode

- Modern OSs do not use segmented memory mechanism as in real address model.
- Segment register is initialized by the OS.
- Program uses 32-bit address.
- All of memory segments have a common base address.
- OS plays a role of mapping logical address (segment:offset) to physical address
  - Logical address $\rightarrow$ Linear address $\rightarrow$ Physical address
  - Linear address (virtual address, 32bit) is calculated on segment tables and a pair of segment:offset
  - Read address (32 bit) is calculated on linear address, page directory and page table

GNU Assembler

- GNU assembler (gas) is an assembler towards multiple different architectures such as: Intel IA32, ARM, ...
- Is a component in the package `binutils` and other system component such as `ld`, `ar`
- Input is an assembly program (`prog.s`) and output is an object program (`prog.o`)
- Usage

```
# as -o prog.o prog.s
```

Assembly program

- Assembly program is a form of intermediate code
- Represent symbols for object code.
- Contains the following information:
  - Instruction for executing program
  - Data definition
  - Memory segmentation
  - Linking and Debug information
- Is a sequence of instruction and pseudo-instruction
  - Instruction: corresponds to instructions of micro-processor, depends on architecture
  - Pseudo-instruction: supports generating object code, depends on the assembler

Ví dụ - prog2.c

```
00: #include <stdio.h>
01:
02: int a;
03: static int b = 33;
04:
05: main()
06: {
07:     static int c;
08:     int d;
09:
10:     d = a + b + c;
11:
12:     printf("%d\n", d);
13: }
```

Ví dụ - prog2.s

```
        .file   "prog2.c"           // Debug information
        .data                       // Mark data area
        .p2align 2                  // Align memory
        .type   b, @object          // Define a type for label b
        .size   b, 4                // Define size for label b
b:                                  // Label b (variable b)
        .long   33                  // Allocate memory for lable b
        .local  c.0                 // Define local range for label c
        .comm   c.0,4,4             // Allocate memory for label c
        .section        .rodata     // Mark read-only data area
.LC0:                               // Label for constant string
        .string "%d\n"              // Allocate memory and set its value
        .text                       // Mark area of instruction code
        .p2align 2,,3               // Align memory
.globl main                         // Define a range for label main
        .type   main, @function     // Define size for label main
main:                               // Label main
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    b, %eax
        addl    a, %eax
        addl    c.0, %eax
        movl    %eax, -4(%ebp)
        pushl   -4(%ebp)
        pushl   $.LC0
        call    printf
        addl    $8, %esp
        leave
        ret
        .size   main, .-main        // Define size for lable main
        .comm   a,4,4               // Allocate memory for label a
```

Addressing mode

- Register addressing(prefix %)
  - 32 bit: `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp`, `%esp`
  - 16 bit: `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, `%sp`
    - 16 low bits of corresponding register
  - 8 bit: `%ah`, `%al`, `%bh`, `%bl`, `%ch`, `%cl`, `%dh`, `%dl`
    - 8 low bits and 8 high bits of corresponding 16-bit register
- Direct value addressing(prefix $)
  - `$1`
- Absolute/direct addressing
  - `1`
- Relative addressing: `-4(%ebp)`

Size of processing data

Ví dụ

```
// C source code
*a = 1;
// Convert to assembly code
mov $1, 0x80800000
```

- Operation with 8bit, 16bit, or 32bit?
- Use instruction with the following postfix
  - 'l' for 32 bit
    `movl $1, 0x80800000`
  - 'w' for 16 bit
    `movw $1, 0x80800000`
  - 'b' for 8 bit
    `movb $1, 0x80800000`

Pseudo-instruction

```
<label> <opcode> <list of parameter>
```

- Pseudo-instruction is directives which is used by assembler to execute a certain work in the process of assembling.
  - Define data (constant, variable, etc.)
  - Memory segmentation
  - Define a range and linking directive.

Label and its attribute

- Label is used as reminder symbol instead of instruction address or data address.
- Type of label
  - `.type <label>, <label type (function/object)>`
- Size of label
  - `.size <label>, <label size>`
- range of label
  - Global
    `.globl <label>`
  - Local
    `.local <label>`

Label and its attribute

Range of label

```
int a;
static int b;

main() {
   static int c;
   int d
   ...
}

func(){
}
```

- Variables `a, b, c, d` can be "seen" from other files?
- Variables `a, b, c, d` can be "seen" from function func()?

Data definition

- Define an integer of 4 byte

```
x: .long 12
```

- Define an integer of 2 byte

```
x: .word 34
```

- Define an integer of 1 byte

```
x: .byte 5
c: .byte 'A'
```

- Define an array

```
a: .space 20
```

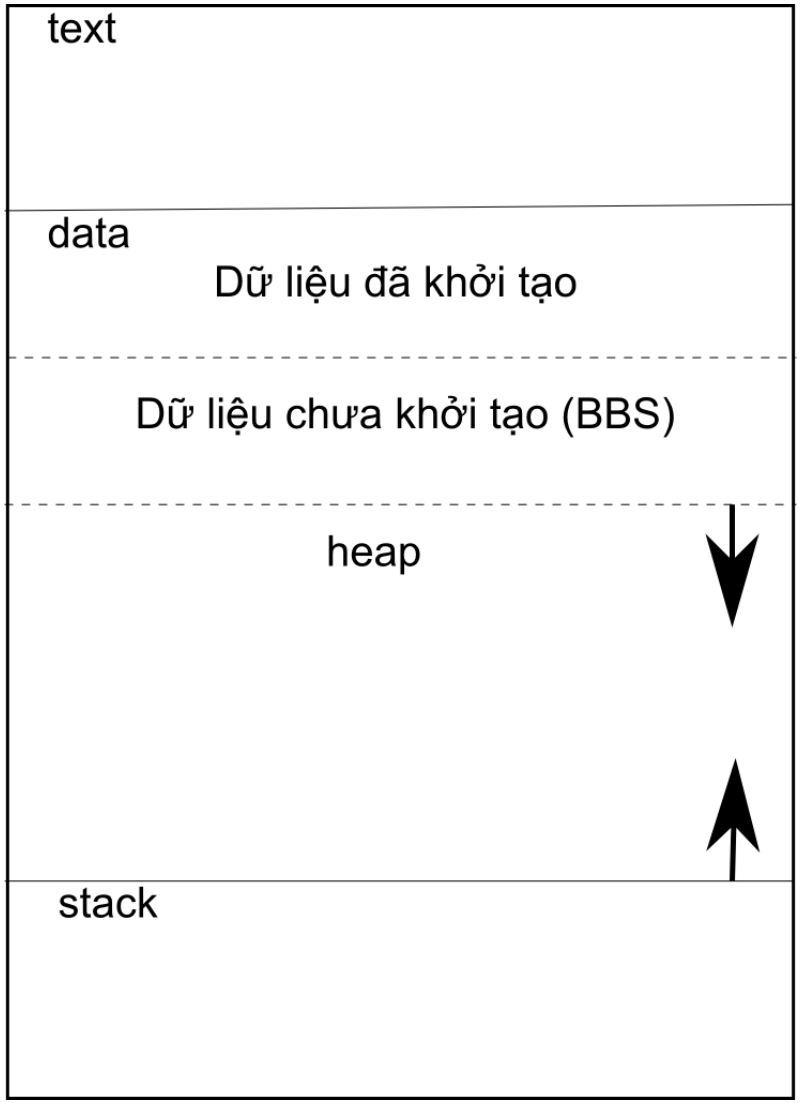
- Define a string

```
m1: .ascii "Hello World!\0"
m2: .asciiz "Hello World!"
```

Memory segmentation

- Area of program code
  - `.text`
  - readable and executable
- Data area
  - Data is initialized
    - Normal data `.data`
    - Read-only data `.section .rodata`
  - Data is not initialized (BBS)
- Stack
  - Data is allocated in execution time.
    - Parameter for function
    - Return address
    - Local variable
  - readable, writable, not executable

Memory segmentation



Memory Alignment

- With some processors, data is not allocated to suitable address will significantly decrease the execution speed of instruction
  - 16-bit data must be allocated at even address
  - 32-bit data must be allocated at the address being divisible by 4
- GNU assembler supports some pseudo-instructions for doing memory alignment.
  - `.p2align x, y`
  - Align to the address being divisible by 2^x
  - Shift maximum of y byte (if being over y byte, then ignore the alignment)

Subroutine (1)

- Subroutine is a portion of code within a larger program that performs a specific task.
  - function, procedure
  - Relatively independent of the remaining code
  - can be re-usable
  - Can accept parameter and return result
- Terms
  - Caller
  - Callee

Subroutine (2)

Ví dụ

```
// C
int add(int a, int b)
{
    int c;

    c = a + b;

    return c;
}
```

```
// Assembly language
add:
  pushl %ebp
  movl %esp, %ebp
  subl $4, %esp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  movl %eax, -4(%ebp)
  movl -4(%ebp), %eax
  leave
  ret
```

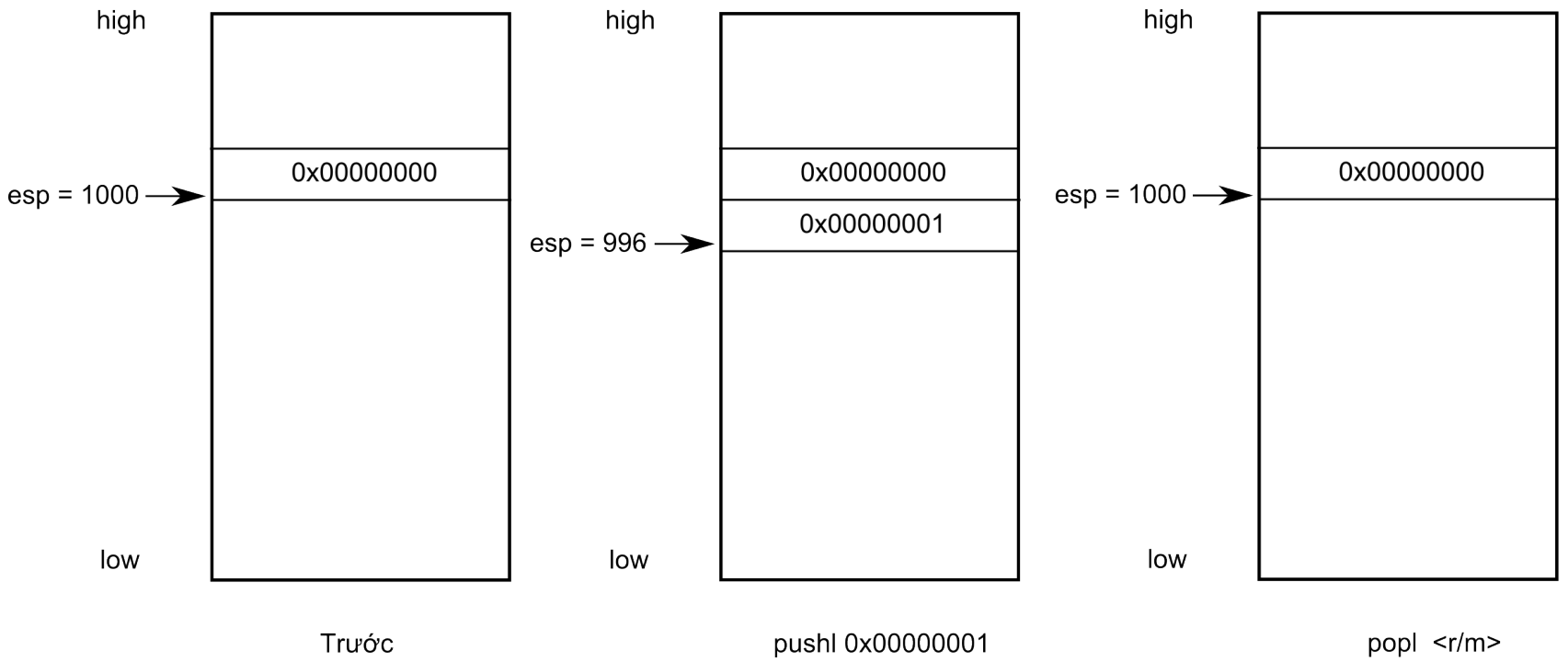
Active and finalize subroutine

- Active a subroutine
  - Pass parameter
  - Store return address and pass the control to the callee
  - Store local variable of the caller
  - Execute the callee
- Finalize subroutine
  - Pass return result
  - Restore local variable of the caller
  - Go back the caller by using return address stored

Stack and stack frame (1)

- Stack is a memory area that stores temporary data provided for operation of software.
  - Parameter
  - Return address
  - Local variable (auto)
- Be allocated on the high address area of address space of program.
- Be organized as LIFO buffers, register %esp contains a pinter to the top of stack (decrease when pushing a value in, and increase when taking a value out)
  - `push <arg16/arg32>`
    - `%esp = %esp - [2|4]; (%esp) = <arg>;`
  - `pop <arg>`
    - `<arg> = (%esp); %esp = %esp + [2|4];`

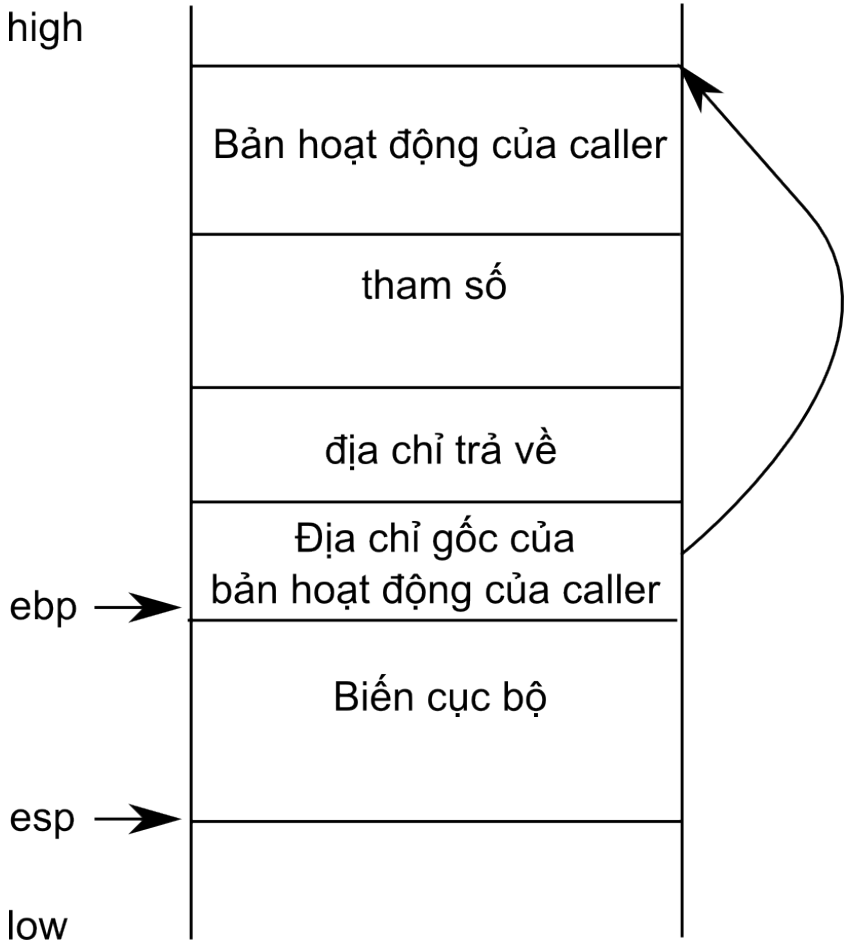
Stack and stack frame (2)



Stack and stack frame (3)

- Stack frame/ activation record is a portion of stack that is allocated whenever a subroutine actives.
  - Store values of local variable in the subroutine
  - Is allocated when a subroutine is executed and deallocated when a subroutine returns to its caller
  - Register %ebp holds a pointer to original address of callee's activation record.
  - Original address of activation record of callee are stored on stack and restored before callee returns to its caller

Stack and stack frame (4)



Stack and stack frame (5)

Allocate stack frame

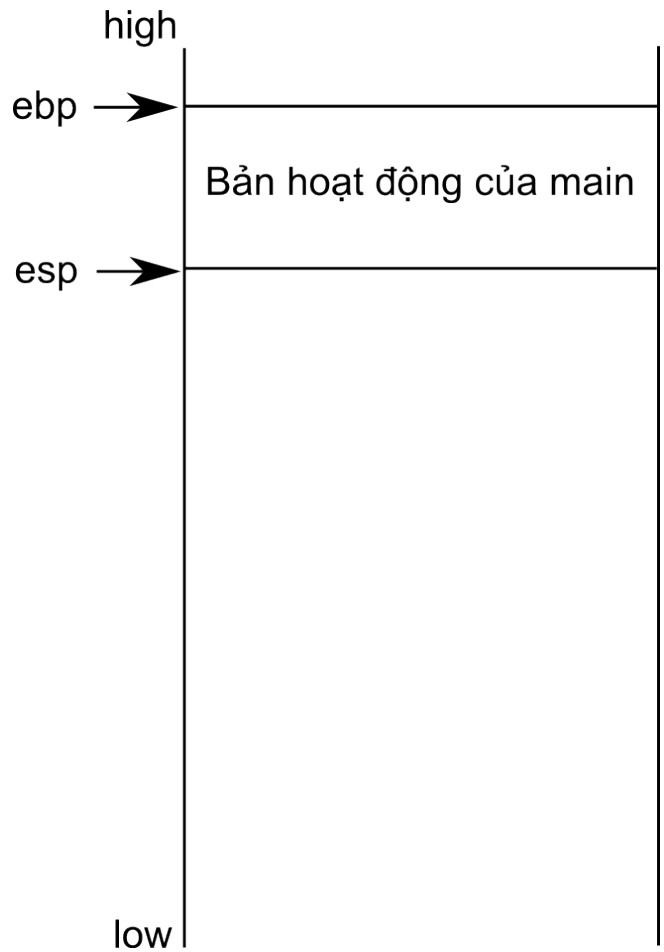
```
pushl %ebp                          enter <size>, 0
movl  %esp, %ebp
subl  <size>, %esp
```

Deallocate stack frame

```
movl  %ebp, %esp                    leave
popl  %ebp
```

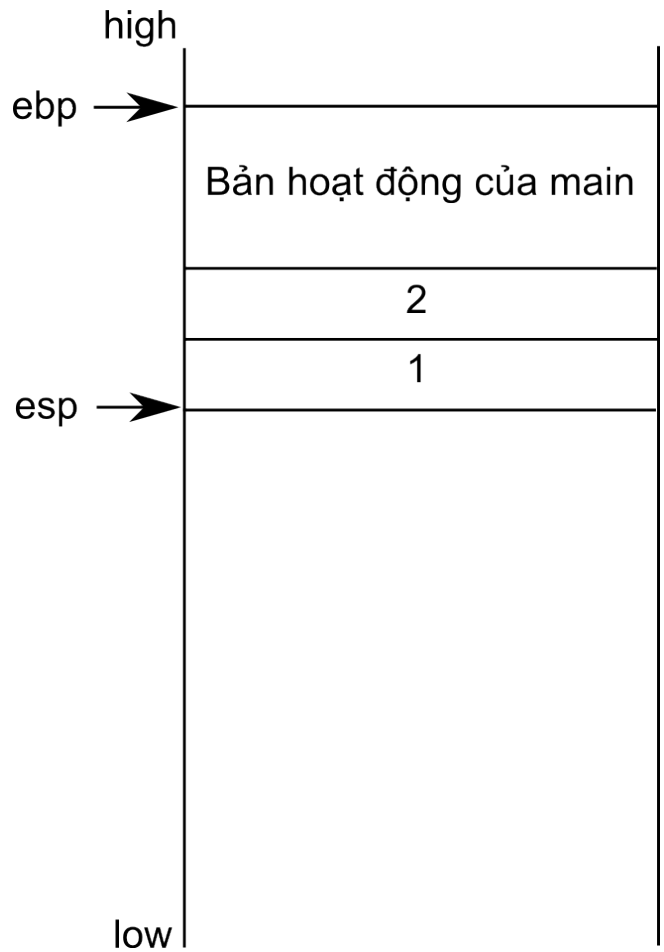
Example (prog3.s)

```
01  add:
02      pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05      movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09      leave
10      ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14->    subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18      movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



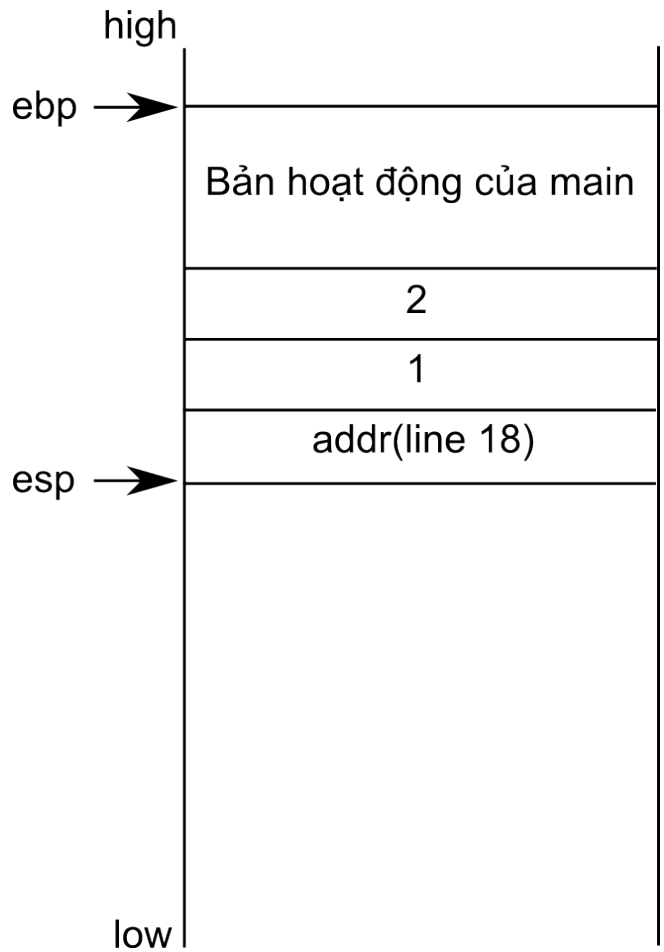
Example (prog3.s)

```
01   add:
02       pushl   %ebp
03       movl    %esp, %ebp
04       subl    $4, %esp
05       movl    12(%ebp), %eax
06       addl    8(%ebp), %eax
07       movl    %eax, -4(%ebp)
08       movl    -4(%ebp), %eax
09       leave
10       ret
11   main:
12       pushl    %ebp
13       movl    %esp, %ebp
14       subl    $8, %esp
15       movl    $2, 4(%esp)
16       movl    $1, (%esp)
17->     call    add
18       movl    %eax, 4(%esp)
19       movl    $.LC0, (%esp)
20       call    printf
21       leave
22       ret
```



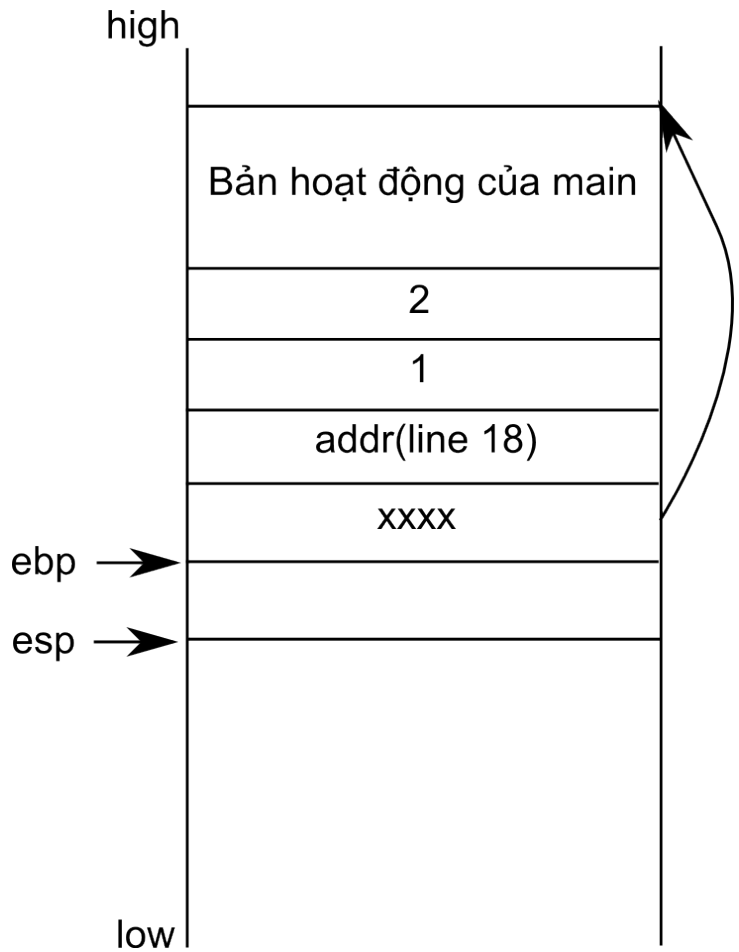
Example (prog3.s)

```
01  add:
02->    pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05      movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09      leave
10      ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14      subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18      movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



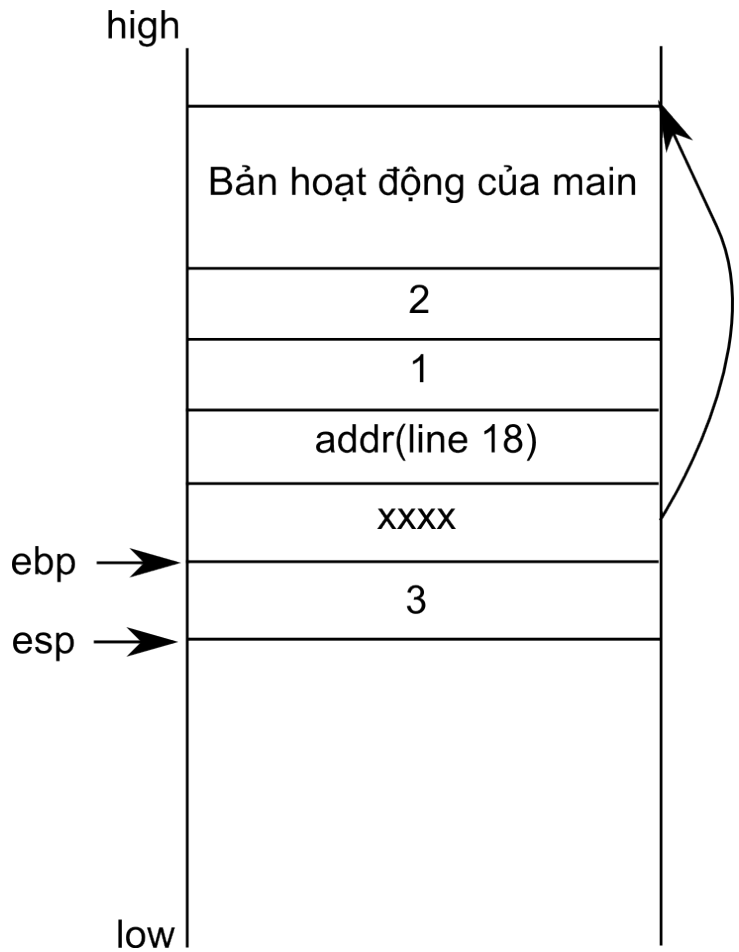
Example (prog3.s)

```
01  add:
02      pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05->    movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09      leave
10      ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14      subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18      movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



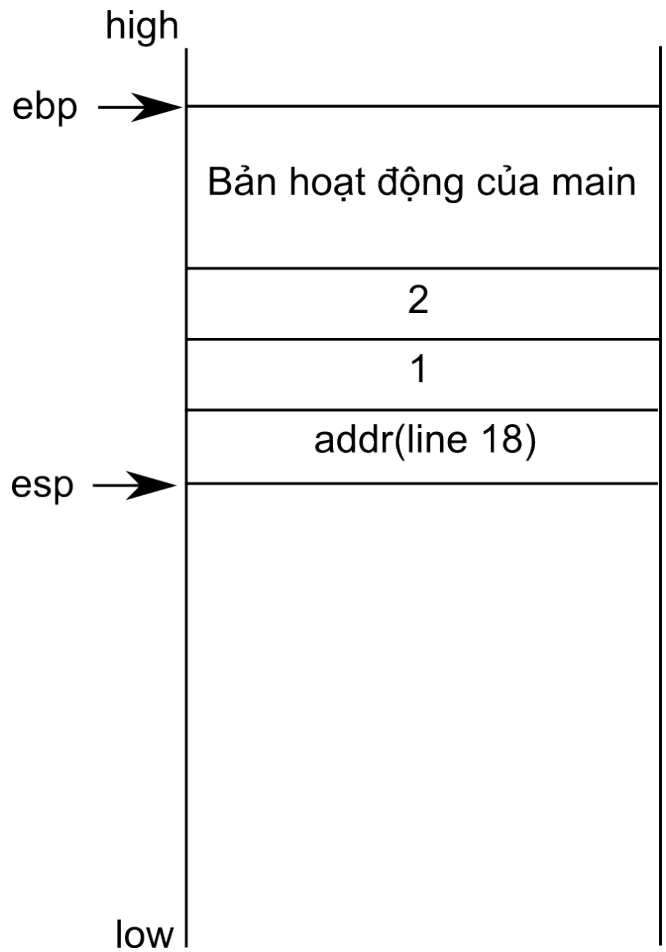
Example (prog3.s)

```
01  add:
02      pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05      movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09->    leave
10      ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14      subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18      movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



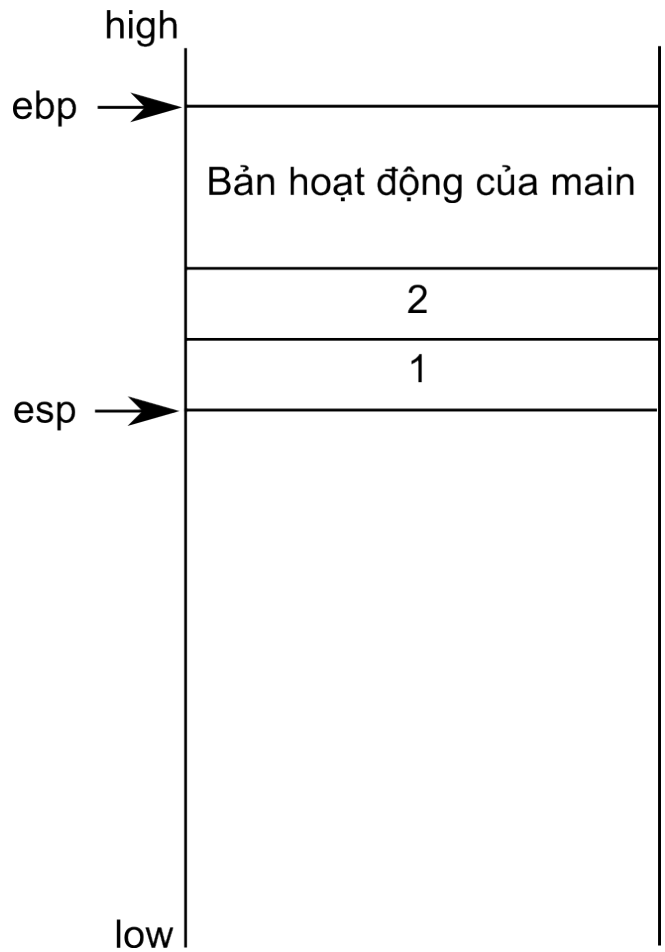
Example (prog3.s)

```
01  add:
02      pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05      movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09      leave
10->    ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14      subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18      movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



Example (prog3.s)

```
01  add:
02      pushl   %ebp
03      movl    %esp, %ebp
04      subl    $4, %esp
05      movl    12(%ebp), %eax
06      addl    8(%ebp), %eax
07      movl    %eax, -4(%ebp)
08      movl    -4(%ebp), %eax
09      leave
10      ret
11  main:
12      pushl    %ebp
13      movl    %esp, %ebp
14      subl    $8, %esp
15      movl    $2, 4(%esp)
16      movl    $1, (%esp)
17      call    add
18->    movl    %eax, 4(%esp)
19      movl    $.LC0, (%esp)
20      call    printf
21      leave
22      ret
```



Calling convention

- To standardize the interface between caller and callee.
  - Parameter passing convention
  - Convention for getting return value
- Function calling convention
  - `cdecl`
  - `pascal`
  - `stdcall`
  - `fastcall`

Function calling convention cdecl

- Apply for C/C++ on Intel x86 processor
  - Parameters are passed via the stack in the right-to-left order
  - Register %eax is used to store return value
  - Caller plays a role of passing parameter and releasing parameter on the stack
  - Allow passing a undefined amount of parameters (varargs)

Example

```
int func (int, int, int);
int a, b, c, d;

d = func (a, b, c);
```

```
push c
push b
push a
call func
add $12,%esp
mov %eax, d
```

Function calling convention pascal

- Apply for applications on Windows 3.x và OS/2
  - Parameters are passed via the stack in the left-to-right order
  - Register %eax is used to store return value
  - Caller plays a role of passing parameter and callee releases a portion of stack that holds parameter (instruction `ret <imm16>`
  - Only accept parameters with an defined amount and size.

Example

```
int func (int, int, int);
int a, b, c, d;

d = func (a, b, c);
```

```
push a
push b
push c
call func
mov %eax, d
```

Function calling convention `stdcall`

- Apply for Win32 API
  - Parameters are passed via the stack in the right-to-left order
  - Register %eax is used to store return value
  - Caller plays a role of passing parameter and callee releases a portion of stack that holds parameter (instruction `ret <imm16>`
  - Only accept parameters with a defined amount and size.

Example

```
int func (int, int, int);
int a, b, c, d;

d = func (a, b, c);
```

```
push c
push b
push a
call func
mov %eax, d
```

Function calling convention `fastcall`

- Apply for Win32 API
  - Parameters are partly passed via register, and partly via stack
  - Register %eax is used to store return value
  - Depend on compiler
  - Caller plays a role of passing parameter and callee releases a portion of stack that holds parameter

Example

```
int func (int, int, int);          mov a, %eax
int a, b, c, d;                    mov b, %edx
                                   mov c, %ecx
d = func (a, b, c);                call func
                                   mov %eax, d
```