Computer Systems
Lecture 9

Overview

- CPU Registers
- CPU status flags
- Inline assembler
- Stack
- PUSH and POP
- Adjusting stack pointer
- Stack as the temporary store
- Passing parameters

CPU Registers in Pentium

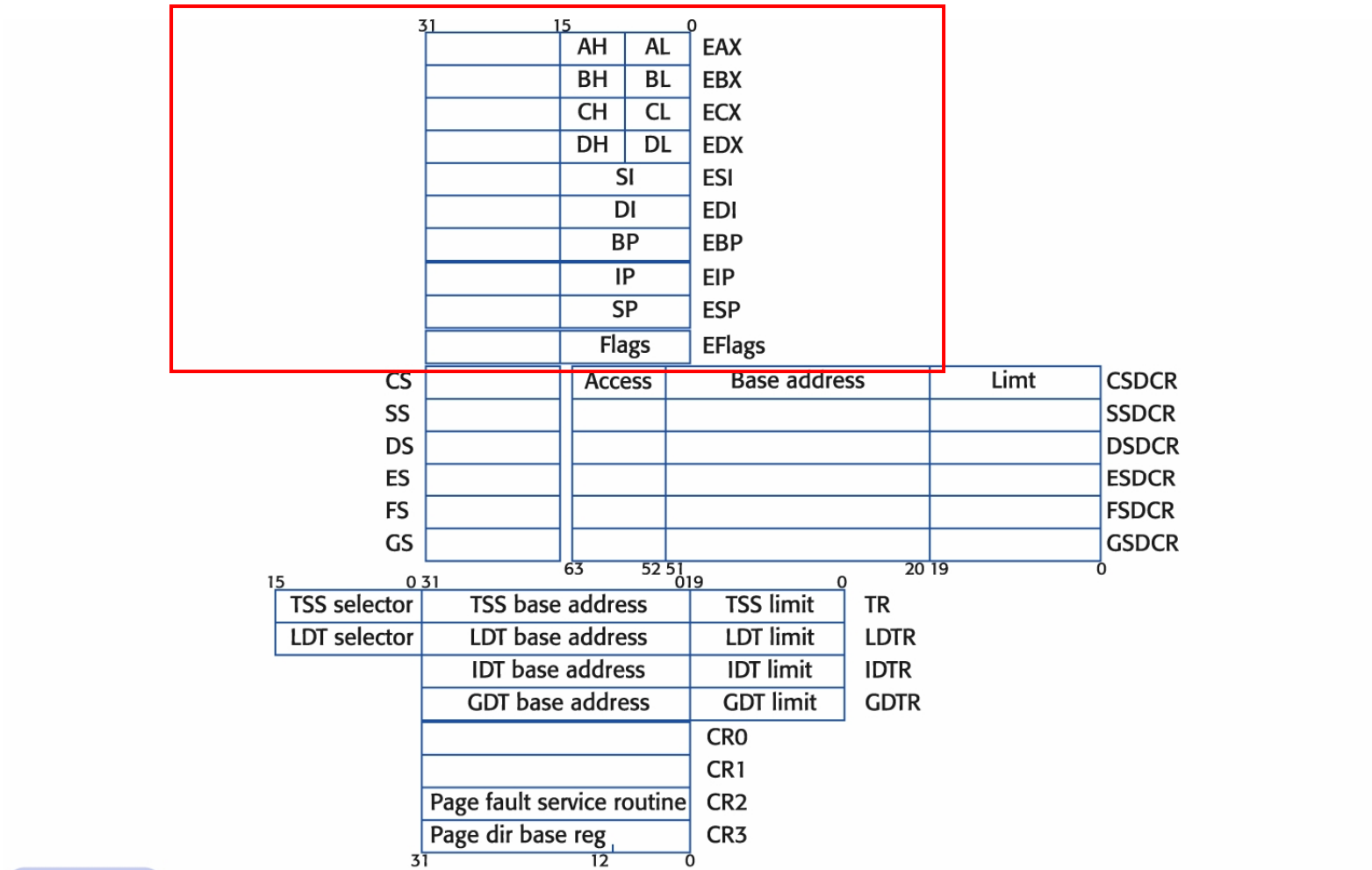
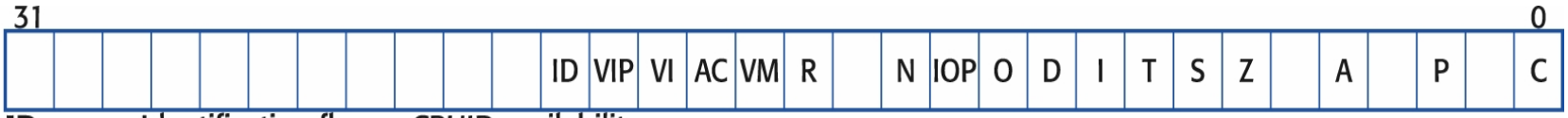


Fig. 7.5 Intel80x86/Pentium CPU register set.

CPU status flags

- **EFLAG**: The Flag register holds the CPU **status flags**.
- The status flags are separate bits in EFLAG where information on important arising conditions such as **overflow** or **carry bits**, is recorded.
- A way of communication between one instruction and the subsequent instructions.

CPU status flags



ID	Identification flag or CPUID availability
VIP	Virtual interrupt pending
VI	Virtual interrupt active
AC	Alignment check
VM	Virtual 8086 mode active
RFR	Resume task after breakpoint interrupt
NT	Nested task
IOPL	IO privilege level
O	Arithmetic overflow error
D	Direction of accessing string arrays
IE	External interrupt enable
T	Trap, single-step debugging, generates an INT #1 after each instruction
S	Sign, MS bit value
Z	Zero, result being zero
A	Auxiliary carry, used by BCD arithmetic on 4 LS bits
P	Parity, operand status
C	Carry, indicates an arithmetic carry or borrow result

Fig. 7.9 CPU status flags.

CPU status flags

- Most often used flags are:
  - S: sign.
  - Z: zero, result being zero.
  - C: carry, indicates an arithmetic carry.
  - O: arithmetic overflow error.
    - The addition of two large positive numbers which may give a negative result in a Two's complement system.

Arithmetic overflow error

- 0111011111111111
- +0110111101110111
- 1110011101110110

- OF will be set.

How CPU flags are operated

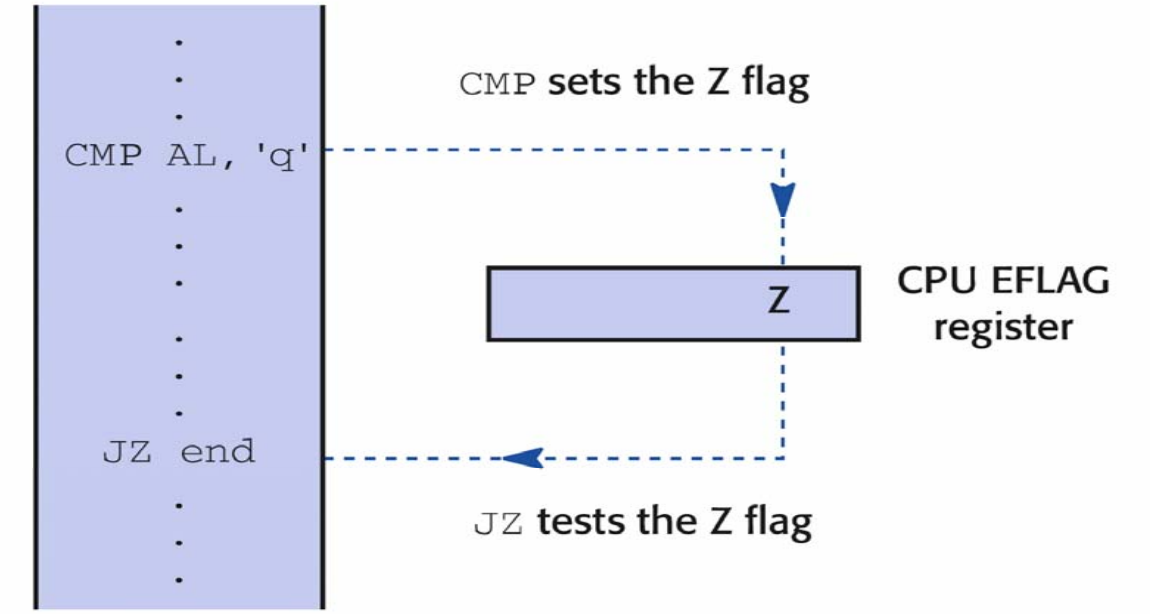


Fig. 7.10 How CPU status flags operate.

- CMP: Compare two values, subtract with no result, only setting flags.
- JZ: Conditional jump according to the Z flag.

Inline Assembler

- Practical sessions designed as part of our self-learning, self-studying exercises
- For the practical sessions we use **inline assembler** within *Microsoft Visual C++*.
- Inline means you can insert assembly codes directly into C/C++ programs, compile and execute them.
- Minimal knowledge of C/C++ is assumed.

Inline Assembler (cont.)

- Inline assembly code can refer to C or C++ variables by name:

```
_asm mov eax,var      ; stores the value of
                      ; var in EAX.
```

- This example illustrates two more aspects:
  - One can use **_asm** without brackets, in that case it works only for one line.
  - Semicolon **;** is used for the comments in the assembly program. Alternatively, one can use // for the comments.

Inline Assembler (cont.)

- An **`_asm{ ... ...}`** block can call C functions, including C library routines.
- We use C library routines **`scanf`** and **`printf`** for input and output in our programs.

- To pass arguments to **`printf`** we use a **`stack`**.
- It is time to discuss in more details the important idea of stack.

printf & scanf

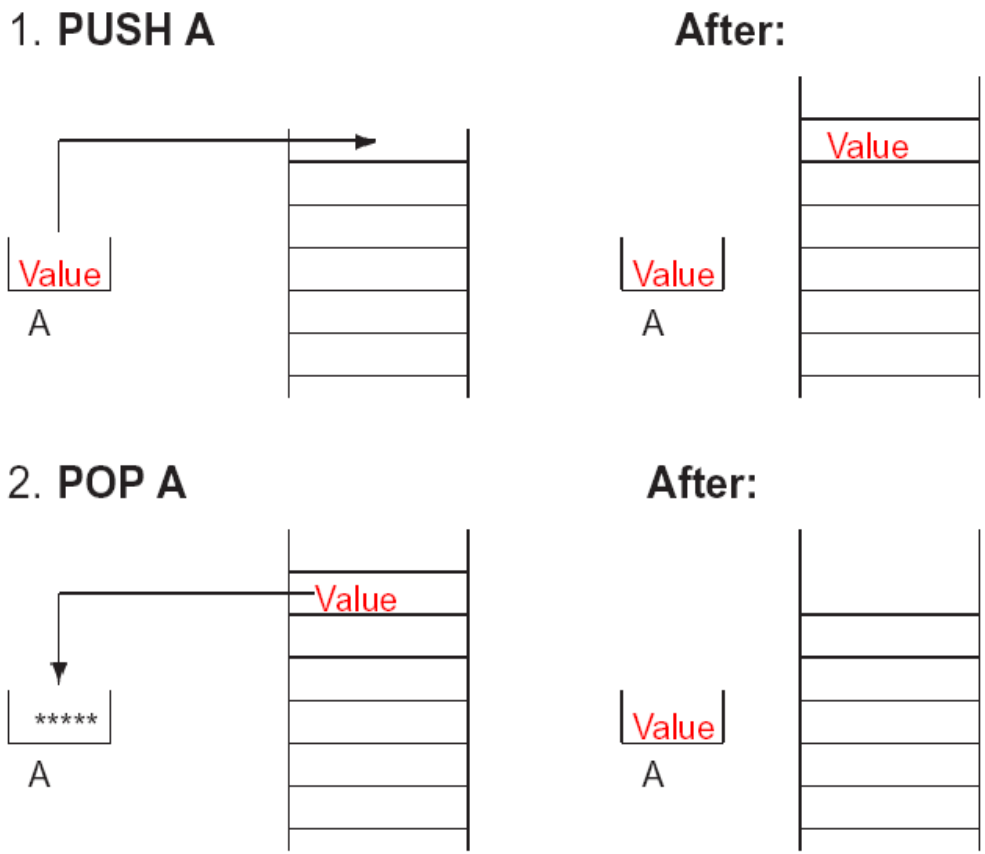
```
int x;
int day, year;
char monthname[20];

printf("%d\n", x);      /* how many parameters? */

scanf("%d %s %d", &day, monthname, &year);
                        /* input parameters */
```

Stack

- **Stack** is a memory arrangement (data structure) for storing and retrieval information (values).
- The order of storing and retrieving the values for the stack can be described as **LIFO** (last in, first out).
- The **ESP** register stores the address of the item that is on top of the stack.
- Example of an alternative memory arrangement is a **queue**: **FIFO**.

Stack (cont.)

- Every stack is equipped with two operations:
- **Push** and **Pop**.



Stack (cont.)

- Stack is a simple but useful data structure.
- Almost any assembly language has special instructions for implementing a stack.

- In inline assembly language there are **PUSH** and **POP** instructions.
- PUSH and POP operations use the stack pointer register ESP to hold the address of the item on top of the stack.

Assuming an upside-down stack… PUSH and POP

- Assume an **upside-down stack** in memory..
- PUSH instruction works:
  - First by decrementing the address in ESP.
  - Then using the ESP address to write the item into stack.
- POP instruction works:
  - The item is read using ESP address.
  - The ESP address is incremented by a correct amount to remove the item from the stack.

Using PUSH and POP

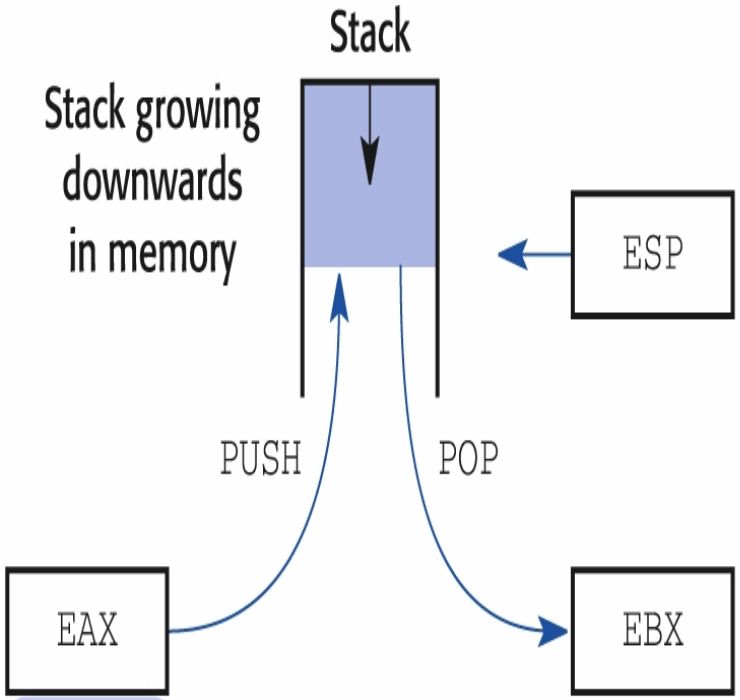


Fig. 8.6 Using PUSH and POP stack instructions.

```
PUSH EAX  ;push 32bit word in A into stack
CALL print;do something
POP EBX   ;pop the 32bit word from stack
```

Adjusting stack pointer

- Sometimes the stack pointer has to be adjusted explicitly by the programmer.

```
ADD ESP,4          ; take 4 bytes off the stack
                   ; (see example in the Hello
                   ; World program).

SUB ESP,256        ; create 256 bytes of
                   ; stack space.
```

Stack as the temporary store

```
 /* demonstration of the use of asm instructions within a C program */
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
   char format[] = "Hello World\n“;     // declare variables in C
   _asm{
        mov ecx,10                              // initialize loop counter
    Lj: push ecx                    // loop count index saved on stack
        lea eax,format              // load the address of the array
        push eax                    // address of string, stack parameter
        call printf                 // use library code subroutine
        add esp,4                   // clean 4 byte parameter off stack
        pop ecx                     // restore loop counter ready for test
        loop Lj                     // dec ECX, jmp back if ECX nonzero
   }                                // back to C
   return 0;
}
```

Stack as a temporary store (cont.)

- In the above example stack was used as a ‘scratch-pad’ temporary store to keep the value of loop counter and free ECX register for other use.

- That is a common use of the stack as a temporary store for variables (values).

Another use: passing parameters

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
   char format[] = "Hello World\n";  // declare variables in  C
   _asm{
        mov ecx,10                   // initialize loop counter
    Lj: push ecx                     // loop count index saved on stack
        lea eax,format               // load the address of the array
        push eax                     // address of string, stack parameter
        call printf                  // use library code subroutine
        add esp,4                    // clean 4-byte parameter off  stack
                                     // same as printf ("Hello World\n“)in C
        pop ecx                      // restore loop counter ready for test
        loop Lj                      // dec ECX, jmp back if ECX nonzero
   }                                 // back to C
   return 0;
}
```

What happens if we don’t “**add esp,4” after finishing the call to ‘printf’?**

Passing parameters

- `printf` routine needs
  - extra information on what to print and how to print.
  - It should be passed as parameters.
- It is done via stack:
  - **`push eax`** places the address of the string to be printed on the stack.
  - **`call printf`** reads the address from the top of the stack and prints the string, but it does not remove this address from the stack.
  - **`add esp,4`** cleans the top of stack (remove address).

Q&A

- Name 3 use cases of the Flag register in Pentium.
- The Flag register can be used to pass information between one instruction and the subsequent instructions. (T or F?)
- Which register is used to store the result of subtraction from this instruction? CMP AL, BL
- Under inline assembly, what mechanism is used to pass arguments to **`printf`** routine ?

- Q. Inline assembler can be used to call a C library function within an assembly segment in a C program (True or False).

- Q. Status flags are set (or cleared) before an instruction is being executed. (True or False)

- Q. D Flag is used to set the direction of looping.

Readings

- [Wil06] Section 7.5 for Eflag register.
- [Wil06] Sections 8.2, 8.4 for stack general information.
- [Wil06] Section 8.5 for stack used to pass parameters.
- [Wil06] Section 7.3 for basic Pentium instruction.
- [Wil06] Section 3.6 for memory organisation.