Sistemi Operativi

Lez. 16

Elementi del linguaggio Assembler AT&T

Data Sizes

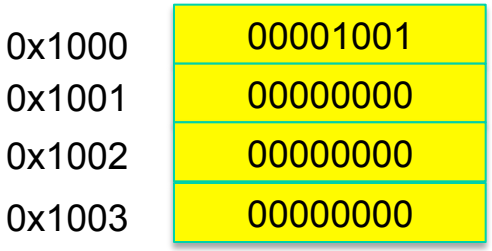
- Three main data sizes
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes
- Separate assembly-language instructions
  - E.g., addb, addw, and addl

Declaring variables

- **.byte**
  - Bytes take up one storage location for each number. They are limited to
  - numbers between 0 and 255.
- **.word**
  - Ints (which differ from the **int** instruction) take up two storage locations for each number. These are limitted to numbers between 0 and 65535.9
- **.long**
  - Longs take up four storage locations. This is the same amount of space the registers use, which is why they are used in this program. They can hold numbers between 0 and 4294967295.
- **.ascii**
  - The **.ascii** directive is to enter in characters into memory. Characters each take up one storage location (they are converted into bytes internally). So, if you gave the directive **.ascii "Hello there\0"**, the assembler would reserve 12 storage locations (bytes).

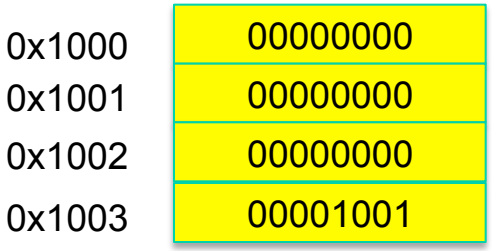
Little endian

- Intel is a little endian architecture
- Least significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”
- Es.: l’intero 9 di un dato di 4 byte è così memorizzato



Big endian

- Some other systems use big endian
- Most significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”
- Es.: l’intero 9 all’indirizzo 1000



Esempio

```
int main(void) {
int i=0x003377ff, j;
unsigned char *p = (unsigned char *) &i;
for (j=0; j<4; j++)
   printf("Byte %d: %x\n", j, p[j]);
}
```

OUTPUT Little endian: ?

OUTPUT Big endian: ?

IA32 Instruction Format

- General format:
  - `[prefix] opcode operands`
- Prefix used only in String Functions
- Operands represent the direction of operands
  - Single operand instruction: **`opcode src`**
  - Two operand instruction : **`opcode src,dest`**

Loading and Storing Data

- Data can be stored in:
  - Registers
  - Variables
- Variables are stored in memory
- Registers are “special” memory locations directly accessible by the processor
- The processor can only manipulate data inside registers
- The instruction to load from and store to memory, is `mov src,dest`

General purpose registers

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

Accessing data

- Processors have many ways to access data known as “addressing modes”

- Register addressing: simply moves data in or out of a register
  - Example: `movl %edx, %ecx`
  - Copy value in register EDX into register ECX
  - Choice of register(s) embedded in the instruction

Immediate Addressing

- Immediate mode is used to load direct values into registers. For example, if you wanted to load the number 12 into **%eax**, you would simply do the following:

```
movl $12, %eax
```

- Notice that to indicate immediate mode, we used a dollar sign in front of the number. If we did not, it would be direct addressing mode, in which case the value located at memory location 12 would be loaded into **%eax** rather than the number 12 itself

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address
- `movl 2000, %ecx`
  - Four-byte variable located at address 2000
  - Read the four bytes value contained at location 2000
  - Load the value into the ECX register
- Can use a label for (human) readability
- E.g. `movl  i,%eax`

Indirect Addressing

- Load or store from a previously-computed address
  - Register with the address is an operand in the instruction
  - Instruction reads from or writes to that address
- Example: `movl (%eax), %ecx`
  - EAX register stores a 32-bit address (e.g., 2000)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Dynamically allocated data referenced by a pointer
- The “(%eax)” essentially dereferences a pointer

Base pointer addressing

- Load or store with an offset from a base address
  - Register storing the base address
  - Fixed offset also embedded in the instruction
  - Instruction computes the address and does access
- Example: `movl 8(%eax), %ecx`
  - EAX register stores a 32-bit base address (e.g., 2000)
  - Offset of 8 is added to compute address (e.g., 2008)
  - Read long-word variable stored at that address
  - Load the value into the ECX register

Indexed Addressing Example

```
int a[20];
…
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

eax = ??
ebx = ??
ecx = ??

```
    movl $0, %eax
    movl $0, %ebx
sumloop:
    movl a(,%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```

Summary

- Immediate addressing: data stored in the instruction itself
  - `movl $10, %ecx`
- Register addressing: data stored in a register
  - `movl %eax, %ecx`
- Direct addressing: address stored in instruction
  - `movl foo, %ecx`
- Indirect addressing: address stored in a register
  - `movl (%eax), %ecx`
- Base pointer addressing: includes an offset as well
  - `movl 4(%eax), %ecx`
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - `movl 2000(,%eax,1), %ecx`

Arithmetic Instructions

- Simple instructions
  - `add{b,w,l} source, dest` dest = source + dest
  - `sub{b,w,l} source, dest` dest = dest – source
  - `inc{b,w,l} dest` dest = dest + 1
  - `dec{b,w,l} dest` dest = dest – 1
  - `cmp{b,w,l} source1, source2` source2 – source1

Mul/Div

- Multiply
  - `mul (unsigned) or imul (signed)`
    - Performs signed multiplication and stores the result in the second operand. If the second operand is left out, it is assumed to be **%eax**, and the full result is stored in the double-word **%edx**:**%eax**
- Divide
  - `div (unsigned) or idiv (signed)`
    - Divides the contents of the double-word contained in the combined **%edx**:**%eax** registers by the value in the register or memory location specified. The **%eax** register contains the resulting quotient, and the **%edx** register contains the resulting remainder

Bitwise logic instructions

- Simple instructions
  - `and{b,w,l} source, dest` dest = source & dest
  - `or{b,w,l} source, dest` dest = source | dest
  - `xor{b,w,l} source, dest` dest = source ^ dest
  - `not{b,w,l} dest` dest = ~dest
  - `sal{b,w,l} source, dest` dest = dest << source
  - `sar{b,w,l} source, dest` dest = dest >> source

Control Flow

- We obtain contro flow using two instructions:

```
cmpl $0, %eax
je   end_loop
```

- The first one is the `cmpl` instruction which compares two values, and stores the result of the comparison in the status register EFLAGS. Notice that the comparison is to see if the second value is greater than the first
- The second one is the flow control instruction `JUMP` which says to jump to the end_loop depending on the values stored in the status register and on the condition expressed

Types of Jumps

- je: Jump if the values were equal
- jg: Jump if the second value was greater than the first value
- jge: Jump if the second value was greater than or equal to the first value
- jl: Jump if the second value was less than the first value
- jle: Jump if the second value was less than or equal to the first value
- jmp:Jump no matter what. This does not need to be preceeded by a comparison

Stack

- Many CPU's have built-in support for a stack
- A stack is a Last-In First-Out (LIFO) list
- The stack is an area of memory that is organized in this fashion. The PUSH instruction adds data to the stack and the POP instruction removes data
- The data removed is always the last data added
- The ESP register contains the address of the data that would be removed from the stack. This data is said to be at the top of the stack
- The processor references the SS register automatically for all stack operations. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.
- Data can only be added in double word units. That is, one can not push a single byte on the stack

Stack

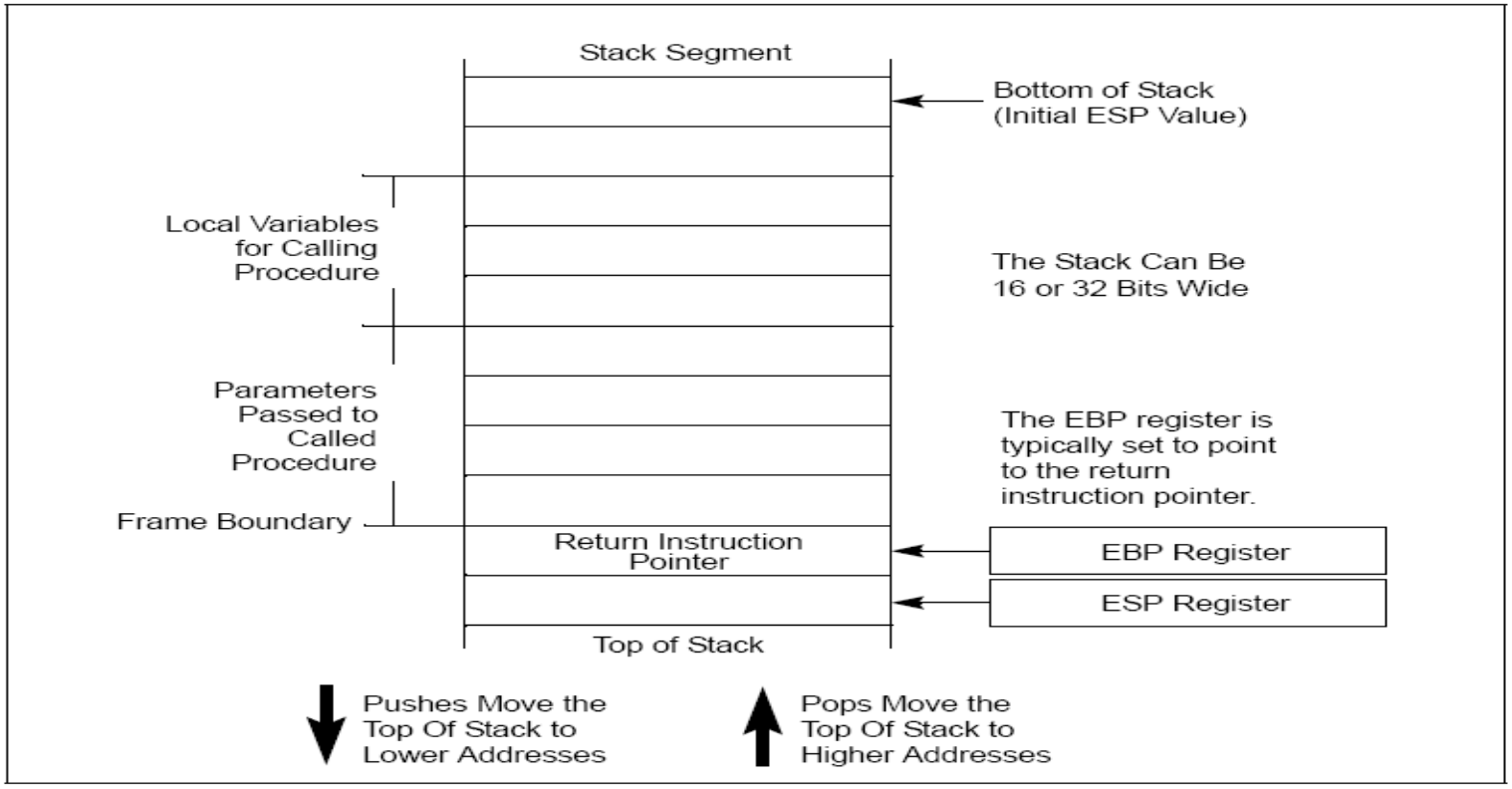


Figure 6-1. Stack Structure

PUSH

- The PUSH instruction inserts a double word on the stack by subtracting 4 from ESP and then stores the double word at [ESP]

```
pushl src        →        subl $4,%esp
                          movl src,(%esp)
```

- The 80x86 also provides a PUSHA instruction that pushes the values of EAX, EBX, ECX, EDX, ESI, EDI and EBP registers (not in this order)

POP

- The POP instruction reads the double word at [ESP] and then adds 4 to ESP

```
popl dest        →   movl (%esp),dest
                     addl $4,%esp
```

- The `popa` instruction, recovers the original values of the registers saved by the `pusha`

CALL/RET

- The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy. The CALL instruction makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack.
- The RET instruction pops off an address and jumps to that address.
- When using these instructions, it is very important that one manage the stack correctly so that the right number is popped off by the RET instruction

Implementation of Call

- `call  subprogram1`
- becomes:

```
pushl   %eip
jmp     subprogram1
```

ESP →

Implementation of ret

- `ret`
- `becomes:`
  - `pop  %eip`

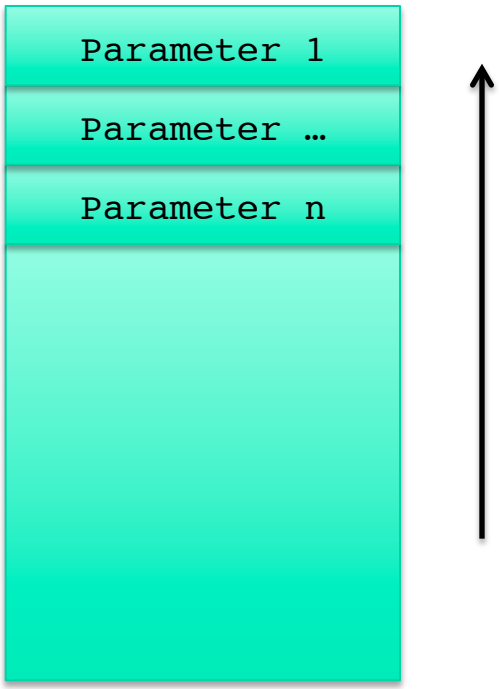
- `ESP` →

Passing Parameters

- How does caller function pass parameters to callee function?
- Attempted solution: Pass parameters in registers
  - Problem: Cannot handle nested function calls
  - Also: How to pass parameters that are longer than 4 bytes?
- Caller pushes parameters before executing the call instruction
- Parameters are pushed in the reverse order
  - Push the n-th parameter first
  - Push 1° parameter last

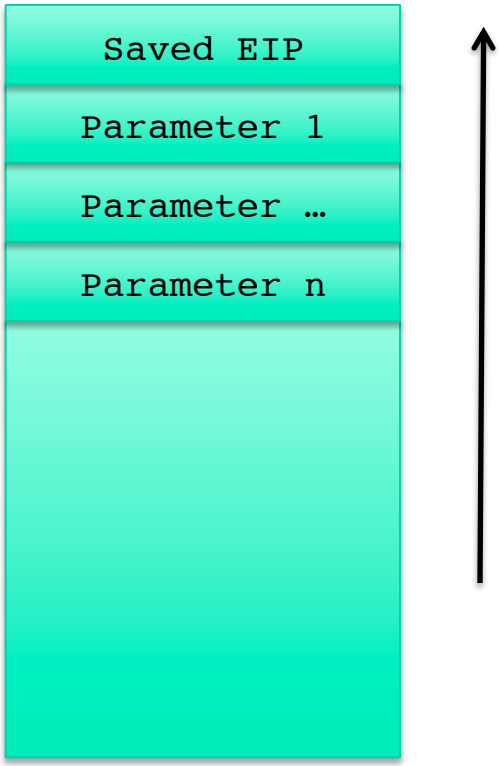
Parameters

ESP before → call



Parameter

ESP after → call



Callee addresses params relative to ESP:
Param 1 as 4(%esp)

Parameter

- After returning to the caller, the caller pops the parameters from the stack

```
…
# Push parameters
pushl $5
pushl $4
pushl $3
call sub
# Pop parameters
addl $12, %esp
```

```
sub:
…
movl 4(%esp),var1
movl 8(%esp),var2
movl 12(%esp), var3
…
ret
```

%ebp

- As callee executes, ESP may change
  - E.g., preparing to call another function
- It can be very error prone to use ESP when referencing parameters. To solve this problem, the 80386 supplies another register to use: EBP. This register's only purpose is to reference data on the stack
- Use EBP as fixed reference point to access params

Using EBP (prolog)

- A subprogram before overwriting ebp first save the old value of EBP on the stack and then set EBP to be equal to ESP. This allows ESP to change as data is pushed or popped off the stack without modifying EBP

```
pushl %ebp
movl  %esp, %ebp
(sub  Local_bytes, %esp)
```

- Regardless of ESP, the subprogram can reference param 1 as 8(%ebp), param 2 as 12(%ebp), etc.

Using ebp (epilog)

- Before returning, callee must restore ESP and EBP to their old values executing the epilog

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

Enter/Leave

- The ENTER instruction performs the prologue code and the LEAVE performs the epilogue
- The ENTER instruction takes two immediate operands.
- For the C calling convention, the second operand is always 0. The first operand is the number bytes needed by local variables. The LEAVE instruction has no operands

```
subprogram_label:
        enter   LOCAL_BYTES, 0      ; = # bytes needed by locals
; subprogram code
        leave
        ret
```

Esempio 1

```
.section .text
proc:
        movl    4(%esp), %ecx
        movl    $1, %eax
1:      mull    %ecx
        subl    $1, %ecx
        cmp     $0, %ecx
        jgt     1b
        ret

.globl _start
_start:
        pushl $5
        call  proc
        addl  $4,%esp
        pushl %eax
        call  exit
```

Esempio 2

```
.text                   # section declaration
                        # we must export the entry point to the ELF linker or
.global _start          # loader. They conventionally recognize _start as
                        # entry point. Use ld -e foo to override the default
_start:
# write our string to stdout
        movl    len,%edx            # third argument: message length
        movl    $msg,%ecx           # second argument: pointer to msg
        movl    $1,%ebx             # first argument: file handle (stdout)
        movl    $4,%eax             # system call number (sys_write)
        int     $0x80               # call kernel and exit
        movl    $0,%ebx             # first argument: exit code
        movl    $1,%eax             # system call number (sys_exit)
        int     $0x80               # call kernel
.data                               # section declaration
msg:    .ascii  "Hello, world!\n"   # our dear string
        len = . - msg               # length of our dear string
```

Esercizio

- Write a program which computes in %ecx the sum of the first 1000 natural numbers

Compiling & Linking

- To assembly the program type in the command

  as name.s -o name.o

- as is the command which runs the assembler, name.s is the source file, and –o name.o tells the assemble to put it's output in the file name.o which is an object file. An object file is code that is in the machine's language, but has not been completely put together.
- In most large programs, you will have several source files, and you will convert each one into an object file
- The linker is the program that is responsible for putting the object files together and adding information to it so that the kernel knows how to load and run it.
- To link the file, enter the command

  ld name.o -o name

Executing

- You can run the executable prog by typing in the command

  ./prog

- The ./ is used to tell the computer that the program isn't in one of the normal program directories, but is the current directory instead

Debugging

- In assembly language, even minor errors usually have results such as the whole program crashing with a segmentation fault error
- Therefore, to aid in determining the source of errors, you must use a source debugger
- The debugger we will be looking at is GDB - the GNU Debugger
- It can debug programs in multiple programming languages, including assembly language

Debugging

- To run a program under gdb you need to have the assembler include debugging information in the executable. All you need to do to enable this is to add the --gstabs option to the as command. So, you would assemble it like this:

```
as --gstabs name.s –o name.o
```

- Linking would be the same as normal
- Now, to run the program under the debugger, you would type in

```
gdb name
```

gdb

```
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or
distribute copies of it under certain conditions. Type
"show copying" to see the conditions. There is
absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

- At this point, the program is loaded, but is not running yet. The debugger is waiting your command. To run your program, just type in run.

Some commands

- A breakpoint is a place in the source code that you have marked to indicate to the debugger that it should stop the program when it hits that point
- To set breakpoints you have to set them up before you run the program. Before issuing the run command, you can set up breakpoints using the break command
- For example, to break on line 27, issue the command break 27. Then, when the program crosses line 27, it will stop running, and print out the current line and instruction.

Some commands

- To follow the flow of a program, keep on entering stepi (for "step instruction"), which will cause the computer to execute one instruction at a time
- To check the contents of register in GDB either use the command info register or print/ $eax to print register eax in hexadecimal, or do print/d $eax to print it in decimal
- x/ addr: print the contents of memory address addr
- For other command see the help command