Exam topics

1. Assembler Operation

- The simplest programming language uses **labels** and **mnemonics** for human readability.
- Mnemonics correspond 1:1 to machine code instructions and are translated by the assembler.
- The general structure of a line is `{Label optional} {Mnemonic {Operands}} {;Comments}`.
- During the **assembly** process, the assembler translates mnemonics into machine code, creating an object file.
- Assembly language is machine-specific because machine code is machine-specific.
- Memory elements are accessed as symbols.
- **Two-pass assemblers** can use **Symbol Tables** and **Location Counters**, which are incremented by the instruction size at each step.
- **One-pass assemblers** require all symbol definitions *before* usage for correct functioning.

---

2. Main Assembler Pseudo-Instructions

- **Pseudo-instructions** are commands for the assembler that do not translate into machine code.
- A **pseudo-opcode** is a directive to the assembler.
- **Pseudo-instructions to define segments:**
  General form

  ```
  segment_name SEGMENT [align_type] [combine_type] ['class']
  ...
  segment_name ENDS
  ```

  Example

  ```
  CODE SEGMENT PARA PUBLIC 'CODE'
  ...
  CODE ENDS
  ```

    - `segment_name` is a label associated with the segment's memory position.
    - `align_type`: `PARA`, `BYTE`, `WORD`, `PAGE`.
    - `combine_type`: `PUBLIC`, `COMMON`, `STACK`, `MEMORY` (same as `PUBLIC`), `AT expression`.
    - `'class'` helps the linker arrange segments of the same class. Recommended values: `'data'`, `'code'`, `'stack'`, `'memory'`.
- **Designate active segment** (communicates symbol definition to the assembler, doesn't load it):

General form

```
ASSUME <seg_reg> : <seg_name>, <seg_reg> : <seg_name>...
```

Example

```
ASSUME CS:CODE, DS:DATA
```

- **Memory location reservations** are pseudo-instructions:
    - **Define & allocate space in memory:**

      ```
      <name> <type> [expression_list] ;Optional: [<factor> DUP
      (<expression_list>)]
      ```

      Example

      ```
      NUM1 DB 10 ; byte with value 10
      ARRAY DB 1, 2, 3, 4, 5 ; 5 bytes long vector of numbers
      STRING ? DUP (10) ; 10 byte long vector that is uninitialized
      ```

        - `<name>`: symbol's name.
        - `<type>`: symbol's type: `DB` (1 byte), `DW` (2 bytes), `DD` (4 bytes), `DQ` (8 bytes) (**define quadword**), `DT` (10 bytes).
        - `<expression_list>`: initialization memory value. Use `?` for no initialization.
        - `<factor>`: how many times to duplicate.
    - **Define symbols (constants):**

```
<name> = <expression>
; or
<name> EQU <expression> (redefinable)
```

- **Declare labels:**

```
<name> LABEL <type>
```

  - `<type>` may be: `BYTE`, `WORD`, `DWORD`, `QWORD`, `TBYTE`, the name of a structure, `NEAR`, or `FAR`.
- **Location Counter (LC) modification:**
  - The LC will be changed to the expression's value.

```
ORG <expression>
```

- **Define procedure** (use `CALL` and `RET` at the end):

```
<proc_name> PROC [NEAR (implicit)], FAR
```

  - Either `NEAR` or `FAR`, not both.
  - Possibility: nested procedures.

---

3. Integer Number Representation in C1, C2, and Sign-Magnitude

- Integers are represented on bytes (8 bits), words (16 bits), double words (32 bits), or quadwords (64 bits).
- For all representations, the **MSB (Most Significant Bit)** is the **sign bit** and is extended.
- Two parts: **sign** (0 for positive, 1 for negative in all three forms) and **absolute value**.
  - **Sign-Magnitude (SM):** The first bit is the sign bit (0 or 1); the rest is the absolute value in binary. Zero has two representations ($+0$ and -0).
  - **One's Complement (C1):** If the number is positive, it's identical to the SM form. If negative, all bits of the number's absolute value are negated, and the sign bit is 1.
  - **Two's Complement (C2):** If the number is positive, the representation is the same as SM and C1. If negative, all bits of the number's absolute value are negated,

and 1 is added to the new value (hence, C1 representation + 1 in the case of negative numbers).

- Attention should be paid to the number of bytes needed for representing a number and sign extension.

---

4. Real Number Representation in IEEE Short Format

- The **IEEE standard** has three representations for real numbers: short (4 bytes), long (8 bytes), and temporary (10 bytes).
- Real numbers are represented in memory in the following format:

Sign	Characteristic	Mantissa
1 bit	8 bits	32 - 8 - 1 = 23 bits

- The sign bit is 0 if the number is positive, 1 if negative.
- The Characteristic is `exponent + 127` or $(127)_{10} = (7F)_{16}$.
- To represent a number, write it in binary and bring it to a form similar to scientific notation: $\text{Number} = 1.<\text{binary digits}> \times 2^{\text{exponent}}$
- Since the first bit before the decimal point is always 1, it is not stored.
- The mantissa is formed by `<binary digits>` extended to 23 bits at most, if not repeating, with 0s.
- Offers 7 decimal digits of precision.

---

5. I-8086 Addressing Modes

- Instructions may have the following pairs of operands:
  - Register-register
  - Register-memory
  - Register-immediate
  - Memory-immediate
- The 8086 provides 17 different ways to access memory; notable examples include:
  - **Displacement-only (direct) addressing mode:**
    - Most common.
    - A 16-bit value specifies the address of the target.

    - By default, all displacement-only values provide offsets into the data segment.

```
mov AL, DS:[8088h]
```

- **Base addressing mode:**
    - Based on a base register (`BX` or `BP`); can have displacement.

```
mov AL, disp[BX]
mov AL, disp[BP]
```

- **Indexed addressing mode:**
    - Indexed by an index register (`SI` or `DI`); can have displacement.

```
mov AL, disp[SI]
mov AL, disp[DI]
```

- **Indirect addressing mode:**
    - Accesses memory indirectly through a register using indirect addressing; can have displacement.
    - 4 possibilities on 8086:

```
mov AL, [BX]
mov AL, [BP]
mov AL, [SI]
mov AL, [DI]
```

    - Registers offset is referenced.

---

6. Physical and Effective Address

- A **physical address** (PA) is the address generated by the hardware after lookups of translation tables, handled by the OS.
- The 8086 uses **segmented addressing**. To translate a **logical address** to a physical one, the processor first calculates a **segment base address**. It then adds the offset (or **relative address**) to this base address to obtain the final physical address of the data.

- The used segment register needs to be specified with a **prefix byte** preceding the opcode.
- Format is `001xx110` where `xx` is:
  - `00`, for `ES`;
  - `01`, for `CS`;
  - `10`, for `SS`;
  - `11`, for `DS`.
- An **effective address** (EA) is the final memory address obtained after all addressing computations are performed (indexing, offsetting, etc.).
  - It is the final offset produced.
  - More complex addressing modes require more computation time to compute the EA.
- The memory address of an operand is composed of:
  - Starting address of segment + (displacement) + (base) + (index)
- PA is computed in the **Bus Interface Unit**.

---

7. CPU Registers and Their Functions

- **8 general-purpose registers:**
  - **AX** - **Accumulator:** Most arithmetic and logical operations take place here. Often the default register for results.
  - **BX** - **Base:** Commonly used to hold a base address.
  - **CX** - **Count:** It counts things, typically used for loops and string lengths.
  - **DX** - **Data:** A general-purpose register, but has a special role in `MUL` / `DIV` operations for holding data, also used for I/O addresses.
  - **SI** and **DI** - **Source** and **Destination Index:** Used for indexing and indirect memory access.
  - **BP** - **Base Pointer:** Similar to `BX`, typically used to access values on the stack.
  - **SP** - **Stack Pointer:** Always points to the top of the stack. It is automatically modified by `PUSH`, `POP`, `CALL`, and `RET`.
- **Segment registers:**
  - **CS** - **Code Segment:** Points to the segment containing currently executed instructions.
  - **DS** - **Data Segment:** Points to the global variables of the program.
  - **SS** - **Stack Segment:** Points to the segment containing the 8086 stack where procedure parameters and subroutine return addresses are stored.
  - **ES, FS, GS** - **Extra Segment Registers:** Point to additional data segments.

- **Special-purpose registers:**
  - **IP** - **Instruction Pointer:** It contains the address of the currently executing instruction (interpretable with `CS`).
  - **Flags register:** A collection of one-bit values that help determine the current state of the processor. It is 16-bits wide, but the 8086 uses only 9 of those bits:
    - **Control-** and **status flags**.
    - Examples: Overflow, Zero, Interrupt, Carry, Parity (all are 1 bit).

---

8. Arithmetical and Logical Instructions

- These instructions handle signed/unsigned operands represented in C2 for addition, subtraction, multiplication, division, etc.
- They also include packed and unpacked BCD (Binary-Coded Decimal) addition, subtraction, and adjustment, as well as unpacked BCD multiplication and division.
- These operations affect and typically use the flags register.
- **Addition instructions:**
  - `ADD dest, source`: equal to `dest` += `source`.
  - `ADC dest, source`: equal to `dest` += `source` + `carry`.
  - `INC operand`: Increments (adds 1) the operand, faster than `ADD`.
- **Subtraction instructions:**
  - `SUB dest, source`: equal to `dest` -= `source`.
  - `SBB dest, source`: equal to `dest` -= (`source` - `borrow`).
  - `DEC operand`: Decrements the operand by 1, faster than `SUB`.
- **Multiplication instructions:** Use a single operand; `AX` or `DX:AX` is the assumed destination.
  - `MUL operand`: Unsigned multiplication. If operand is 8-bit, `AX` = `AL` *`operand`*. *If operand is 16-bit,* *`DX:AX`* = *`AX`* `operand`.
  - `IMUL operand`: Same as `MUL` but signed.
- **Division instructions:** Quotient in `AL` / `AX`, remainder in `AH` / `DX`.
  - **8-bit Division:** The 16-bit dividend must be in `AX`.
  - **16-bit Division:** The 32-bit dividend must be in `DX:AX`.
  - Before division, you must extend the dividend into the full register pair:
    - `DIV`: Zero-extend the value (e.g., `MOV AH, 0`).
    - `IDIV`: Sign-extend using `CBW` (Byte to Word, copies bit 7 of `AL` throughout bits 8-15 of `AX`) or `CWD` (Word to Double, sign-extends `AX` into `DX:AX`).
    - Execute `DIV` or `IDIV reg/mem`.

- **Compare instructions:**
  - `CMP dest, source`: Compares, similar to `SUB`, but does not store the difference; only sets flags, usually used for jumps.
- `NEG operand`: Negates the operand (takes the two's complement of a byte or word).
- **Logical instructions:**
  - Bitwise operations useful for masking.
  - `AND`, `OR`, `NOT`, `XOR`.
- **Unpacked BCD (UBCD) adjustments:**
  - `AAA`, `AAS`, `AAM`, `AAD`: Adjusts for addition, subtraction, multiplication, and division, respectively.
- **Packed BCD (PBCD) adjustments:**
  - `DAA`, `DAS`: Adjusts for addition and subtraction, respectively.

---

9. Shift and Rotate Instructions

Every shift instruction either takes an immediate or ONLY THE CL register as the count parameter

- **Shift instructions:** Move bits around in a register or memory location. General format:

```
shl reg, count
```

  - `SHL` / `SAL` - **Shift Left/Shift Arithmetic Left:** Zeroes fill vacated positions at the LSB; the MSB shifts into the carry flag. Performs multiplication by two.
  - `SHR` - **Shift Right:** Shifts all bits in the destination operand to the right one bit, shifting a zero into the H.O. bit. The LSB shifts into the carry flag.
  - `SAR` - **Shift Arithmetic Right:** Shifts all bits in the destination operand to the right one bit, replicating the H.O. bit. The LSB shifts into the carry flag. Performs a signed division by two.
- **Rotate instructions:** Shift bits around, but the bits shifted out of the operand recirculate through the operand. General format:

```
rol reg, count
```

  - `ROL` - **Rotate Left:** Shifts the operand's MSB, rather than the carry, into bit zero. Copies the output of the MSB into the carry flag.

- `ROR` - **Rotate Right:** Shifts the LSB into the MSB, rather than the carry. The LSB is copied into the carry flag.
- `RCL` - **Rotate Through Carry Left:** Rotates bits to the left, through the carry flag and back into the LSB.
- `RCR` - **Rotate Through Carry Right:** Rotates bits to the right, through the carry flag and back into the MSB.

---

10. Data Transfer Instructions

- These instructions facilitate data movement between:
  - Register to register (`reg` <- `reg`)
  - Register to/from memory (`reg` <-> `memory`)
  - Immediate value to register (`reg` <- `immediate`)
  - Immediate value to memory (`mem` <- `immediate`)
  - Segment register to/from 16-bit memory location (`segreg` <-> `mem_16`)
  - Segment register to/from 16-bit general-purpose register (`segreg` <-> `reg_16`)
- `MOV` **operation:**
  - No direct memory-to-memory transfers.
  - No immediate value to segment register transfers.
  - Operands must be of the same size.
  - Immediate data is extended by the CPU to the required size.
- `XCHG` **instruction - Exchange:** Swaps two values; operands must be of the same size.

```
xchg reg, mem
xchg reg, reg
xchg ax, reg16
```

- `XLAT` **instruction - Translate:** Puts the translated value into `AL` based on a lookup table stored with base `BX` and index `AL`. No operand is specified.

```
AL = DS:[BX + AL]
```

---

11. Address Transfers and Stack Instructions

- Used for loading **effective** (16-bit) or **physical address** (32-bit) into registers.
- `LDS`, `LES`, `LFS`, `LGS`, `LSS` **(Load Pointer into Segment Register):**

  ```
  LXS reg_16 (destination register for offset), mem_32 (DW from memory)
  ```

  - Loads the upper word of the mem_32 into the corresponding segment register (LDS, LES, LSS, etc.) and the lower word into a register of choice. This is a quick way to set up segment register pairs for addressing.
- `LEA` **- Load Effective Address:** Used for pointer values.

  ```
  LEA dest, source
  LEA reg_16, mem ; explicit and better alternative to MOV reg_16, OFFSET
  mem
  ```

  - Loads the register with the Effective Address (EA) of the memory location.
- `SAHF`, `LAHF`: Stores `AH` to and loads `AH` from the flags register, respectively.
- **Stack instructions:** Manipulate the hardware stack.
  - `PUSH` **and** `POP`**:**
    - Operands are 16-bit registers, memory locations, or segment registers (except `CS` for `POP`).
    - `PUSH` puts a word onto the top of the stack:

      ```
      SP -= 2
      [SS:SP] = operand_16
      ```

    - `POP` removes a word from the top of the stack:

      ```
      operand_16 = [SS:SP]
      SP = SP + 2
      ```

    - `(SS:SP)` always contains the address of the value at the top of the stack.

---

12. Jump Instructions

- **Unconditional jumps -** `JMP`**:** Unconditionally transfers execution to another point in the program.

    - **Intrasegment jumps:** Between statements in the same `CS` (`NEAR`).
    - **Intersegment jumps:** Can be between different `CS` segments (`FAR`), changing both `CS` and `IP`.
- **Direct jumps:** Use a relative addressing scheme for short and near jumps. The target address is typically defined using a statement label.
    - **Short jump:** Transfers execution to an address within -128 to +127 bytes from the current instruction.
    - **Near jump:** Transfers execution to a label after the jump, within a $\pm 32\text{KB}$ distance in the same code segment.
    - **Far jump:** Transfers execution to a different segment.
        - If defined before: The instruction code is followed by a far pointer (segment:offset).
        - If defined after: Specify the target as `FAR` type.
- **Indirect jumps:**
    - **Near indirect:** Fetches a 16-bit offset from a register or memory into `IP`.
    - **Far indirect:** Fetches a 32-bit pointer (segment:offset) from memory into `CS:IP`.

---

13. Conditional Jump and Loop Instructions

- Similar to if/else, they allow the program to make decisions.
- They test flag values, typically used together with `CMP` (compare) instructions.
- They do not affect any flags.
- There are two categories: specific flag checks or testing conditions based on `CMP` results.
- Most are 2 bytes long and have a range of ± 128. For range extensions, a trick involving an unconditional jump (`JMP`) can be used. Replace it with an unconditional jump and wrap it with a conditional jump that is the ! (not) equivalent of the original jump.
- **Loop instructions:** (the `DEC` / `JNE` instructions are generally faster)
    - $\pm$ 128 bytes distance.

Instruction	Description	Condition	Aliases	Opposite
`JC`	Jump if carry	Carry = 1	`JB`, `JNAE`	`JNC`
`JNC`	Jump if no carry	Carry = 0	`JNB`, `JAE`	`JC`

Instruction	Description	Condition	Aliases	Opposite
`JZ`	Jump if zero	Zero = 1	`JE`	`JNZ`
`JNZ`	Jump if not zero	Zero = 0	`JNE`	`JZ`
`JS`	Jump if sign	Sign = 1		`JNS`
`JNS`	Jump if no sign	Sign = 0		`JS`
`JO`	Jump if overflow	Overflow = 1		`JNO`
`JNO`	Jump if no overflow	Overflow = 0		`JO`
`JP`	Jump if parity	Parity = 1	`JPE`	`JNP`
`JPE`	Jump if parity even	Parity = 1	`JP`	`JPO`
`JNP`	Jump if no parity	Parity = 0	`JPO`	`JP`
`JPO`	Jump if parity odd	Parity = 0	`JNP`	`JPE`
`JG`	Jump if greater (>)	(Sign = Overflow) or (Zero = 0)	`JNLE`	`JNG`
`JNLE`	Jump if not less than or equal (not $\leq$)	(Sign = Overflow) or (Zero = 0)	`JG`	`JLE`
`JGE`	Jump if greater than or equal ($\geq$)	Sign = Overflow	`JNL`	`JL`
`JNL`	Jump if not less than (not <)	Sign = Overflow	`JGE`	`JL`
`JL`	Jump if less than (<)	Sign $\neq$ Overflow	`JNGE`	`JNL`
`JNGE`	Jump if not greater than or equal (not $\geq$)	Sign $\neq$ Overflow	`JL`	`JGE`
`JLE`	Jump if less than or equal ($\leq$)	(Sign $\neq$ Overflow) or (Zero = 1)	`JNG`	`JNLE`
`JNG`	Jump if not greater than (not >)	(Sign $\neq$ Overflow) or (Zero = 1)	`JLE`	`JG`
`JA`	Jump if above (>)	(Carry = 0) and (Zero = 0)	`JNBE`	`JNA`
`JNBE`	Jump if not below or equal (not $\leq$)	(Carry = 0) and (Zero = 0)	`JA`	`JBE`

Instruction	Description	Condition	Aliases	Opposite
`JAE`	Jump if above or equal ($\geq$)	Carry = 0	`JNC`, `JNB`	`JNAE`
`JNB`	Jump if not below (not <)	Carry = 0	`JNC`, `JAE`	`JB`
`JB`	Jump if below (<)	Carry = 1	`JC`, `JNAE`	`JNB`
`JNAE`	Jump if not above or equal (not $\geq$)	Carry = 1	`JC`, `JB`	`JAE`
`JBE`	Jump if below or equal ($\leq$)	(Carry = 1) or (Zero = 1)	`JNA`	`JNBE`
`JNA`	Jump if not above (not >)	(Carry = 1) or (Zero = 1)	`JBE`	`JA`

- **`LOOP label`**:
  - Decrements `CX`.
  - If `CX` $\neq 0$, then `JMP label`.
  - If `CX` is initially zero, this will behave wrongly (executes 2^{16} iterations).
- Typically used with a condition tested with `CMP`:
  - **`LOOPE label`** / **`LOOPZ label`**:
    - Decrements `CX`.
    - If (`CX` $\neq 0$) and (`ZF` = 1), then jump.
  - **`LOOPNE label`** / **`LOOPNZ label`**:
    - Decrements `CX`.
    - If (`CX` $\neq 0$) and (`ZF` = 0), then jump.

---

14. Software Interrupts

- There are three types of interrupts: **traps**, **exceptions**, and **interrupts**.
- **Trap:**
  - Is a software-invoked interrupt, usually unconditional.
  - Handled by an **Interrupt Service Routine (ISR)**.
  - Can be interpreted as a **specialized subroutine call** that uses `INT` instead of `CALL`.

- The `INT` instruction is the main one for executing a trap:

```
INT <num>_8bit
```

- `IP` = [`num` * 4], `CS` = [`num` * 4 + 2].
- Allows for calling one of the 256 **interrupt routines**: loads a new `IP` and `CS`, and pushes the flags to the stack.
- **Flow of calling an interrupt:**
  - Flags register pushed to stack.
  - `CS` and `IP` pushed to stack.
  - The processor uses the number to copy the double-word from the **interrupt table** into `CS:IP`.
  - `IRET` is used for return.

---

15. Procedures

- Procedures are snippets of reusable code that execute specific functions.
- Comparable with C functions.

```
proc_name PROC [NEAR (implicit) | FAR]
    ; Instructions
    RET [constant]
proc_name ENDP

; --------------
; Call to procedure:
CALL label
```

- `proc_name` must be unique in the program.
- `CALL` - pushes the return address onto the stack and jumps to the address of the called procedure.
- `RET` - pops the address saved by `CALL` from the stack and returns control to the instruction after `CALL`.
- **Ways of transmitting parameters:**
  - Registers
  - Memory (through address)
  - Stack

- **Types of procedures:**
  - `NEAR`:
    - `IP` is saved on the stack; `CS` is not modified. The caller and called procedures must be part of the same `CS`.
  - `FAR`:
    - `CS` and `IP` are saved on the stack. Procedures must belong to different `CS` segments.

---

16. Macro definitions

Macro definitions are pseudo-operations. It allows the repeated inclusion of a code snippet in the main program.

- The assembler expands the macro, replacing it with the macro contents

```
name
MACRO {comma separated list of macro parameter names}
    ... ; macro body, good practice to save to the stack all registers
that
        ; the
        ; macro tampers with and to restore it at the end to be safe for
        ; calling
        ; from anywhere
ENDM
```

Usage: `macro_name {parameters}`

---

17. Macro and procedure libraries

Macro library	Procedure Library
Macros grouped into another file creates a macro library	Procedures grouped into another file creates a procedure library
Contains ***UNASSEMBLED*** code!!	Contains ***ASSEMBLED*** code
Use `INCLUDE {filename}` in the main program	Procedures in external procedure libraries must be declared **PUBLIC** for use in other modules and should be redeclared when used with *EXTRN*

---

18. Single and multiple segment programs

- ***only*** *6* segments (4 for old architectures) can be accessed at a time

.COM programs

- contains ***ONLY ONE SEGMENT*** ==> code and data only have 64kB space
- In conclusion, all references are made in a relative manner from the beginning

An example .COM source file

```
CODE SEGMENT PARA PUBLIC 'CODE'
    ASSUME CS:CODE, DS:CODE
    ORG 100H ; reserve 100 bytes for the DOS program segment prefix (PSP)

START: JMP ENTRY
    VEC     DB      1, 2, 3, 4
    LEN     DB      $-VEC
    SUM     DB      0

ENTRY:
    ; program starts here


    ; end program
```

```
    MOV AX, 4C00H ; 4C is the return function and AL contains the return
code
    INT 21H

CODE ENDS

END START
```

.EXE programs

- may have **several** segments
- for correct execution the corresponding ***segments (DS, ES, SS, CS) must be initialized***
- .EXE programs should be conceived such that the main procedure is of *FAR* type to return to definitions

It is very important to include the following code snippet at the beginning of every program

```
PUSH DS ; DOS automatically loads DS with the address of the PSP
XOR AX, AX ; clear AX
MOV DS, AX ; good to have an intermediate stable zero state, not necessary
PUSH AX ; creates the DS:AX return address for DOS when the program exits
        ; this is way the CS:IP pair gets loaded with the start of the PSP
when
        ; the program returns from the main function
        ; everything up until this point is absolutely necessary for
correct
        ; function of the program
MOV AX, DATA
MOV DS, AX ; ; set up the data segment, not absolutely necessary to do it
here
             ; but it is strongly recommended
```

---

19. Single and multiple module programs

Procedures may be defined FAR or NEAR

	NEAR	**FAR**
Stack operations	pushes IP	pushes CS and IP
Return operations	IP <== SP; SP+2;	IP <== [SP]; SP += 2; CS <== [SP]; SP += 2
Caveats	The invoked procedure must be in the same named code segment	Use only when near is impossible, because FAR calls are slower

---

20. Protected mode operation of I-80x86 processor

***real* mode**	**protected mode**
Physical addresses	Linear addresses
1 privilege level	4 privilege levels
Segment registers	Segment descriptors
Single user, single task	Multi user, multi task

Protected mode

- Space for physical address is $16\mathrm{MB}$ (real mode:$1\mathrm{MB}$) ==> divided into segments
- Segments
  - May have variable length up to $64\mathrm{kB}$
  - Can start from any address (not just paragraph address)
- Offers protection methods between programs running at the same time;

Advantages

- larger memory addressing
- multitasking (each user has its own private, virtual protected memory)

- there are **priority levels** assigned to tasks (from 0 to 3)
  kernel level has the most priority
- **access levels** also provide **access control**
- only ring 0 has access to I/O, HLT (halt)
  For addressing the whole space
  Segment register dimensions remain unmodified
  The memory contains selectors and tables contain descriptors

---

21. Memory management

This chapter is really important and almost always appears on the exam.

Provides the OS with powerful capabilities such as **segmentation** and **paging**.
Segmentation is ***mandatory*** whilst paging is optional.

Paging

- Mechanism to implement virtual memory
- Used by all operating systems and uses a bit of disk storage and some RAM
  - Simulates a large linear address space
  - Restrict access to specific portions of the memory
  - Can be enabled using **CR0 register**. Additionally,
    Page Directory Entries (PDE) and **Page Table Entries** (PTE) must be set up.
- Converts virtual addresses into physical addresses

Modes

	Physical address space	Segment length
Real mode	$1\,\mathrm{MB}$	$64\,\mathrm{kB}$
Protected mode	$16\,\mathrm{MB}$	$64\,\mathrm{kB}$
Virtual mode	$1\,\mathrm{GB}$	$64\,\mathrm{kB}$

---

22. Main data structures used in protected mode operation

In **protected** mode, segment registers are indices into special tables maintained by the operating system, but interpreted by the CPU

Program-generated addresses go through several stages to become a physical address:

- **Logical Address (Far Pointer):** This is the address used by a program, composed of a **Segment Selector** (an index into a descriptor table) and an **Offset** (position within the segment).
- **Linear Address:** The segmentation unit translates the logical address into a linear address. If paging is disabled, this is the physical address. If paging is enabled, it's further processed.
- **Physical Address:** The final, actual address in physical RAM. The paging unit, using structures like the **Page Directory** and **Page Table**, translates the linear address into the physical address, which the memory controller uses.

Key Data Structures for Memory Management

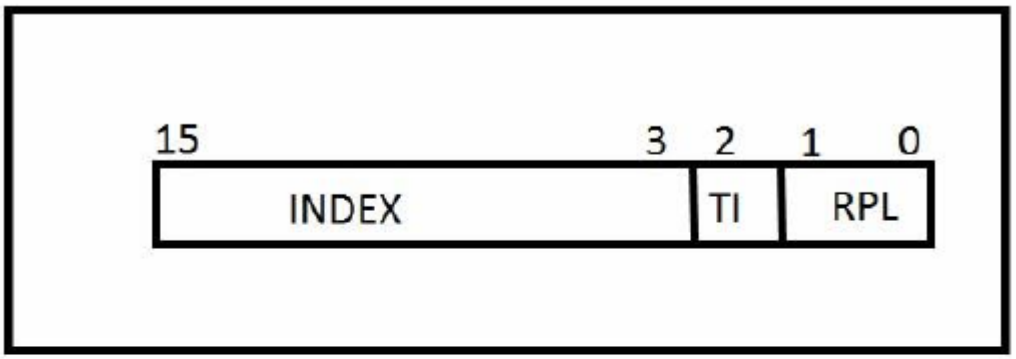
In protected mode, the processor uses several important data structures:

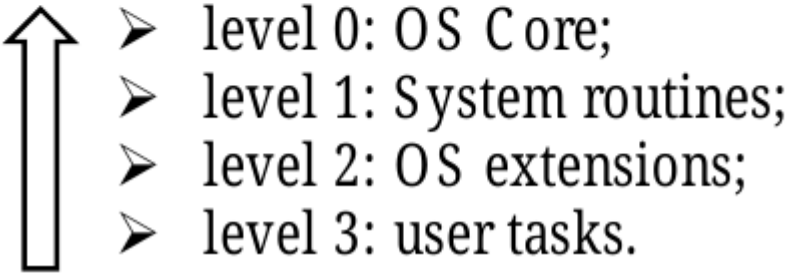
- **Global Descriptor Table (GDT):** Contains **segment descriptors** for all running programs. A segment descriptor holds information like a segment's base address, length (limit), and access rights. Unique, always accessible
- **Local Descriptor Table (LDT):** Similar to the GDT, but specific to individual tasks or programs, providing memory isolation. Usually one per task.
  There may be many present in the system, but only one is active.
- **Interrupt Descriptor Table (IDT):** Used to handle interrupts and exceptions by pointing to their respective handler routines.

Processor Operating Modes and Memory

The i-80x86 processor has different operating modes that affect memory addressing:

- **Real Mode:** The initial mode, compatible with the 8086. It has a 1 MB address space with 64 KB segments.
- **Protected Mode:** Offers advanced features like memory protection, multitasking, and a much larger address space (e.g., 16 MB, but modern processors support much more). It fully utilizes segmentation and paging.
- **Virtual Mode:** Allows real-mode programs to run within a protected-mode environment, each with its own virtual 1 MB address space.





- A selector contains
  - 13 bit index
  - 1 bit table identification `GDT (TI = 0)` or `LDT (TI = 1)`

Segment descriptors

- Contains segment information like base address, length and access rights

Code segments

- is read-only, loadable and executable

Bit	Name	Purpose
G	Granularity bit	`G=0` - segments range from 1 byte to 1 MB `G=1` Segments range from 4kB to 4 GB
U	User bit	OS defined
X	Reserved	Reserved by intel

---

23. System function calls

A collection of routines provided by the OS to achieve high level functions.
There are two types

BIOS functions

- Allows hardware access
- Saves the following registers: `CS, SS, DS, ES, BX, CX, DX`. The others are ***destroyed***.

Most common system functions:

Interrupt (INT)	Description
INT 10h	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	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.

DOS functions

Written for OS specific purposes, mainly for handling **files**, **date** and **time**, etc.
It is **always** called using `INT 21h`.

A table with the most commonly used DOS interrupts, it is good to know at least the string reading and printing functions:

Interrupt (INT)	Function (AH Register Value)	Description	Typical Usage (Registers)
INT 20h	**00h**	**Terminate Program (Obsolete):**he current program and returns control to DOS.	No specific registers required before call.
INT 21h	**00h** (Program Terminate)	**Terminate Program:** Similar to INT 20h, but often preferred for more controlled exits.	No specific registers.
INT 21h	**01h** (Read Character from STDIN)	**Read Character from Standard Input:** Reads a single character from the keyboard and echoes it to the screen. Returns the character in AL.	No input registers. Output: AL = character.
INT 21h	**02h** (Write Character to STDOUT)	**Write Character to Standard Output:** Displays a character on the screen.	Input: DL = character to display.
INT 21h	**09h** (Write String to STDOUT)	**Write String to Standard Output:** Displays a string on the screen. The string must be terminated with a '$' character.	Input: DS:DX = pointer to string.
INT 21h	**35h** (Get Interrupt Vector)	**Get Interrupt Vector:** Returns the address of a specified interrupt handler.	Input: AL = interrupt number. Output: ES:BX = segment:offset of handler.
INT 21h	**3Ch** (Create File)	**Create File:** Creates a new file or truncates an existing one.	Input: CX = file attributes, DS:DX = pointer to ASCIIZ filename. Output: AX = file

Interrupt (INT)	Function (AH Register Value)	Description	Typical Usage (Registers)
			handle (success), CF set, AX = error code (failure).
INT 21h	**3Dh** (Open File)	**Open File:** Opens an existing file.	Input: AL = access mode (0=read, 1=write, 2=read/write), DS:DX = pointer to ASCIIZ filename. Output: AX = file handle (success), CF set, AX = error code (failure).
INT 21h	**3Eh** (Close File)	**Close File:** Closes an open file handle.	Input: BX = file handle. Output: CF set, AX = error code (failure).
INT 21h	**3Fh** (Read from File/Device)	**Read from File or Device:** Reads bytes from a file or device.	Input: BX = file handle, CX = number of bytes to read, DS:DX = pointer to buffer. Output: AX = number of bytes read (success), CF set, AX = error code (failure).
INT 21h	**40h** (Write to File/Device)	**Write to File or Device:** Writes bytes to a file or device.	Input: BX = file handle, CX = number of bytes to write, DS:DX = pointer to buffer. Output: AX = number of bytes written (success), CF set, AX = error code (failure).
INT 21h	**4Ch** (Terminate Process with Return Code)	**Terminate Process with Return Code:** Exits the current program and returns an exit code to DOS. This is the most modern and recommended way to exit a DOS program.	Input: AL = exit code.
INT 21h	**4Eh** (Find First File)	**Find First File:** Searches for the first file matching a specified file mask.	Input: CX = file attributes, DS:DX = pointer to ASCIIZ filename mask. Output: CF set, AX = error code (failure); if success, DX points to DTA (Disk Transfer Area) with file info.

Interrupt (INT)	Function (AH Register Value)	Description	Typical Usage (Registers)
INT 21h	**4Fh** (Find Next File)	**Find Next File:** Continues a file search initiated by INT 21h/4Eh.	No input registers. Output: CF set, AX = error code (failure); if success, DX points to DTA (Disk Transfer Area) with file info.

---

24. Math co-processor structure and operation

This is also an important chapter. The laboratories don't mention the FPUs but you absolutely have to write code using the x87 floating point co-processor at the exam or at least know the internal structure of it.

The math co-processor provides arithmetic capabilities to the processor and shares the ***data*** , ***address*** and **control buses** of the main processor.
The math co-processor

- Requires a special **ESC** (escape) sequence
- Operates in parallel to the main processor
- May overtake the **BUS** for significant amounts of time if additional data is needed
- Uses registers for addressing
- ***Lacks immediate mode***
- ***Needs special synchronization signals to cooperate***
- Instructions need ***tens to hundreds*** of clock cycles to finish. It's slow!

These are the following conditions to execute floating point operations in the co-processor:

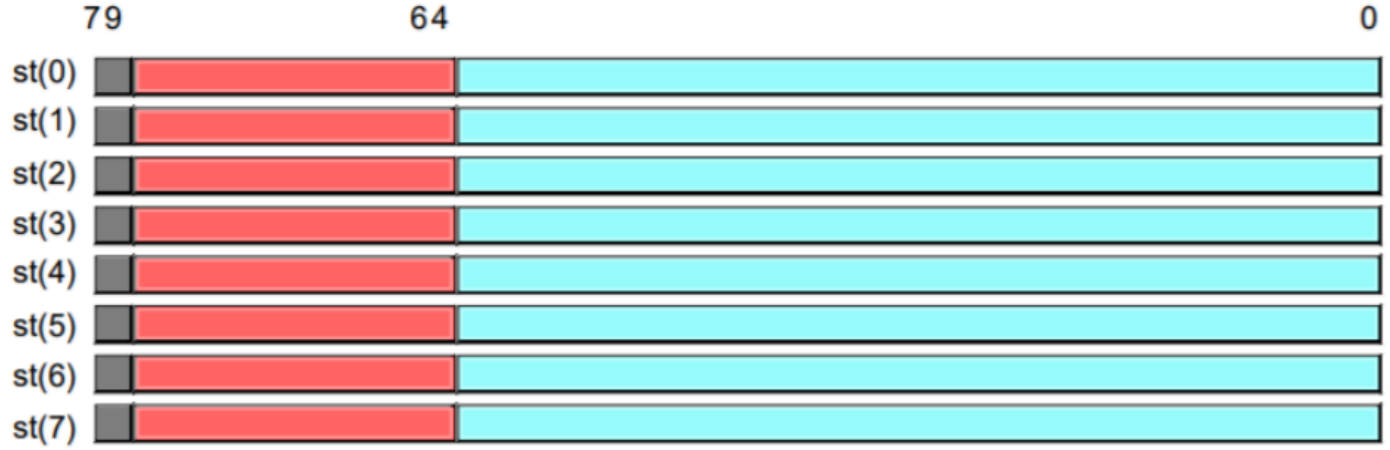
- At least one operand **has** to be at the top of the stack

The co-processor endows the processor with the following capabilities

- Load data into the co-processors internal stack from *RAM*
- Execute arithmetic operation
- Store the results in *RAM*

1. Register stack

The stack is structured in the following manner



It has 8 by 80 bit data registers organised as a stack and are referred to as $St(0)$, $St(1)$, ..., $St(7)$.

2. Control register

It is a *16-bit* register. The first 8 bits controls *how* arithmetic is handled and the other 8 bits control what happens when errors occur.

Bit	Purpose
IC	Infinite control
RC	Rounding control
PC	Precision control
Mask	Validates or invalidates interrupt requests
Others	PM, UM, OM, etc. control which error actually calls the interrupt

3. Status register

16-bit register. Provides information about error states and what interrupt generated it.

4. Tag register

Contains '*tags*' that describe the data in each register on stack

Value	Bit meaning
00	Valid data

Value	Bit meaning
01	Is zero
10	Contains special value
11	Is empty

5. Instruction pointer

Contains co-processors last op-code. Helps in debugging.

6. Data pointer

Contains the physical address of the last external data utilized in the last floating point operation.

---

25. Math co-processor data types

The co-processor can handle the following data types

- 2's complement integers
- PBCD
- IEEE 754/854 format floating point (no mention about short and long format)

These following representations are **only** in memory. Internally, **all** numbers are represented on 80 bits.

Integer types

Type	Size	Representation
DW	16-bit	C2
DD	32-bit	C2
DQ	64-bit	C2
PDDT[1]	80-bit	Numbers up to 18 decimal digits can be represented

Real types

Type	Size	Representation
Short	32-bit	Single-precision float
Long	64-bit	Double-precision float
Extended	80-bit	Extended-precision[2]

---

26. Math co-processor data transfer and constant load instructions

The FPU either loads **from memory** or using a **constant loading instruction.**
It can not load from registers or immediate values, because it does not have access to the general purpose registers of the processor. It only shares the buses that access memory.

Load Instructions

Instruction	Description
FILD adr	Loads an **integer** variable from `adr` onto the FPU stack. Converts it to the FPU's internal format.
FLD adr	Loads a **real (floating-point)** variable (single or double precision) from `adr` onto the FPU stack. Converts to internal format.
FBLD adr	Loads a **packed decimal** variable from `adr` onto the FPU stack. Converts to internal format.

Store Instructions

Instruction	Description
FIST adr	Stores the value from $St(0)$ (top of stack) as an **integer** to `adr`. The stack pointer remains unchanged. Conversion occurs during storage.

Instruction	Description
FISTP adr	Stores the value from $St(0)$ as an **integer** to `adr`. $St(0)$ is then popped, and the stack pointer is decremented. Conversion occurs during storage.
FST adr	Stores the value from $St(0)$ as a **short integer or double-precision real** to `adr`. The stack pointer and stack data remain unchanged. Conversion occurs during storage.
FSTP adr	Stores the value from $St(0)$ as a **floating-point number** (single, double, or extended precision) to `adr`. $St(0)$ is then popped, and the stack pointer is decremented. Conversion occurs during storage.
FBSTP adr	Stores the value from $St(0)$ as a **packed decimal number** (DT) to `adr`. $St(0)$ is then popped, and the stack pointer is decremented. Conversion occurs during storage.

Internal Data Transfer Instructions

Instruction	Description
FLD $St(i)$	Pushes a copy of the value from $St(i)$ onto the top of the stack ($St(0)$).
FST $St(i)$	Copies the value from $St(0)$ into $St(i)$, overwriting its previous content. The stack is unchanged.
FSTP $St(i)$	Copies the value from $St(0)$ into $St(i)$, overwriting its previous content. $St(0)$ is then popped.
FXCH $St(i)$	Swaps the values between $St(0)$ and $St(i)$.

Constants Loading Instructions

Instruction	Description
FLDZ	Loads `0.0` onto the top of the stack.
FLD1	Loads `1.0` onto the top of the stack.

Instruction	Description
FLDPI	Loads π onto the top of the stack.
FLDL2T	Loads $\log_2(10)$ onto the top of the stack.
FLDL2E	Loads $\log_2(e)$ onto the top of the stack.
FLDLG2	Loads $\log_{10}(2)$ onto the top of the stack.
FLDLN2	Loads $\ln(2)$ onto the top of the stack.

---

27. Math co-processor arithmetic and control instructions

Addition Instructions

Instruction	Description
FADD	Adds $St(1)$ to $St(0)$. Result in $St(0)$: $St(0)$ <= $St(0)$ + $St(1)$.
FADD op (FP)	Adds `op` (from memory or stack) to $St(0)$ as a floating-point operation. Result in $St(0)$.
FADD op (Integer)	Adds `op` (from memory or stack) to $St(0)$ as an integer operation. Result in $St(0)$.
FADD $St(i)$, $St(0)$	Adds $St(0)$ to $St(i)$. Result in $St(i)$: $St(i)$ <= $St(i)$ + $St(0)$. $St(0)$ is then popped.

Subtraction Instructions

Instruction	Description
FSUB	Subtracts $St(1)$ from $St(0)$. Result in $St(0)$: $St(0)$ <= $St(0)$ - $St(1)$.
FSUB op (FP)	Subtracts `op` (from memory or stack) from $St(0)$ as a floating-point operation. Result in $St(0)$.

Instruction	Description
FSUB op (Integer)	Subtracts `op` (from memory or stack) from $St(0)$ as an integer operation. Result in $St(0)$.
FSUB $St(i)$**,** $St(0)$	Subtracts $St(0)$ from $St(i)$. Result in $St(i)$: $St(i)$ <= $St(i)$ - $St(0)$. $St(0)$ is then popped.
FSUBR $St(i)$	Subtracts $St(i)$ from $St(0)$. Result in $St(i)$: $St(i)$ <= $St(0)$ - $St(i)$. (Reverse subtract)

Multiplication Instructions

Instruction	Description
FMUL	Multiplies $St(0)$ by $St(1)$. Result in $St(0)$: $St(0)$ <= $St(0)$ * $St(1)$.
FMUL op (FP)	Multiplies $St(0)$ by `op` (from memory or stack) as a floating-point operation. Result in $St(0)$.
FMUL op (Integer)	Multiplies $St(0)$ by `op` (from memory or stack) as an integer operation. Result in $St(0)$.
FMULP $St(i)$**,** $St(0)$	Multiplies $St(i)$ by $St(0)$. Result in $St(i)$: $St(i)$ <= $St(i)$ * $St(0)$. $St(0)$ is then popped.

Division Instructions

Instruction	Description
FDIV	Divides $St(0)$ by $St(1)$. Result in $St(0)$: $St(0)$ <= $St(0)$ / $St(1)$.
FDIV op (FP)	Divides $St(0)$ by `op` (from memory or stack) as a floating-point operation. Result in $St(0)$.
FDIV op (Integer)	Divides $St(0)$ by `op` (from memory or stack) as an integer operation. Result in $St(0)$.

Instruction	Description
FDIVP $St(i)$, $St(0)$	Divides $St(i)$ by $St(0)$. Result in $St(i)$: $St(i)$ <= $St(i)$ / $St(0)$. $St(0)$ is then popped.
FDIVR $St(i)$	Divides $St(0)$ by $St(i)$. Result in $St(i)$: $St(i)$ <= $St(0)$ / $St(i)$. (Reverse divide)

Command (Control and State Management) Instructions

Usually, they have no *direct* arithmetic meaning, but some of them do influence drastically the actions of the co-processor, because they control the state of the co-processor.

Instruction	Description
FINIT	Initializes the FPU to its default state, performing a 'software reset'.
FENI	Enables FPU interrupts.
FDISI	Disables all FPU interrupts, regardless of command register settings.
FLDCW adr	Loads the FPU Control Word (command register) from the memory address `adr`.
FSTCW adr	Saves the FPU Control Word (command register) to the word located at memory address `adr`.
FSTSW adr	Saves the FPU Status Word (status register) to the word located at memory address `adr`.
FCLEX	Clears (erases) the exception flags within the FPU Status Word.
FSTENV adr	Saves the FPU environment (control word, status word, tag word, instruction pointer, data pointer) to a 14-byte memory location starting at `adr`.
FLDENV adr	Loads the FPU environment from a 14-byte memory location starting at `adr`.
FSAVE adr	Saves the complete FPU state, including all internal registers and the entire FPU stack, to a 94-byte memory location starting at `adr`.
FRSTOR adr	Restores the complete FPU state (registers and stack) from a 94-byte memory location starting at `adr`.

Instruction	Description
FINCSTP	Increments the FPU stack pointer by 1.
FDECSTP	Decrements the FPU stack pointer by 1.
FFREE $St(i)$	Marks the i-th element in the FPU stack as empty (free). This operation does not change the stack pointer.
FNOP	Performs no operation. It's essentially a placeholder instruction.
FWAIT	Waits for any pending FPU operation to complete before the CPU proceeds. (Similar to the 8086's WAIT instruction).

---

28. Math co-processor mathematic functions

Mathematical Function Instructions

Instruction	Description
FSQRT	Calculates the square root of $St(0)$. The result replaces the value in $St(0)$: $St(0)$ <= $\sqrt{ST(0)}$.
FSCALE	Multiplies $St(0)$ by a power of 2. The result in $St(0)$ is $St(0)$ * $2^{ST(1)}$.
FPREM	Calculates a partial remainder. Divides $St(0)$ by $St(1)$, and the remainder is stored back into $St(0)$.
FRNDINT	Rounds the value in $St(0)$ to the nearest integer, according to the current rounding mode set in the FPU Control Word. The rounded value replaces $St(0)$.
FXTRACT	Extracts the exponent and mantissa from the value in $St(0)$. The exponent is placed in $St(0)$, and the mantissa (significand) is placed in $St(1)$.
FABS	Replaces the value in $St(0)$ with its absolute value.
FCHS	Changes the sign of the value in $St(0)$.
FPTAN	Calculates the partial tangent of the angle in $St(0)$. The result is $\frac{ST(1)}{ST(0)}$, where the original $St(0)$ is the angle in radians (must be between 0 and $\pi/4$). After the operation, $St(0)$ contains the ratio and $St(1)$ is popped or set up for the division.

Instruction	Description
FPATAN	Calculates the partial arctangent. The arctangent of the ratio $\frac{ST(1)}{ST(0)}$ is computed and the result is stored in $St(0)$. The initial value in $St(0)$ must be positive, and $St(1)$ must be greater than $St(0)$.
F2XM1	Calculates $2^{ST(0)} - 1$. The result replaces the value in $St(0)$.
FYL2X	Calculates $ST(1) \times \log_2(ST(0))$. The result replaces the value in $St(0)$.
FYL2XP1	Calculates $ST(1) \times \log_2(ST(0) + 1)$. The result replaces the value in $St(0)$. $St(0)$ must be a positive number lower then 0.3, while $St(1)$ has to be a finite number.

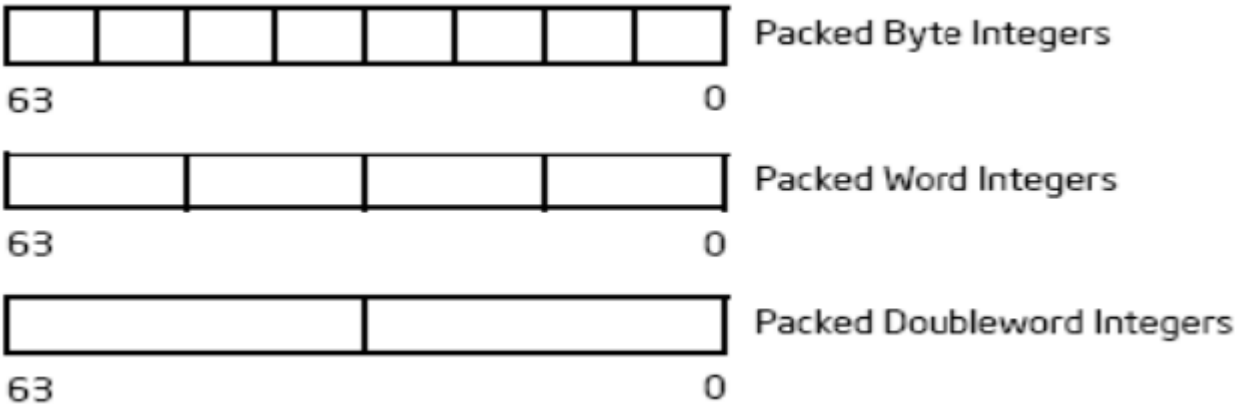
---

29. MMX extensions

This chapter should also not be overlooked. The structure and working principle appears on the exam quite frequently.

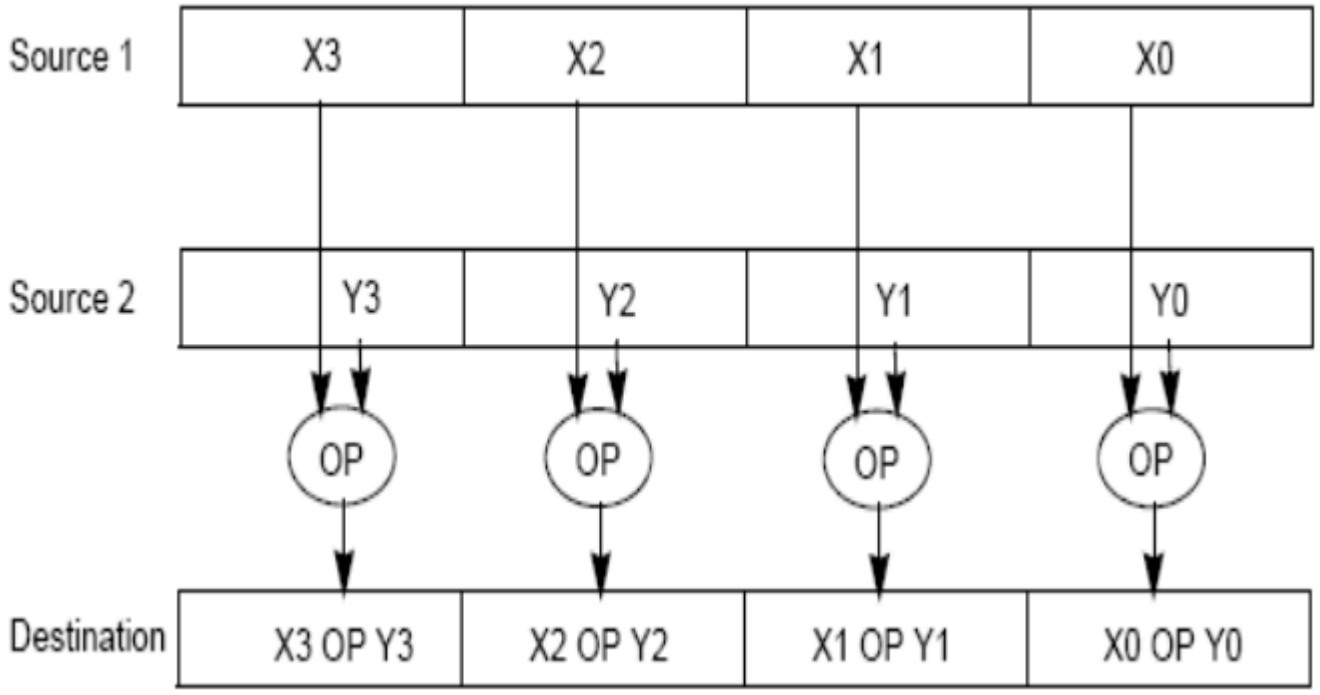
Overview

- **8 new 64-bit registers, called MMX registers**
- 3 new packed data types:
  - 64-bit packed integers (signed/unsigned)
  - 64-bit packed word integers (signed/unsigned)
  - 64-bit packed double word integers (signed and unsigned)



SIMD Execution Model

SIMD stands for **S**ingle **I**nstruction **M**ultiple **D**ata. MMX Instructions move 64-bit ***packed*** data types between MMX registers.
The MMX unit performs the same arithmetic operations we are used to but more of them in parallel. The quantity of the parallel operations depend on the size of the operands and *how many* can be ***packed*** into a 64-bit MMX register.



---

30. Multimedia calculus and main instruction families

New arithmetic is introduced

- Wraparound arithmetic - a true out of range result is truncated and overflow bits are ignored
- (Un)signed saturation arithmetic - out-of-range results are limited to the representable range of (un)signed integers operated on

MMX instructions

A set consisting of 47 instructions grouped into the following categories:

- data transfer
- arithmetic
- comparison
- conversion

- unpacking
- logical
- shift
- EMMS (empty MMX state instructions)

---

31. Program Optimization, General Issues, Optimization Levels

This is also a really important chapter as it gets you easy points on the exam and appears ⅔ of the time.

- The optimization process is not cheap; you may have to redesign and rewrite major portions of the program to get acceptable performance.
- Two main approaches:
  - **Optimize late:**
    - **"90% of a program's execution time is spent in 10% of the code."**
    - Write the program normally, then find and optimize the heaviest part.
    - The slow 10% may be spread throughout your program, making it difficult to optimize.
  - **Optimize early:**
    - No need to search through your code because you optimize in-place from the beginning.
- **Optimization levels:**
  - **High level:** Choose a **better algorithm**.
    - Often difficult.
    - Always the first step you should try, as it could reap the greatest benefits.
  - **Medium level:** Implementing the algorithm better.
    - The most powerful possibility in assembly, can be machine-specific.
    - Steps include:
      - **Unrolling loops.**
      - Using **table lookups** rather than computations.
      - Eliminating computations from a loop whose value doesn't change within the loop.
      - Using `INC` and `DEC` and shifts instead of `MUL` and `DIV` if possible.
      - Trying to keep variables in registers as long as possible.
  - **Low level:** Counting clock cycles.
    - Tedious and requires a very careful analysis of the program and the system.

- Ensures that an instruction sequence uses **as few clock cycles as possible**.
- Difficult, requiring knowledge of each instruction's clock time.
- Pay attention to:
  - Instruction size.
  - Addressing mode.
  - Cache handling.

---

32. Non-Optimal Code Analyzation Methods

- **Trial & error:**
  - Make an educated guess about where the program is spending most of its time.
  - If the guess is right, the program will run much faster after optimization.
  - Optimizing a part that doesn't need much optimization is ineffective.
- **Optimize everything:**
  - Can be used for short sections.
  - No need for analysis.
  - By optimizing everything, you are sure to optimize the slow code.
- **Analyze the program:**
  - Study your code and determine where it will spend most of its time based on the data you will process.
  - In theory, it is the best technique and should always work; however, practically, it can be slow and difficult.
  - Try to locate all recursive function calls and loop bodies.
- Use a **profiler** program:
  - The profiler measures how long the code takes to execute specific portions.
  - Works by interrupting the code periodically.
  - Returns a **histogram** of interrupts for timing analysis.

---

33. Optimization Techniques in Assembly Language Programming (ALP)

- **Unrolling loops**: Do the work of multiple iterations instead of using the costly `LOOP` instruction.
- Using **table lookups** rather than computations.

- Eliminating computations from a loop whose value does not change within the loop, for example, with `XLAT`.
- Using `INC` and `DEC` instead of `ADD 1` or `SUB 1`.
- Using shifting instead of multiplication where possible.
- Keeping values in registers for as long as possible.
- Use `XOR` to zero a register instead of `MOV reg, 0`.

Example FPU code

This code snippet calculates the area of a circle and stores it to short and long floats and demonstrates proper FPU management.

```
DATA SEGMENT PARA PUBLIC 'DATA'
    RADIUS DD 5.0 ; decimal point is necessary for the assembler to
                  ; recognize that it is a floating point number
    AREA_LONG_FLOAT DQ ? ; leave uninitialized
    ; OR
    AREA_SHORT_FLOAT DD ?
DATA ENDS


CODE SEGMENT PARA PUBLIC 'CODE'
ASSUME CS:CODE, DS:DATA

MAIN PROC FAR

; Set up return to dos
PUSH DS
XOR AX, AX
PUSH AX
MOV AX, DATA
MOV DS, AX

; CODE START
FINIT
FLD RADIUS
FLD RADIUS
FMULP St(1)
FLDPI
FMULP St(1)
FST AREA_LONG_FLOAT
FSTP AREA_SHORT_FLOAT
```

```
RET
; CODE ENDS
MAIN ENDP

CODE ENDS

END MAIN
```

---

1. Packed decimal DT ↩
2. Co-processor extended 80-bit format ↩