ALP Subjects

13) jump, conditional jump and loop instructions

JMP short [-128;127]
near
far

call: JMP label;

Conditional jumps:
- are of type short;
- if jump>128, replace condition with the negated one and use the JMP statement
- CMP command used for comparing values (for compare & jump)

<table>
<tr><th>Type</th><th>Instruction</th><th>Condition</th><th>Type</th><th>Instruction</th><th>Condition</th></tr>
<tr><td rowspan="11">Compare & Jump</td><td>JE</td><td>ZF=1</td><td rowspan="11">Flag based</td><td>JO</td><td>OF=1</td></tr>
<tr><td>JNE</td><td>ZF=0</td><td>JNO</td><td>OF=0</td></tr>
<tr><td>JG</td><td>ZF=0 & SF=OF</td><td>JC</td><td>CF=1</td></tr>
<tr><td>JL</td><td>SF<>OF</td><td>JNC</td><td>CF=0</td></tr>
<tr><td>JGE</td><td>SF=OF</td><td>JZ</td><td>ZF=0</td></tr>
<tr><td>JLE</td><td>ZF=1 || SF=OF</td><td>JNZ</td><td>ZF=1</td></tr>
<tr><td>\\\\\\\\</td><td>\\\\\\\\\\\\\\\\\\\\\\\\\\</td><td>JS</td><td>SF=1</td></tr>
<tr><td>\\\\\\\\</td><td>\\\\\\\\\\\\\\\\\\\\\\\\\\</td><td>JNS</td><td>SF=0</td></tr>
<tr><td>\\\\\\\\</td><td>\\\\\\\\\\\\\\\\\\\\\\\\\\</td><td>JP/JPE</td><td>PF=1</td></tr>
<tr><td>\\\\\\\\</td><td>\\\\\\\\\\\\\\\\\\\\\\\\\\</td><td>JNP/JPO</td><td>PF=0</td></tr>
<tr><td>\\\\\\\\</td><td>\\\\\\\\\\\\\\\\\\\\\\\\\\</td><td>JCXZ</td><td>CX=0</td></tr>
</table>

LOOP label (will decrement CX)
LOOPE/LOOPZ label (ZF=1)
LOOPNE/LOOPNZ label (ZF=0)

14) **software interrupts**

Although the terms trap and exception are often used synonymously, we will use the term *trap* to denote a programmer initiated and expected transfer of control to a special handler routine. In many respects, a trap is nothing more than a specialized subroutine call. Many texts refer to traps as *software interrupts* . The 80x86 int instruction is the main vehicle for executing a trap. Note that traps are usually *unconditional* ; that is, when you execute an int instruction, control *always* transfers to the procedure associated with the trap. Since traps execute via an explicit instruction, it is easy to determine exactly which instructions in a program will invoke a *trap handling* routine.

Software interrupts use an INT vector table as the one drawn.
The applied interrupt is called as INT n where n=[0,255]
This will load in IP=[n*4]
CS=[n*4+2]

Push flags in stack
IRET finishes the INT n and after which
Flags<- Stack
CS<- Stack
IP<-Stack

15) **procedures**

- Snippets of code that execute specific tasks
- Can be modularized

```
Syntax:   label PROC  [NEAR | FAR]
                  Instructions
          RET [constant]
          Label ENDP
```

Call: CALL label

16) **macro definitions**

- a macro is a pseudo-operation which permits a repeated inclusion of a code snippet in the main program
- are executed "in – line" (sequential)

```
Syntax:  name MACRO {parameters}
         LOCAL      list of local labels
                ……………
                Instructions
                ……………
         ENDM
```

Call: macro_name {parameters};

17) **macro and procedure libraries**

LIB program:

- + add modulename to the library
- - remove modulename from the library
- * extract modulename without removing it
- -+ or +- update modulename in library
- -* or *- extract modulename and remove it

Methods of executing LIB:

a) By answers to the console
   - after executing "LIB" command in prompt, program will be loaded and will print 3 prompts, to which the user has to type in the requested answers

b) By command line
   - LIB <library> <operations> <listing>

   Example: LIB PASCAL –HEAP +HEAP
   - if no listing option present, it will be considered null by default

c) Automatic answers
   - answers will be placed in a text file, which will be the parameter of the following line

   Syntax: LIB @<file_name>

18) **single and multiple segment programs**

- contains >=1 segments;
- 4(6) segments active at one time

NEAR procedure: IP, state register -> saved on stack; CS remains untouched, not saved
FAR procedure: IP, CS, state register -> saved on stack;

Example:

```
STCK  SEGMENT PARA STACK 'stack'
      DB          64 DUP ("my_stack")
STCK  ENDS

DATA1       SEGMENT PARA PUBLIC 'data'
; data definitions
DATA1       ENDS

COD1  SEGMENT PARA PUBLIC  'code'
MAIN  PROC  FAR
ASSUME      CS: COD1, DS:DATA1, SS:STCK
                  PUSH  DS
                  XOR         AX,AX
                  PUSH  AX
                  MOV         AX,DATA1
                  MOV         DS,AX
RET
MAIN  ENDP
COD1  ENDS
END   MAIN
```

19) **single and multiple module programs**

a) Procedures
- defined in the same file, different or same code segment, or in different files
- for procedures defined in different files:

  CODE SEGMENT PARA PUBLIC 'CODE'

```
PUBLIC name
ASSUME CS:CODE
name PROC [NEAR/FAR]
……………………………
    Instructions
……………………………
RET
name ENDP
CODE ENDS
END
```

Also, before main program, extern procedure must be specified:

```
EXTRN name: [NEAR/FAR]
……………..
```

Passing parameters to procedures:
- trough registers
- trough memory/pointer
- trough stack

b) Macros
- can be defined in the same file as the main project or in different file
- if defined in different file, the syntax remains unchanged. The only difference is we must insert the following code snippet before the main program:

```
IF1
        INCLUDE path_to_file_containing_macro
ENDIF
```

20) **protected mode operation of I-80x86 processor**

- space for physical address is 16Mb
- segments may have variable length (max 64k)
- can start from any address (not just a paragraph address)
- enables multitasking
- offers protection methods between programs running at the same time

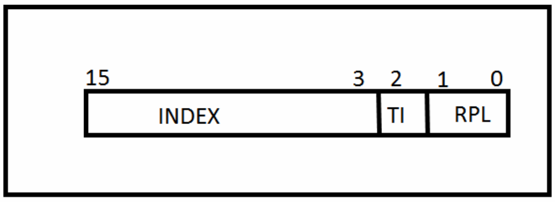
21) **memory management**

- memory management system translates virtual addresses into physical addresses
- segment registers length has not been modified, but instead of containing segment addresses, they now contain **selectors**. (indexes in some system tables which contain **segment descriptors**)

Modes:
- real mode: 1 MB space for physical addresses; 64 KB segment length;
- protected mode: 16MB space for physical addresses; 64 KB segment length;
- virtual mode: 1GB space for virtual addresses; 64 KB segment length;

22) **main data structures used in protected mode operation**

a) Segment selector



RPL = Request Privilege Level
TI = Table Indicator
INDEX = contains index in table GDT (TI=0) or LTD(TI=1)

Tables can be:
- GTD (Global Description Table)
- LTD (Local Description Table)
- ITD (Interrupt Description Table)

There exist 4 privacy levels, for protecting users and tasks:

Level 0: OS Core
Level 1: System routines
Level 2: OS extensions
Level 3: user tasks

b) Segment descriptor

Can be:
- code and data descriptors
- system descriptors
- gate descriptors

b.1 Code and data descriptors
- can execute only code snippets;
- those segments cannot be written, only read or load instruction code
- data segments can extend to high addresses

Offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LIM 15…0															
2	BAZA 15…0															
4	P	DPL			DT	TIP			BAZA 23…16							
6	BAZA 31…24								A	D	G	AV	LIM 19…15			

b.2 System descriptors
- were implemented for commuting speed between tasks in the multi-tasking system
- can be of multiple type:
  - 0 – invalid descriptor;

- 1 – descriptor TSS (TSS segment available)
- 2 – LTD descriptor;
- 3 – TSS descriptor (TSS segment not available)

b.3 Gate descriptors

- used for **call gates**, which are used for system function calls.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OFFSET 15…0															
SELECTOR															
P	DPL			0	TIP			000			no. words				
OFFSET 31…16															

23) **system function calls**

1. BIOS functions

INT 10h - This interrupt facilitates the use of the video terminal
- Input subfunction code: AH

INT 11h - returns in AX a word (16 bits) with information about the system's peripheral devices
INT 12h - stores in AX the amount of RAM memory in kilobytes
INT 13h - allows writing and reading disk sectors directly, without considering the existent file-system on the disk.
INT 14h - This interrupt facilitates access to the system's serial interface
INT 15h - assures control for the peripherals regarding their state (on/off).
INT 16h - used both for reading characters from keyboard and for obtaining keyboard's current state
INT 17h - allows access to parallel ports
INT 19h - After POST the processor executes the code for this interrupt by trying to read a code named bootstrap from the floppy or from the hard disk.
INT 1Ah - access to system's clock, for reading and setting the time.

2. DOS – INT21h functions, for keyboard and monitor

00h - End execution of a program
01h - Read character from keyboard and send it in echo to screen. (AL) – inserted character
02h - Show character on screen. (DL) – the character
09h - Show a row from memory ending with $ (24h) (DS:DX) – row address

- BIOS functions save registers CS, SS, DS, ES, BX, CX, DX, and destroy the others, so the user must save them and rebuild them.
- Before modifying the monitor's functioning regime it is recommended to save the current attribute and reset it at the end.

24) **math coprocessor structure and operation**

Math Coprocessor Operation

- Shares the same Data, Address and Control BUS as the main processor

- Two different chips for old processors
- On the same silicon die starting with 486DX
- Instructions preceded by an ESC sequence
-Operates in parallel with main processor
- Coprocessor may overtake BUS for longer periods if more data is needed
- Coprocessor has no access to registers but can use registers for addressing
- All addressing modes are available except immediate addressing
- Uses special synchronization signals to cooperate
- Instructions take tens to hundreds of cycles to complete

Math Coprocessor Structure

- Register Stack
- Control Register
- Status Register
- Tag Register
- Instruction Pointer
- Data Pointer

25) **math coprocessor data types**

- Integer numbers in C2
- Integer numbers in Packed Decimal
- Real Numbers in IEEE 754/854 standard format

All these representations are ONLY in memory. Internally ALL numbers are represented on 80 bits as temporary real's

Integer Data Types

- Word Integer DW 16 bit C_2 representation
- Short Integer DD 32 bit C_2 representation
- Long Integer DQ 64 bit C_2 representation
- Packed Decimal DT 80 bit Value & Sign

26) **math coprocessor data transfer and constant load instructions**

LOAD instructions
FILD adr - Loads on the stack the integer variable located at address „adr". The variable stored in memory of type DB, DW, or DD is converted to the coprocessor's internal format.
FLD adr - Loads on the stack the real variable (long or short) located at address „adr". The variable stored in memory of type DD, DQ, or DT is converted to the coprocessor's internal format.
FBLD adr - Loads on the stack the packed decimal variable located at address „adr". The variable stored in memory of type DT is converted to the coprocessor's internal format.

STORE instructions
FIST adr -Stores at the address „adr" the value located on top of the stack (ST (0)) as a number. The stored value can be only an integer represented on one byte or a short integer, corresponding to the

data stored at address „adr” (DW or DD). The stack pointer remains unchanged after the data is stored. The conversion is done during the store process.
FISTP adr - Stores at the address „adr” the value located on top of the stack (ST (0)) as an integer number. The stored value can be any integer (byte integer, short integer, long integer) corresponding to the data stored at address „adr” (DW, DD or DQ). The conversion is done during the store process. The instruction changes the stack: ST (0) is popped and the stack pointer is decremented.
FST adr - Stores at the address „adr” the value located on top of the stack (ST (0)) as an integer number. The stored value can be either a short integer or a double precision, corresponding to the data stored at address „adr” (DD or DQ). The stack pointer and the data on the stack remain unchanged after the data is stored. The conversion is done during the store process.
FSTP adr - Stores at the address „adr” the value located on top of the stack (ST (0)) as a floating point number. The value can be a short real with double or extended precision, corresponding to the data stored at address „adr” (DD, DQ or DT). The conversion is done during the store process. The instruction changes the stack: ST (0) is popped and the stack pointer is decremented.
FBSTP adr - Stores at the address „adr” the value located on top of the stack (ST (0)) as a packed decimal number (defined at “adr” with DT). The stack pointer is decremented. The conversion is done during the store process.

<u>Internal data transfer instructions</u>
FLD ST (i) - Puts the value from ST (i) on top of the stack. Thus the value from ST (i) will be found twice: in ST (0) and ST (i+1).
FST ST (i) - The value from ST (0) is copied in the stack’s “i_{th}” position. The old ST (i) is lost.
FSTP ST (i) - The value from ST (0) is copied in the stack’s “i_{th}” position. The old ST (i) is lost. ST (0) is popped, the stack pointer is decremented.
FXCH ST (i) - swaps ST (0) and ST (i).

<u>Constants loading instruction</u>
FLDZ - Loads 0 in the top of the stack
FLD1 - Loads 1.0 in the top of the stack
FLDPI – Loads π (‘pi’) in the top of the stack
FLDL2T - Loads $\log_2 10$ in the top of the stack
FLDL2E - Loads $\log_2 e$ in the top of the stack
FLDLG2 - Loads $\log_{10} 2$ in the top of the stack
FLDLN2 - Loads ln(2) in the top of the stack

27) **math coprocessor arithmetic and control instructions**

Arithmetic instructions
FADD ST (0) ←ST (0) +ST (1)
FADD op ST (0) ←ST (0) +”op” from memory or stack. Floating point operation.
FADD op ST (0) ←ST (0) +”op” from memory or stack. Integer operation.
FADD ST (i), ST (0) ST (i) ←ST (i) +ST (0); ST (0) popped
FSUB ST (0) ←ST (0) -ST (1)
FSUB op ST (0) ←ST (0) –„op” from memory or stack. Floating point operation.
FSUB op ST (0) ←ST (0) -“op” from memory or stack. Integer operation.
FSUB ST (i), ST (0) ST (i) ←ST (i) -ST (0); ST (0) popped
FSUBR ST (i) ST (i) ←ST (i) -ST (0) ; opposite instruction of FSUB ST (i)

FMUL ST (0) ←ST (0) * ST (1)
FMUL op ST (0) ←ST (0) * ”op” from memory or stack. Floating point operation.
FMUL op ST (0) ←ST (0) x ”op” from memory or stack. Integer operation.
FMULP ST (i), ST (0) ST (i) ←ST (i) x ST (0); ST (0) popped
FDIV ST (0) ←ST (0): ST (1)
FDIV op ST (0) ←ST (0):”op” from memory or stack. Floating point operation.
FDIV op ST (0) ←ST (0):”op” from memory or stack. Integer operation.
FDIVP ST (i), ST (0) ST (i) ←ST (i): ST (0); ST (0) popped
FDIVR ST (i) ST (i) ←ST (i): ST (0);opposite instruction of FDIV ST (i).

Command Instructions
FINIT Initialization - the coprocessor is brought in an initial status known as ‘software reset’.
FENI accept interrupt
FDISI Ignore interrupt - this instruction ignores all interrupts regardless of the command register’s bits;
FLDCW adr The command register is loaded from the memory location indicated by ‘adr’
FSTCW adr The command register is saved in a word located at the memory location indicated by **‘**adr’
FSTSW adr The status register is saved in a word located at the memory location indicated by **‘**adr’.
FCLEX The bits that define the exceptions are erased
FSTENV adr Save the environment - the coprocessor’s internal registers are saved in a memory location starting at address ‘adr’ that has a size of 14 bytes.
FLDENV adr Load environment
FSAVE adr Save status - the coprocessor’s internal registers and its stack are saved in a memory location starting at address ‘adr’ that has a size of 94 bytes.
FRSTOR adr Load status.
FINCSTP Increment stack pointer- after this instruction, the stack pointer is incremented with 1;
FDECSTP Decrement stack pointer
FFREE ST (i) Delete the i_{th} element in the stack. The operation does not influence the stack pointer.
FNOP No operation.
FWAIT Waits for the current action to finish (similar to the 8086’s WAIT instruction)

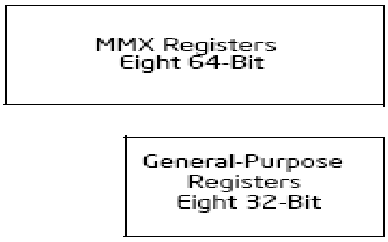
28) **math coprocessor mathematical functions**

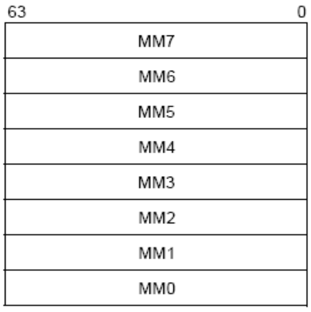
Floating point functions
FSQRT Square root – ST(0)‘s square root is put in ST(0).
FSCALE 2’s power. Puts in ST(0) the ST(0)’ s value multiplied with $2^{**}_{ST(1)}$
FPREM Partial remainder. ST(0) is divided by ST(1) and stored in ST(0).
FRMDINT Round. ST(0) is replaced with ST(0) rounded.
FXTRACT The value stored in ST(0) is split into characteristic (in ST(0)) and mantissa (in ST(1)).
FABS ST(0) is replaced with its absolute value.
FCHS ST(0)’s sign is changed.
FPTAN Partial tangent. The tangent of the angle contained in ST(0) is determined as fraction of ST(1) /ST(0). The initial value of the angle contained in ST(0) must be between 0 and $\pi/4$.
FPATAN Partial Arctangent. The arctangent of the value ST(1)/ST(0) is stored in ST(0). The initial value contained in ST(0) must be positive, while ST(1) must be larger than ST(0).
F2XM1 calculates 2’s powers. ST(0) will be replaced by $2^{**}_{ST(0)}$-1.
FYL2X Logarithm. ST (0) ←ST (1)*$\log_2$(ST (0)).
FYL2XP1 Logarithm. ST (0) ←ST (1)* $\log_2$ (ST (0) +1).

29) **MMX extensions, data types and operating principles**

Extensions

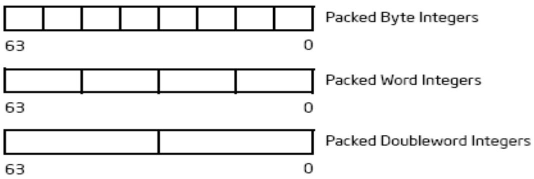
- Eight new 64-bit data registers, called MMX registers
- Three new packed data types:
  - — 64-bit packed byte integers (signed and unsigned)
  - — 64-bit packed word integers (signed and unsigned)
  - — 64-bit packed doubleword integers (signed and unsigned)
- Instructions that support the new data types and to handle MMX state
- Management
- Extensions to the CPUID instruction





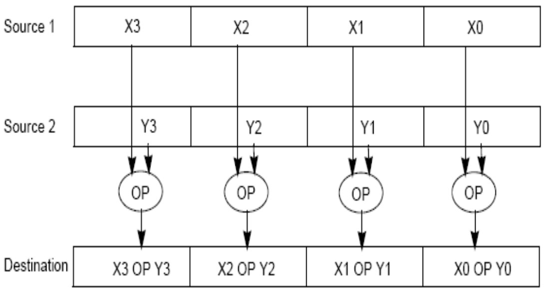
Data Types

- 64-bit packed byte integers — eight packed bytes
- 64-bit packed word integers — four packed words
- 64-bit packed doubleword integers — two packed double words



Execution Models

- MMX instructions move 64-bit packed data types (packed bytes, packed words, or packed double words) and the quadword data type between MMX registers and memory or between MMX registers in 64-bit blocks
- However, when performing arithmetic or logical operations on the packed data types, MMX instructions operate in parallel on the individual bytes, words, or double words contained in MMX registers



30) **multimedia calculus and main instruction families**

The MMX instruction set consists of 47 instructions, grouped into the following categories:

- Data transfer
- Arithmetic
- Comparison
- Conversion
- Unpacking
- Logical
- Shift
- Empty MMX state instruction (EMMS)

31) **program optimization, general issues, optimization levels**

- Optimisation is expensive in labour and time
- Is an open ended process may fail or not reach desired performance
- Various optimisation techniques used in different development phases
- Some of them need a lot of experience
- Tools are available

Types of optimisation

- High Level - Choose a better algorithm
  Research and development
  May not find a better one and have to design one
- Medium Level - make a better implementation
  Language independent but best performance improvement obtained in assembly language
- Low level (cycle counting)
  Heavily hardware dependent

32) **non optimal code determination methods**

Trial and Error - Educated guess
Program trace and analysis - Hard to find recursive functions
Optimize everything - Only for small projects
Profiler

33) **optimization techniques in ALP**

A better implementation generally involves steps like:

- unrolling loops,
- using table lookups rather than computations
- eliminating computations from a loop whose value does not change within a loop
- taking advantage of machine idioms (such as using a shift or shift and add rather than a multiplication) trying to keep variables in registers as long as possible