ASSEMBLY THEORY:

I.

1. Machine code = is any low-level programming language, consisting of machine language intructions, which are used to control a computer’s central processing unit(CPU). Each instruction causes the CPU to perform a very specific task, such as a load, a store, a jump or an arithmetic logic unit(ALU) operation on one or more units of data in the CPU’s registers or memory.

Computers “speak” in binary because of how they are built. A computer is a vast collection of switches, because of a physical limitation of how computers work:

1 == TRUE/ON

0 == FALSE/OFF

* The bits of 0 and 1 are transmited as semnals of 1 or 0 for microtransistors.

BASE 16/HEXA -is a zipped version of base 2

Base 16 is just an interpretation for better understanding and efficiency, because the computers do not use hexa

BIT = Basic unit for representing information inside a computer

The memory of a computer is constructed of bits, but it is organized in bigger elements called bytes

The data segment is separate because what is written in high level programming language is sent to a data segment. The data segment is not executable, it is “DEAD”

Addresses are naming memory locations when creating a variable. An address points to a location where data may be accessed.

Byte = smallest unit of information that can be accesed and addressed by a microprocessor

1. COMPONENTS:

* ALU = Arithmetic and logic unit
  + ALU is a component of the CPU. It can apply the instruction code and activate a circuit and utilizes an already fixed program.

OBS: The native architecture doesn’t work with real numbers, only integers

The task to use real numbers is made by the mathematical co-processor

The ALU component works only with addition, substraction, multiplication and division. It works only with addressed, no values or commands.

BACKWORDS COMPATIBILITY – based on what was created before

For not being in danger of mixing everything, any kind of information flows on different channels, like highways. An insruction goes on a command bus. Gor being able to access the operators and operands we need the BUS INTERFACE UNIT(BIU)

* EU – EXECUTIVE UNIT
  + Runs the machine instructions by means of ALU
* BUS INTERFACE UNIT(BIU):
  + Is responsible for grabbing from memory the operands. It needs addresses to work.
  + It prepares the execution of every machine instruction. Reads an instruction from memory, decodes it and computes the memory address of an operand, if any. The output configuration is stored in a 15 bytes buffer, from where EU will take it

BUFFER = Region of memory used to temporarily hold data while is it being moved from one place to another

OBS: EU and BIU work in parallel – while EU runs the current instruction, BIU prepares the next one. These two actions are synchronized – the one that ends first waits after the other.

* REGISTERS
  + These are the storage capacities. These are very small in terms of size(8, 16, 32, 64, 128 bits)
  + Registers are fast in terms of access speed. Registers are used for temporary store(commands, codes, data, addresses)

DATA TYPE = meaning of trying to interpret something. From a constructive and logical point of view, data type is a couple of structure + associated operations(specific to that type)

Interface = variables + functions

Implementation = variables + functions that are private

RAM – Random acces memory

* Volatile memory(needs constant power in order to retain data)
* The acces time to one memory area is the same independently of the location of that memory area (the memory area can be randomly far from the beginning of the memory)
* RAM accepts read & write in any order(random how many r&w and random order)

II. EXECUTIVE UNIT – Registers, E- flags, ALU

1. There are 8 general registers: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI. They are called general registers because the programmer has access to them and he can use them in whatever way he wants. On the other hand, every one of them was introduced on 16 bits having in mind a specific role and functionality
2. EAX = ACCUMULATION REGISTER
   1. Is the most used register in any architecture
   2. It is usually used as one of the operands
   3. Is used for multiplication and division
3. EBX = BASE REGISTER
   1. Abase can be the beginning of an array
4. ECX = COUNTER REGISTER
   1. This register is mostly used as a numerical upper limit for instructions that need repetitive runs
   2. It is a so called “iteration variable”(NOT A VARIABLE!)
   3. If cx = 0 => the repetitive instruction finish
5. EDX = DATA REGISTER
   1. Frequently used with EAX, when the result exceeds a dword

There is no direct way to access the HIGHER PART of these registers!

OBS: “WORD SIZE” refers to the number of bits processed by a computer’s CPU in one go(these days typically 32 bits or 64 bits). Data bus size, instruction size, address size are usually multiples of the word size. So, for the CPU, the “word size” is a basic attribute/feature influencing the above mentioned elements.

* + ESP and EBP – are stack registers. The stack is a LIFO area

1. ESP - STACK POINTER:
   1. It points to the last element put on the stack(the element from the top of the stack)
2. EBP – BASE POINTER:
   1. It points to the first element put on the stack(points to the stack’s basis)
   * EDI and ESI are index registers usually used for accessing elements from bytes and words strings. Their frunctioning in this context(DESTINATION INDEX and SOURCE INDEX) will be later clarified

OBS: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI are doubleword registers (32 bits). Every one of them may also be seen as the concatenation of two 16 bits subregisters. The upper register, which contains the most significant 16 bitsof the 32 bits register, doesn’t have a name and it isn’t available separately. But the lower registers could be used as single so we have the 16 bits registers AX, BX, CX, DX, DP, BP, DI, SI. Among these registers, AX, BX, CX, DX are also a concatenation of two 8 bits subregisters. So, we have AH, BH, CH, DH registers which contain the most significant 8 bits of the word (the upper part of AX, BX, CX and DX registers) and AL, BL, CL, DL registers which contain the least significant 8 bits of the word(the lower part)

An execution takes place in the following order:

* Usually there exists a main program
* When the main program starts, the microprocessor associates to the main programming unit the main program stack frame/activation record(all of this is done by the compiler)
* This takes place until a new function appears. The execution ends here, it is all saved and a new stack frame is created. Therefore, the old stack frame creates a the new one that is now the main stack frame
* Everything will be called from the top to the bottom

A POINTER VARIABLE is a special variable in the surse that is used to store an address of another variable. To differentiate it from other variables that do not store an address, we use \* symbol in declaration(differencing operator) – sticla de vin moment

The name of an array in C = its starting address

The name of a variable in assembly is the address of the value

The reference to a certain element is made through syntactics.

[] – operator helps(2am formula)

FLAGS:

A FLAG IS AN INDICATOR REPRESENTED ON 1 BIT.

A configuration of the flags registers shows a synthetic overview of the execution of the each instruction(uitandu-ne la FLAGS register vedem efectele instructiunilor)

For x86 the EFLAGS register(the status register) has 32 bits, but only 9 are actually used.

EFLAGS REGISTER:

1. CF – carry flag = the transport flag. It will be set to 1 if in the LPO there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise.

We are in binary logic. Because everything is binary – any instruction has max 2 operands. Binay numbers – only 0 and 1. Since 1 is a bit, it doesn’t need to sacrifice womething bigger(byte, word, etc), so for marking that there is an extra 1 – it will use a flag

Why no operations with 3 operands?? – Because the possibility of having a transport digit becomes 2 – not binary

!!! CF flags the UNSIGNED overflow!!

1. PF – parity flag – Its value is set so that together with the bits 1 from the least significant byte of the representation of the LPO’s result an odd number of 1 digits to be obtained
2. AF – auxiliary flag – shows that transport value from bit 3 to bit 4 of the LPO’s result

AF is set to 1, if for example, during an add operation there is a carry from the low to the high nibble, or a borrow from the high nibble to the low nibble(subtraction case)

* + It works like a CF, but reporting to a nibble(half of a byte)

1. ZF – zero flag – is set to 1 if the result of the LPO is strictly 0 and set to 0 otherwise
2. SF- sign flag – is set to 1 if the result of the LPO is strictly negative and set to 0 otherwise

!!WHAT MEANS TO REPRESENT NEGATIVE NUMBERS? It is need to reserve one bit for the sign, so only seven bits are used for the absolute value.

#Mathematicians invented a mechanism in order to keep the importance of the sign bit while it is still participating ar the construction of the value(2’s complement)

SF – marks the last bit

!!FOR ANY ARCHITECTURE AT COMPUTE SCIENCE, ZERO IS POSITIVE!!

Why? Because when interpreting zero as a signed number, using 2’s complement, the MSB is zero(positive) – the SF = 0

1. TF – trap flag – I a debugging flag. If set to 1 – the machine stops after every execution. If set to 0 – let everything flow
   * This is why programmers can use debuggers even step by step
2. IF – interrupt flag – If set to 1 interrupts are allowed. If set to 0 interrupts will not be handled
   * CRTICAL SECTION = section of code that cannot be interrupted.
   * When IF = 0, the code cannot be interrupted. 1 – opposite.
   * The time between the switch of if from 0 to 1 is called a TIME CRITICAL SECTION
3. DF – direction flag – for operating instructions
   * If set to 0 – string parsing will be performed in ascending order
   * If set to 1 – string parsing descending order
   * THIS IS NOT SET BY THE PROCESSOR
4. OF – overflow flag – flags the signed overflow
   * If the result of the LPO(interpreted in signed interpretation) didn’t fit the reserved space, then OF = 1, else OF = 0
   * Like CF, but in signed representation
   * Never able to have an overflow on multiplication(because the result will be 2\*operand\_size)

ADDRESS OF A MEMORY LOCATION = nr of consecutive bytes from the beginning of the RAM memory and the beginning of that memory location

An uninterrupted sequence of memory locations, used for similar purposes during a program execution, represents a SEGMENT. So, a segment represents a logical section of a program’s memory, featured by its basic address(beginning), by its linit(size) and by its type. Both basic address and segment size have 32 bits value representations.

FLAGS CATEGORIES:

* 1. Reporting the status of the LPO(having a previous effect):
     1. CF,PF,AF,ZF,SF,OF
     2. ADC(Conditional jumps JA = JNBE, JG = JNLE;JZ..)
  2. Flags to be set by the programmer(having a future effect) on the next flags: CF, TF, IF, DP

Using the special instructions:

CLC – effect: CF = 0

STC – effect: CF = 1

CMC – effect: complements of the value of CF

CLD – effect: DF = 0

STD – effect: DF = 1

CLI – effect: IF = 0

STI – effect: IF = 0

!!There are no instructions to directly access TF, because of the risk of accidentaly setting the value from TF and also because its absolutely special role is to develop debuggers.

ADDRESS REGISTERS AND ADDRESS COMPUTATION:

ADDRESS OF A MEMORY LOCATION = no. of consecutive bytes from the beginning of the RAM memory to the beginning of that memory location

In the family of 8086-based processors, the term segment has two meanings:

1. A block of memory pf discrete size, called a physical segment. The number of bytes in a physical memory segment is:
   1. 64K for 16-bit processors
   2. 4GIGABYTES for 32 bit processors
2. A variable sized block of memory, called a logical segment occupied by a program’s code or data

A segment represents a logical section of a program’s memory, featured by its basic address(beginning), by its limit(size) and by its type(code, data, stack)

* + Both basic address and segment’s size have 32 bits value representations.
    - When the people wanted to build a computer using intel with 1mb of RAM, the developers had to create a processing mechanism that could compute up with 20 bits addresses. How can you do that on 16 bits? It is needed a logic mechanism to use more than one register. Therefore, you need a 2-step mechanism. They created a memory segment. This segment was initiated for every address and after that, you can specify how far from the beginning of the segment you go. Any beginning of the memory segment must also be a 20 bits address, so they proposed that any segment will start only at a multiple of 16 bits. The last 4 binary digits are 0, so a 16 bit utility is enough to represent the most significant digits.
    - This is only an address specification. Who is counting the fnal address? It is used a formula – an address computation: only use the first 16 bits from the segment as a number, multiply it by 16 and then add the offset.
    - [SEGMENT\_CODE] \* 16 + OFFSET

ADDRESSING MECHANISM ON 32 BITS:

1 GB = 2^30

4 GB = 2 ^ 32??

If this design could be started the, the 2 step mechanism wouldn’t be needed because 32 bits would have been enough. From a logical point of view, a single structure would have been needed.

MEMORY FLAT MODEL = no memory segments with start – finish, only one long memory segment

\*In 8086 processors, the term segment has 2 meanings:

1. block of memory of discrete size = physical segment:

* + No of bytes in a physical memory segment:
    - 64K – 16 bit processors
    - 4 GB – 32 bit processors

2.Variable – sized block of memory = logical segment

* + Occupied by a program: code & data

OFFSET = the address of a location relative to the beginning of a segment, or, in other words, the number of bytes between the beginning of the segment and the particular memory location. An offset is valid only if his numerical value, on 32 bits, doesn’t exceeds the segment’s limit which is referring to!

OBS: The flat memory model means that now there are no longer starting addresses of memory segment. It is a very protective way of memory processing, because it is hidden by the operating system which is hiding all the segments that are possible to be handed to the programmer. So, the OS is managing a SELECTOR TABLE.

ADDRESS SPECIFICATION = pair of a segment selector and an offset

SEGMENT SELECTOR = numeric value of 16 bits that selects uniquely the accessed segment and his features

= it is defined and provided by the OS

In hexadecimal, an address specification can be written as:

S – selector

O – offset (base, limit) – has this obtained by the processor after the segmentation process

! < ! - condition to give access to a specific location

THE ACTUAL SEGMENTATION – address computation:

* Address specification = FAR address
* Spec. only by offset = NEAR address

To compute the linear address, the processor will:

1. Check if the segment with value 8 was defined by the OS and blocks the access if such a segment wasn’t defined ( memory violation error!!)
2. It extracts the base address(B) and the segment’s limit (L)
3. It verifies if the offset doesn’t exceed the limit. If it does, the access will be blocked
4. It adds up the offset to the base address. This computation = made by ADR component of BIU
   * Segmentation segmented addressing model

OBS: The x86 processor also have a memory access control mechanism called PAGING(independent of address segmentation)

* PAGING = dividing the virtual memory into pages which are associated(translated) to the physical memory( 1 page = bytes = 4096 bytes
* The configuration, control of segmentation and paging are performed by the OS. Only segmentation interferes with addresses
  + Paging = transparent relative to user programs

Protected mode on 32 bits – using paging and segmentation

X86 architecture allows 4 types of segments:

1. CS – Code segment
2. DS – Data Segment
3. SS – Stack Segment
4. ES – Extra Segment
   * Every program – composed by one or more segments. At any moment there is at most one active segment of any type (not synchron). Registers CS, DS, SS, ES from BIU contain the values of the selectors of the active segments corresponding by to every type. So CS, DS, SS, ES determine the starting addresses (unique base) and the dimensions(limit) of these four active segments.
   * EIP contains the offset of the current instruction inside the current code segment(like an index through the current segment) – register managed by BIU

OBS: A x86 machine instruction represents a sequence of 1 to 15 bytes: these values specifying an operation to be run, the operands to which it will be applied and also possible suplimentary modifiers.

A x86 machine instruction has max. 2 operands. For most instructions, they are called SOURCE and DESTINATION

From these two operands, only one can be stored in the RAM memory. The other one = EU register/constant

INSTRUCTION NAME DESTINATION, SOURCE = general form of an instruction

* The interval format of an instruction varies between 1-15 bytes and has the following general form:

[prefixes] + code + [Mode R/M] + [SIB] + [displacement] + [immediate]

1. PREFIXES – control how an instruction is executed
   1. Optional (0 to max 4) – occupy 1 byte each
   2. Ex: may request repetitive execution of the current instruction; may block the address BUS during execution to not allow concurrent access to operand and results
2. CODE – the operation to be run
   1. Identified by 1 to 2 bytes
   2. Mandatory bytes
   3. Ex: add EAX, EBX; add (opcode) memory location B to memory loc. A
3. MODE R/M – register/memory mode
   1. 1 byte
   2. Specifies for some instructions the nature and the exact storage of operands(reg/mem)
   3. Allows the specification of a register/memory location described by an offset

OFFSET SPECIFICATION FORMULA:

[base] + [index\*scale] + [constant]

* + [base] + [index \* scale] = SIB (sale index base)
  + Constant = displacement + immediate

!!!This formula is a value, not an address

[] – deferencing operator

* An indication to the processor that you as a programmer need the value not the address
  + BASE: EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP
  + INDEX: EAX, EBX, ECX, EDX, EBP, ESI, EDI

!!!!!!! ESP IS NEVER AN INDEX!!!!!

* + SCALE: 1/2/4/8

OBS: As a consequence of the impossibility of appearing more than one Mode R/M, SIB and displacement fields in one instruction – the x86 architecture doesn’t aloe encoding of two memory addresses in the same instruction

* With the immediate value – define an operand as a numeric constant on 1/2/4 bytes

EXEMPLE: MOV EAX, [EBX]:

* + [EBX] – will be used as an offset into the memory
    - The value will go into the memory and will transfer the value from there to EAX
    - Basically, EBX – pointer value (faces to go into memory first and take the address from there
    - Obeys the offset spec. formula => correct

OBS: for an instruction there are 3 ways to express a required operand:

1. REGISTER MODE (the required operand = reg)
2. IMMEDIATE MODE (use directly the operand’s value, no adr or reg holding it)
3. MEMORY ADDRESING MODE(with offset spec. formula)

The offset of a variable can always be determined at assembly time. This is why you are using at least one register between [] it will be called indirect addressing.

If it is used only the name of the variable – from the point of off. Sp.f. it will be seen as a constant

MODES TO ACCESS MEMORY:

1. Directly addressing – when only constant is present
2. Based addressing – if in the computing one of the base reg. is present
3. Scale – indexed addressing – if in computing one theof the ndex reg is present
   1. These methods can be combined

Indirect addressing – based and/or indexed – one for which there is at least one register in []

JUMP INSTRUCTIONS:

* Use relative addressing

Relative addressing indicated the position of the next instruction to be run relative to rhe current position

This distance = expressed as the number of the bytes to jump over

CONVERSIONS CLASIFICATION:

* 1. DESTRUCTIVE – cbw, cwd, cwde, cdq, movzx, movsx; mov ah, 0; mov dx, 0; mov edx, 0

NON-DESTRUCTIVE – type operators(byte, word, dword, qword)

* 1. SIGNED - cbw, cwd, cwde, cdq, movsx

UNSIGNED – movzx; mov ah, 0; mov dx, 0; mov edx, 0, byte, word, dword, qword

* 1. BY ENLARGEMENT – all the destructive ones! + word, dword, qword

BY NARROWING – byte, word, dword

d.IMPLICIT VS EXPLICIT CONVERSIONS:

e = a+b+c

e,b = float

a,c – integer

TWO’S COMPLEMENT:

Mathematically, the 2’s complement representation of a NEGATIVE number is the value V- absolute value of the represented value

The value of a binary number starting with 1 is:

* + –(2’s complement of the initial binary configuration)

VARIANTS OF COMPUTING THE 2’S COMPLEMENT:

VARIANT 1: substracting the binary content from 100..000(where the nr of zeros are exactly the same with the no. of binary digits that need to be complemented)

VARIANTA 2: reversing the vlues of the bits from the initial binary configuration, after which it is added 1

VARIANT 3: there are left unchanged the values of the bits, starting from the right, until the first 1 bit inclusive, then all the other bytes are inversed

VARIANT 4: This method is used ONLY IF interested in the absolute value in base 10 of the 2’s complement:

* + The sum of the absolute values of the 2 complementary values is the cardinal of the set representable on that size.

Admissible representation intervals UNSIGNED/SIGNED (the same no. of values, only the interpretation matters)

1 BYTE:

* + [0, ] – UNSIGNED
  + [ ] – SIGNED

1 WORD:

* + [0, ] – UNSIGNED
  + [ ] – SIGNED

N BYTES:

* + [0, ] – UNSIGNED
  + [ ] – SIGNED

TYPES OF QUESTIONS TO ASK STARTING FROM A BINARY CONFIGURATION:

1. A number that begins with 0 in base 2 is the same in both representations!
2. Representation = 0xxxx..x

Value = abc

-abc = the 2’s complement of the initial binary configuration

A complementary value of an integer that stars with “0” will always start with 1(exception: 0) and it will fit in the admissible representation interval(both values are the same size: bytes, words, etc = -109 and 109)

1. Representation = 1xxx…x

(unsigned) = abc

Signed = -(2’s complement of the initial binary configuration)

1. Representation = 1xxx..x

Value = abc

-abc = also the 2’s complement of the initial binary configuration

BUT performing this, the result in binary will start with a 0(and we expect a negative number) = it will provide a natural number in binary from the same interval(or the same representation size)

Starting from value 1xxx..x = +abc, it will not obtain the value -abc on the same representation size!

WHY ISN’T THIS 2’S COMPLEMENT A BIDIRECTIONAL THING?

Look at the definitions which says: “Mathematically the 2’s complement representation of a negative number is the result of ”. So, we can relate to this formula only when our representation starts with 1(in our mind this means that binary thing)

We have a negative number, so we eant to find out it’s value – we use the -2’s complement of the binary config. Logically, this is why 2’s complement was invented. Because of computing the negative value of a binary number, without limitating the possible binary digits and decreasing the value.

Why numbers that start with o are always considered positive?

* + [0, ] – UNSIGNED – only positives
  + [ ] – SIGNED

The signed and unsigned interpretations of any binary configuration starting with 1 will be DIFFERENT and they will never be parts of the same admissible interval!

The absolute values of the 2 interpretations represent a 2 complementary values

OVERFLOW CONCEPT ANALYSIS:

DEF: At the level of the assembly language an overflowis a situation(condition which expresses the fact that the result of the LPO didn’t fit the reserved space for it OR does not belong to the admissible representation interval for that size OR the operation is a mathematical surse in the particular interpretation(signed/unsigned)

* + An addition is seen like this: b + b = b; w+w = w …
    - CF = 1 the unsigned interpretation in base 10 of the base 2 result addition is correct, BUT IF CF = 1 the unsigned representation in base 10 of the base two result addition is incorrect
    - OF = 0 the signed interpretation in base 10 of base 2 addition is correct, BUT IF OF = 1 the unsigned representation in base 10 of the base two result addition is incorrect

OVERFLOW FOR ADDITION:

When?

1. Positive + positive = negative (OF = 1)
2. Negative + Negative = positive (OF = 1)

Overflow for substraction:

1. Positive – negative = negative
2. Negative – positive = positive

Overflow for multiplication:

* + The multiplication does not produce overflow(the reserved space is enough for both interpretations), BUT in case of multiplication OF == CF!!!!
  + CF = OF = 0 if byte \* byte = byte sau word \* word = word
  + CF = OF = 1 if byte \*byte = word sau word \* word = dword

Overflow for division:

* + The worst effect in case of overflow is in the case for the division operation: the quotient doesn’t fit in the reserved space, then the division overflow will signal a “RUN TIME ERROR” and the OS will stop the program issuing one of these three messages:
    - DIVIDE OVERFLOW
    - DIVISION BY ZERO
    - ZERO DIVIDE

In the case of a correct division, CF and OF are undefined

OBS: In cazul in care adunam doua numere de semne diferite = NEVER OVERFLOW!

Daca adunam numere de acelasi semn – rezultatul are acelasi semn – no overflow

Necessitate imprumutului = OF = 1

ASSEMBLY LANGUAGE

DEF: Assembly language = programming language in which the basic instructions corresponds with the machine operations and which data structures are the machine primary structures

* + Is a symbolic language

SYMBOLS = mnemonics + labels

The basic elements with which an assembler works are:

1. LABELS = user-defined names for pointing to data or memory areas
2. INSTRUCTIONS = mnemonics which sugest the underlying action
   1. The assembler generates the bytes that codifies the corresponding instruction
3. DIRECTIVES = indications given to the assembler for correctly generating the corresponding bytes
4. LOCATION COUNTER = an integer number managed by the assembler for every separate memory segment
   1. At any given moment, the value of the location counter is the number of generated bytes correspondingly with the instructions and the directives already met in that segment(current offset inside that segment)
   2. Read-only access for the programmer - $ symbol

$ - evaluates to the assembly position at the beginning of the line containing the expression(JMP $ = LOOP)

$$ - evaluates to the start of the current section

$ - $$ = DISTANCE(scalar)

$, $$ - OFFSETS, POINTER TYPE, ADDRESS

$ Means “ADDRESS OF HERE”

$$ Means “ADDRESS OF THE START OF THE CURRENT SECTION”

$ - $$ == current size of section

SOURCE LINE FORMAT:

[label [:]] [ prefixes ] [ mnemonic ] [ operands ] [ ;comment ] – this is how a line of code look in asm

TWO CATEGORIES OF LABELS:

1. Code labels = present at the level of instructions sequences for defining the destinations of the control transfer during a program execution
   1. Can appear in data labels!
2. Data labels = provide symbolic identification for some memory locations
   1. From a semnatic point of view, it’s for variables
   2. Can appear in code segments

! The value associated with a label in assembly language is an integer number representing the address of the instruction/directive following that label

EXPRESSION = operands + operators

* + Expressions are evaluated at assembly time
    - Their values = computable at assembly time

Exception: operands = register contents (they are evaluated at run time) – offset specification formula

OPERANDS SPECIFICATION MODES:

* AT ASSEMBLY TIME:
  + Immediate operands
  + Direct addressed operands(offset part only)
* LOADING TIME:
  + Memory operands in direct addressing mode (as a complete FAR address)
  + Segment address is determinable here so the whole FAR address is known now – address relocation process(adjusting an address by fixing its segment )
* RUN TIME:
  + Register operands
  + Indirectly accessed memory operations

IMMEDIATE OPERANDS – computable at assembly time:

1. H/X – hexadecimal
2. D/T – decimal
3. Q/0 - octal
4. B/Y - binary

[straight brackets] = the variable name denotes the value of the variable

In any other context, the variable name denotes the address of the variable

OBS: Using [] will always indicate accessing an operand from memory

MNEMONICS = instruction names (guide the processor)

= directive names (guide the assembler)

OPERANDS = parameters which define the values to be processed by the instruct.

1. REGISTERS
2. CONSTANTS
3. LABELS
4. EXPRESSIONS
5. KEYWORDS
6. OTHER SYMBOLS

OBS: The offset of data labels and code labels are values computable at assembly time and they remain constant during the whole program’s runtime

* A variable once allocated in memory segment will never change its location(its position relative to the start of that segment)
* This information(the position) is determinable at assembly time based on the order in which variables are declared in the source code, and due to the dimension of representation inferred from the associated type information

Register operands:

* Direct using: mov eax, ebx
* Indirect usage and addressing: mov eax, [ebx] – for pointing memory locations

Memory addressing operands – can be direct or indirect

1. DIRECT ADDRESSING OPERANDS = constant or symbol – rep the address(segment and offset) for instruction/data:
2. LABELS
3. PROCEDURES NAMES
4. VALUE OF THE LOCATION COUNTER

The offset of a direct addressing operand is computed at assembly time. The address of every operand relative to the executable program’s structure(establishing the segments to ehich the computed offsets are relative to) is computed t linking time. The actual physical address is computed at loading time.

OBS: O adresa mereu e intr-un segment, fie ca e specificat sau nu. Daca nu e specificat, e ales implicit pe baza unor reguli.

The effective address always refers to a segment register. This register can be EXPLICITLY specified by the PROGRAMMER or otherwise a segment register is IMPLICITLY associated by the ASSEMBLER.

The implicit rules for performing the association with an EXPLICIT SPECIFIED OFFSET OPERAND are:

1. CS – code labels target of the control transafer intructions(jmp, call,ret,jz etc)
2. SS – in SIB addressing when using EBP or ESP as base(no matter of index of scale)
3. DS – for the rest of data accesses

ES: only explicit

* + Certain string instructions(MOVSB)

1. INDIRECT ADDRESSING OPERANDS = use registers for pointing to memory addresses

The actual registers values are known at runtime – indirect addressing is suited for dynamic data operations(move memory)

The general form for indirect addressing:

Base\_register + index \* scale + constant

When the operand is not specified by the complete formula (come components are missing) the assembler will solve the ambiguity by choosing the shortest encoding.

OBS: All codifications considered by the assembler are equivalent and its final decision doesn’t affect the functionality of the resulted code.

OPERATORS = perform computations only with constant SCALAR values computable at assembly time.

* + EXCEPTION|: adding/subtracting a constant from a pointer
  + EXCEPTION: offset computation formula(allows “+” operator)

Expression evaluation is done on 64 bits, the final result being adjusted to the sizeof available in the available usage context of that expression:

* + + performs addition at assembly time
  + ADD performs addition at runtime

BITWISE OPERATORS: ~, &, |, ^

~ = 1’s complement

& = and

| = or

^ = XOR

! = logical negation

**1. AND (Bitwise AND)**

* **Description**: Compares each bit of two operands. The result is 1 only if both bits are 1.
* **Purpose**: Clear bits or mask specific bits.
* **Syntax**:  
  AND destination, source

**2. OR (Bitwise OR)**

* **Description**: Compares each bit of two operands. The result is 1 if either bit is 1.
* **Purpose**: Set specific bits.
* **Syntax**:  
  OR destination, source

**3. XOR (Bitwise XOR)**

* **Description**: Compares each bit of two operands. The result is 1 if the bits are different.
* **Purpose**: Toggle bits.
* **Syntax**:  
  XOR destination, source

**4. TEST**

* **Description**: Performs a bitwise AND operation but does not store the result. Affects flags only.
* **Purpose**: Check if specific bits are set.
* **Syntax**:  
  TEST destination, source

**5. NOT (Bitwise NOT)**

* **Description**: Inverts each bit of the operand (1 becomes 0, and 0 becomes 1).
* **Purpose**: Perform bitwise negation.
* **Syntax**:  
  NOT destination

**6. SHL (Shift Left Logical)**

* **Description**: Shifts all bits to the left, filling with zeros. Each shift multiplies the value by 2.
* **Purpose**: Multiplication by powers of 2.
* **Syntax**:  
  SHL destination, count

**7. SHR (Shift Right Logical)**

* **Description**: Shifts all bits to the right, filling with zeros. Each shift divides the value by 2 (unsigned).
* **Purpose**: Division for unsigned integers.
* **Syntax**:  
  SHR destination, count

**8. SAL (Shift Arithmetic Left)**

* **Description**: Identical to SHL, shifts bits to the left, filling with zeros. Used for signed integers.
* **Purpose**: Multiplication by powers of 2 for signed numbers.
* **Syntax**:  
  SAL destination, count

**9. SAR (Shift Arithmetic Right)**

* **Description**: Shifts bits to the right, preserving the sign bit (MSB). Each shift divides the value by 2 (signed).
* **Purpose**: Division for signed integers.
* **Syntax**:  
  SAR destination, count

**10. ROL (Rotate Left)**

* **Description**: Rotates all bits to the left, with the most significant bit (MSB) becoming the least significant bit (LSB).
* **Purpose**: Circular shifts for cyclic operations.
* **Syntax**:  
  ROL destination, count

**11. ROR (Rotate Right)**

* **Description**: Rotates all bits to the right, with the least significant bit (LSB) becoming the most significant bit (MSB).
* **Purpose**: Circular shifts for cyclic operations.
* **Syntax**:  
  ROR destination, count

OBS: Segment specification operator (:) performs the FAR address computation of a variable or label relative to a certain segment.

segment: expression

TYPE OPERATORS:

type expression – synthax

What it does? It causes expression to be treated temporarily(limited to that particular instruction) as having “type” sizeof without destructively modifying its initial value( also called “non-destructive temporary conversion operators”)

* Byte, word, dword, qword, tword = memory stored operands

FAR NEAR = CODE LABELS

(6 bytes address) (4 bytes address)

IMPORTANT:

BYTE/DWORD/WORD/QWORD specifiers have always the task to clarify an ambiguity!

(inclusive when we talk about a memory variable)

LOCATION COUNTER AND POINTER ARITHMETIC:

Offsets of variables can be determined at assembly time. Offsets of segments cannot be determined at assembly time(known only at loading time)

Error types in Computer Science:

* Synthax error – diagnosed by assembler/compiler
* Run-time error(execution error) – program crashes(stop executing)
* Logical error – program, runs until its end or remains blocked in an infinite loop
* Fatal: linking error(ex: in case of a variable defined multiple times in a multimodule program(1 variable defined only in a single module))

STRING INSTRUCTIONS:

1. Data transfer:
2. Lodsb – byte from DS:ESI loaded in AL ; ESI este automat incrementat/decrementat cu 1
3. Lodsw – byte from DS:ESI loaded in AX ; ESI este automat incrementat/decrementat cu 2
4. Lodsd – byte from DS:ESI loaded in EAX ; ESI este automat incrementat/decrementat cu 4
5. STOSB – AL byte is stored in <ES:EDI: ; EDI este automat incrementat/decrementat cu 1
6. STOSW – AX byte is stored in <ES:EDI: ; EDI este automat incrementat/decrementat cu 2
7. STOSD – EAX byte is stored in <ES:EDI: ; EDI este automat incrementat/decrementat cu 4
8. MOVSB = LODSB + STOSB (byte/word/dword from <DS:EDI> TO <ES:EDI>

ESI/EDI = +/- 1,2,4

<DS: ESI> = source string

<ESI, EDI> = destination string

OBS: Because of flat memory model OS initializes DS = ES

1. Data comparison:

|  |  |
| --- | --- |
| SCASB | CMP AL, <ES:EDI> If DF=0 Then inc(EDI), Else dec(EDI) |
| SCASW | CMP AX, <ES:EDI> If DF=0 Then EDI:= EDI+2, Else EDI:= EDI-2 |
| SCASD | CMP EAX, <ES:EDI> If DF=0 Then EDI:= EDI+4, Else EDI:= EDI-4 |
| CMPSB | CMP <DS:ESI>, <ES:EDI> If DF=0 Then inc(ESI), inc(EDI), Else dec(ESI), dec(EDI) |
| CMPSW | CMP <DS:ESI>, <ES:EDI> If DF=0 Then ESI:= ESI+2, EDI:= EDI+2, Else ESI:= ESI-2, EDI:= EDI-2 |
| CMPSD | CMP <DS:ESI>, <ES:EDI> If DF=0 Then ESI:= ESI+4, EDI:= EDI+4, Else ESI:= ESI-4, EDI:= EDI-4 |

LODSB, STOS.., MOVS.. don’t change any flags

SCAS.., CMPS.. – change flags (because of cmp instruction)

1. Prefixes for repetitive instructions:

**repetitive\_prefix** **string\_instruction**

* equivalent to

**Again:**

**string\_instruction**

**LOOP Again**

* where *repetitive\_prefix* can be REP, equivalent to REPE (*Repeat While Equal*), REPZ (*Repeat While Zero*) - which repeat the execution of instructions SCAS or CMPS until ECX becomes 0 or an unmatch occurs ( => ZF=0)
* or it can be REPNE (*Repeat While Not Equal*) or REPNZ (*Repeat While Not Zero*) - which repeat the execution of instructions SCAS or CMPS until ECX becomes 0 or when a match occurs ( => ZF=1)

DATA TRANSFER INSTRUCTIONS:

* 1. PUSH:
* A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location in the stack pointed to by the stack pointer.
* The PUSH instruction first decrements ESP and then copies a source operand into the stack.   
  A 16-bit operand causes ESP to be decremented by 2. A 32-bit operand causes ESP to be decremented by 4.
* Figure 2 shows the effect of pushing 000000A5 on a stack that already contains one value (00000006). Notice that the ESP register always points to the top of the stack.
* The figure shows the stack ordering opposite to that of the stack of plates we saw earlier, because the runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, ESP = 00001000h; after the push, ESP = 00000FFCh.

A diagram of a graph

Description automatically generated with medium confidence

2. POP:

* A *pop*operation removes a value from the stack. After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next-highest location in the stack.
* The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:
* Figure 4 shows the stack before and after the value 00000002 is popped.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 4

* The area of the stack below ESP is *logically empty*, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

3. PUSHFD/POPFD:

* The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

**pushfd**

**popfd**

* 16-bit programs use the PUSHF instruction to push the 16-bit FLAGS register on the stack and POPF to pop the stack into FLAGS.
* in 32 bits programming both PUSHF and PUSHFD can be used in order to push the 32-bit EFLAGS on the stack
* The MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags. There are times when it is useful to make a backup copy of the flags so you can restore them to their former values later. Often, we enclose a block of code within PUSHFD and POPFD:
* When using pushes and pops of this type, be sure the program’s execution path does not skip over the POPFD instruction. When a program is modified over time, it can be tricky to remember where all the pushes and pops are located.

The need for precise documentation is critical!  
A less error-prone way to save and restore the flags is to push them on the stack and immediately pop them into a variable:

1. PUSHAD/PUSHA/POPAD/POPA:

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order:  
EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI.

The POPAD instruction pops the same registers off the stack in reverse order.

Similarly, in 16 bits programs the PUSHA instruction, introduced with the 80286 processor, pushes the 16-bit generalpurpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed.

The POPA instruction pops the same registers in reverse order.

in 32 bits programming POPA and POPAD, respectively PUSHA and PUSHAD have the same behavior

If you write a procedure that modifies a number of 32-bit registers, use PUSHAD at the beginning of the procedure and POPAD at the end to save and restore the registers.

* An important exception to the foregoing example must be pointed out; procedures returning results in one or more registers should not use PUSHA and PUSHAD.

1. PUSH s/ POP d:
   1. Operands s and d must always be doublewords, because the stack is organized on doublewords.
   2. The stack grows from big address to small address, 4 bytes at a time ESP pointing always at the top of the stack
2. PUSH/POP ESP
   1. PUSH ESP: The top of the stack = one position higher
   2. POP ESP: The top of the stack moves one position lower
3. XCHG instruction allows interchanging the contents of 2 operands of the same size:
   1. At least one operand = register
   2. The other one = register/memory address
   3. Because both operands mulst be L-values

L-values = any object that has an address

1. XLAT “translates” the byte from AL to another byte, using for that purpose a user-defined corespondence table called translation table

[reg\_segment] XLAT

Translation table = direct address of a string of bytes

The instruction requires the FAR address of the translation table provided:

* DS:EBX (implicit, if the segment is missing)
* Segment register : EBX(if the seg\_reg specified)

The definition of a variable:

* + Declaring its attributes(can be done MANY times)
  + Allocation(can be done ONCE)

The definition must be unique(that’s why the linkeditor gues an error)

ALLOCATION = means to associate the space neccesary to that var.

TIMES = directive that can be also applied for instructions

EQU = directive that allows assigning a numeric value or a string during assembly time to a label without allocating any memory space

CONCEPTS OF A VARIABLE:

VARIABLE = we express the fact that the content of that item is modifiable

= a variable is not a symbol, not just a name

A variable = pair of the memory address of some location and the contents

The symbol = name, but in assembly it denotes the memory address of that location

From a structural point of view, a variable is 4 things that represent its structure:

* 1. NAME
  2. SET OF ATRIBUTES:
  + Type (main attribute, in assembly = size)
    - Defines the domain of values
  + Visibility domain:
    - Interval of source program(measured in the distance in which that variable is accessible)
  + Lifetime variable:
    - How much does it live(from where you give it birth to where it dies)
    - The space attributed for a variable is cleared - val = dead
    - A variable is born when it’s allocated
    - Measured in seconds
  + Memory class:
    - How a variable is allocated

REFFERENCE (address) – optional to formal varriables

VALUE = You have a refference – you have a value

POINTER ARITHMETIC:

Arithmetic operations allowed for pointers(only operations that express as a result a correct location in memory useful as information for the programmer/processor)

Pointer arithmetic is the set of arithmetic operations allowed to be performed with pointers, this meaning using arithmetic expressions which have addresses as operands.

Contains only 3 operations that are possible:

1. Subtracting 2 addresses (address-address = ok)

Q – P = subtraction of two pointers = the number of bytes between these 2 addresses

1. Adding a numerical constant to a pointer (address + numerical constant)
   1. Identification of an element by indexing a[7], q+ 9
2. Subtracting a numerical constant from a pointer = address – numerical constant
   1. If you take the last possible memory address – which is representing on a dword – go outside a dword => MEMORY ACCESS VIOLATION

OBS: P+Q is supported in NASM. But P+Q doesn’t mean addition of 2 pointers (it is an expression with pointers, but not pointer arithmetic)

* + - Adding 2 addresses makes no sense, because you have no idea what you would point to.

VARIABLE VAR:

* + Var = address(offset)
  + [var] = its contents(deferencing operator) like \*(var)

L-value VS R-value:

1. LHS – Left Hand Side of an assignment – contains the address
2. RHS – Right Hand Side of an assignment – contains the content

LEA – LOAD EFFECTIVE ADDRESS – transfers the offset of the memory operand into the destination register

LEA general\_register, contents of a memory operand = SYNTHAX

GENERAL\_REGISTER

OBS: LEA has the advantage that the source operand may be an addressing expression(unlike “mov” which allows as a source operand only a variable with direct addressing)

PUSHF – transfers all the flags on top of the stack

POPF – extracts the word from top of the stack and transfer its contents into the EFLAGS register

CLC

CMC

STC

CLD

STD

The following four instructions are *flags transfer instructions*:

**LAHF** (***L****oad register* ***AH*** *from* ***F****lags*) copies SF, ZF, AF, PF and CF from FLAGS register in the bits 7, 6, 4, 2 and 0, respectively, of register AH. The contents of bits 5, 3 and 1 are undefined. Other flags are not affected (meaning that LAHF does not generate itself other effects on some other flags – it just transfers the flags values and that’s all).

**SAHF** (***S****tore register* ***AH*** *into* ***F****lags*) transfers the bits 7, 6, 4, 2 and 0 of register AH in SF, ZF, AF, PF and CF respectively, replacing the previous values of these flags.

**PUSHF** transfers all the flags on top of the stack (the contents of the EFLAGS register is transferred onto the stack).The values of the flags are not affected by this instruction. The **POPF** instruction extracts the word from top of the stack and transfer its contents into the EFLAGS register.

DIRECTIVES:

* Direct the way in which code and data are generated during assembling

Segment Directive = allows targeting the bytes of code or of data emitted by an assembler to a given segment, having a name and some specific characteristics.

SEGMENT\_NAME [type] [ALIGN = alignment] [ combination ]

[usage] [ CLASS = class ]

* Numeric value assigned to the segment name is the segment address(32 bits) corresponding to the memory’s segment’s position during runtime

$$ = current segment address(without knowing the current segment name)

OPTIONAL ARGUMENTS: alignment, combination, usage, class – give information to the link editor and assembler regarding the way in which segments must be loaded and combined in memory.

Type – allows selecting the usage mode of the segment:

* + Code(text) – segment = code(!the content can not be written, only executed)
  + Data – data segment allows reading&writing, no execution
  + Rdata-segment – only read, containing definitions of const. data

Alignment = specifies the multiple of bytes number from which that segment may start(only powers of 2)

OBS: alignment = missing

Combination = similar named segments will be combined with the current segment at linking time

* PUBLIC : tells to link-editor to concatenate this segment with other segment with the same name => a single segment (length = sum of concatenate segments)
* COMMON: larger segment, will include the current one with the same name
* PRIVATE: segment cannot be combined with others with the same name
* STACK: segments with the same name = concatenated during run-time => stack segment

Usage = allows choosing another word size more than 16 bits(default)

Class = allows choosing the order in which the link-editor puts the segments in memory; all segments with the same class will be placed in a contiguous block of memory – no implicit value

* + Segment code 32 class = code
  + Segment data use 32 class = data

DATA DEF DIRECTIVES:

Data definition = declaration(atrib spec) + allocation(reserving required memory space)

(UNIQUE) (NOT UNIQUE) (UNIQUE)

Data type = size of representation

Value = address of its first byte

Allocation type = uninitialized data reservation (RESB, RESD, RESQ)

Times directives allows repeated assembly of an instruction/data def

EQU directive – allows assigning a numeric value or a string during assembly time to a label without allocating any memory space or bytes generation

INSTRUCTION PREFIX BYTES:

X86 instruction can have up to 4 prefixes; each prefix adjusts interpretation of the opcode

STRING MANIPULATION (instr. Prefixes – provided explicitly by the programmer)

REP – repeats instruction the number of times specified by iteration count ECX

REPNE – allow to terminate loop on the value of ZF CPU flag

**Types of string instructions:**

* which use a source string and a destination string (MOVSB, MOVSW, MOVSD, CMPSB, CMPSW, CMPSD)
* which use only a source string (LODSB, LODSW, LODSD)
* which use only a destination string (STOSB, STOSW, STOSD, SCASB, SCASW, SCASD)

**You will remember them more easily:)**

* Move String = MOVS
* Compare String = CMPS
* Load String = LODS
* Store String = STOS
* Scan String = SCAS

**A string is characterized by:**

* *the type of the elements(bytes or words)*=> is given by the last letter of the instruction that is used (B=byte, W=word, D=doubleword), both strings having the same type
* *the address of the first element* => is a FAR memory address which is memorised in:
  + in DS:ESI - for the source string
  + in ES:EDI - for the destination string
* *the parsing direction* => is given by the value of the DF flag (0 - from small addresses to large addresses, 1 - from large addresses to small addresses.)
* *the number of elements*=> when needed is placed in CX or ECX

Segment override prefix causes memory access to use specified segment instead of default segment for instruction operand.

NEAR/FAR ADDRESSES

We present below a very relevant example for understanding the control transfer to a label, highlighting the

differences between a direct transfer vs. an indirect one.

segment data

aici DD here ;equiv. with aici := offsetul of label here from code segment

segment code

. . .

mov eax, [aici] ;we load EAX with the contents of variable aici (that is the offset of here

inside the code segment *–* same effect as **mov eax, here**)

mov ebx, aici ;we load EBX with the offset of aici inside the data segment

A diagram of a flowchart

Description automatically generated

jmp [aici] ;jump to the address indicated by the value of variable aici(which is the address of

here), so this is an instruction equiv with jmp here ; what does in contrast **jmp**

**aici** ??? – the same thing as jmp ebx ! **jump to CS:EIP with EIP=offset (aici)**

**from SEGMENT DATA (00401000h)** ; jump to some instructions going until

the first „access violation”

jmp here ;jump to the address of here (or, equiv, jump to label here); **jmp [here]** ?? – JMP

DWORD PTR DS:[00402014] – most probably „Access violation”

....

jmp eax ;jump to the address indicated by the contents of EAX (accessing register in direct

mode), that is to label here ; in contrast, what does **jmp [eax]** ??? JMP DWORD PTR DS:[EAX] – most probably „Access violation”....

jmp [ebx] ;jump to the address stored in the memory location having the address the contents of

EBX (indirect register access – the only indirect access from this example) – what

does in contrast **jmp ebx** ??? - jump to **CS:EIP with EIP=offset (aici) from the**

**SEGMENT DATA (00401000h)** ; jump to some instructions going until the

first „access violation” ;in EBX we have the address of variable aici, so the contents of this variable will be accessed. In this

memory location we find the offset of label here, so a jump to address here will be performed *-*

**consequently, the last 4 instructions from above are all equivalent to jmp here**

A diagram of a diagram

Description automatically generated

Explanations on the interaction between the implicit association rules of an offset with the

corresponding segment register and performing the corresponding jump to the specified

offset

JMP [var\_mem] ; JMP DWORD PTR DS:[00401704] - NEAR jump to offset from DS:[00401704]

JMP [EBX] ; JMP DWORD PTR DS:[EBX] - NEAR jump to offset from DS:[EBX]

JMP [EBP] ; JMP DWORD PTR SS:[EBP] - NEAR jump to offset from SS:[EBP]

Accordingly to the implicit association rules of an offset with the corresponding segment register

The INDIRECT addressing operand after JMP tells us FROM WHERE to take the OFFSET to perform the NEAR jump to (jump within the current code segment). The last instruction expresses a DIRECT jump to the offset computed as the value associated with the here label (jump to CS:here).

Even if we use explicit prefixing with a segment register of the destination operand, the jump will NOT be a FAR one. FAR will only be the ADDRESS from which the OFFSET will be taken where the NEAR jump will be made.

If we want to perform the jump in a different segment (FAR jump) we must explicitly

specify this by means of the FAR type operator, which will impose the destination operand

of the JMP instruction to be treated as a FAR ADDRESS:

jmp far [ebx + 12] => CS : EIP <- far address (48 bits = 6 bytes) equiv. to

jmp far [DS: ebx + 12] => CS : EIP <- far address (48 bits = 6 bytes

In short and as a conclusion we have :

- the value of the pointer representing the address where the jump has to be made can be stored

anywhere in memory, this meaning that any offset specification that is valid for a MOV instruction can be also present as an operator for a JMP instruction (for example jmp [gs:ebx + esi\*8 - 1023])

- the contents of the pointer (the bytes taken from that specific memory address) may be near or far, depending on how the programmer specifies or not the FAR operator, being thus applied either only to EIP (if the jump is NEAR), either to the pair CS:EIP if the jump is FAR.

Any jump can be considered hypothetically equivalent to MOV instructions such as:

- jmp [gs:ebx + esi \* 8 - 1023] <=> “mov EIP, DWORD [gs:ebx + esi \* 8 - 1023]”

- jmp FAR [gs:ebx + esi \* 8 - 1023] <=> “mov EIP, DWORD [gs:ebx + esi \* 8 - 1023]” +

“mov CS, WORD [gs:ebx + esi \* 8 - 1023 + 4]”

Final conclusions.

- NEAR jumps – can be accomplished through any of the three operand types (label, register,

memory addressing operand)

- FAR jumps (this meaning modifying also the CS value, not only that from EIP) – can be

performed ONLY by a memory addressing operand on 48 bits (pointer FAR). Why only so

and not also by labels or registers ?

- - if we would have used labels, even if we jump into another segment (an action possible

as you can see above) this is not considered a FAR jump because CS is not modified (due to

the implemented memory model – Flat Memory Model). Only EIP will be changed and the

jump is technically considered to be a NEAR one.

- - if we would have used registers as operands we may not perform a far jump, because

registers are on 32 bits and we may so specify maximum an offset (NEAR jump), so we are

practically in the case when it is impossible to specify a FAR jump using only a 32 bits

operand.

FILE OPERATIONS:

File descriptor = numerical value that identifies an open file

= is for file

= it is used to reference the file when reading from or writing to it

! Arguments for functions are placed on the stack(in reverse order) from right to left

* when call is executed, it pushes the address of the instruction immediately after itself(the return address) – WHY? So when ‘ret’ is executed to pop that address and the program to know where should be after the call)

call print\_hello – stores the location where the program should continue executing once the procedure is finished

print\_hello:

……….

ret

When the procedure is finished, it uses ‘ret’ instruction to pop the return sddress off the stack and transfers control back to the instruction after the ‘call’ instruction.

Implicit result returned by the functions are stored in EAX? The function doesn’t empty the stack, it is the responsibility of the programmer to pop the arguments after the function call.

The fopen function respects the *cdecl* convention and it can be found in the *msvcrt.dll* library.

***Arguments of the fopen function:***

The first argument of the function is the address of a character string containing the name of the function. The second argument is the address of a character string containing the access mode for opening the file.

|  |  |  |
| --- | --- | --- |
| Access mode | Meaning | Description |
| r | read | - Open file for reading. The file must exist. |
| w | write | - If the file does not exist, it creates a new file with the given name and opens it for writing.  - If a file with the given name exists, it opens it for writing. It overwrites the content of the file and starts writing from the beginning of the file. |
| a | append | - If the file does not exist, it creates a new file and opens it for writing.  - If a file with the given name exists, it opens it for writing. It does not overwrite the content, it continues writing at the end of the file. |
| r+ | read and write from/into existing file | - Open file for reading and writing. The file must exist. |
| w+ | read and write | - If the file does not exist, it creates a new file and opens it for reading and writing.  - If a file with the given name exists, it opens it for reading and writing. It overwrites the content of the file and starts writing at the beginning of the file. |
| a+ | read and append | - If the file does not exist, it creates a new file and opens it for reading and writing.  - If a file with the given name exists, it opens it for reading and writing. It does not overwrite the content, it continues writing at the end of the file. |

***Observations:***

* The name of the function must include the extension (ex: name.txt, example.asm).
* File are created or opened in the current directory (in the same directory where the asm source is located). Important: in order to open a file using its name, the file must be placed in the same folder as the asm source file, otherwise the file will not be found.
* Writing operations will fail if the file was opened only for reading (ex. access mode "r"). Reading operations will fail for files opened only for writing or appending (ex. access mode "w", "a").
* Both arguments of the *fopen* function represent character strings that have to be terminated with a 0 (similar to the format for the printf function).
* **The value returned by the fopen function:**
* If the file is successfully opened, EAX will contain the file descriptor (an identifier) which can be used for working with the file (reading and writing). If an error occurs, fopen will set EAX to the value 0.
* ***Other observations:***
* It is important to check the value returned by the function in EAX before continuing to work with the file, in order to know whether the file was correctly opened. If a program opens more files using the fopen function, each value returned by the function must be stored separately, since each file has a unique identifier. When we are done working with a file, it is important to close the file ( it can be done at the end of the program - before exit). For closing the file we use the *fclose* function

The fprintf function respects the *cdecl* convention and it can be found in the *msvcrt.dll* library. Syntax of the *fprintf*function is similar to the syntax of the *printf* function (used for printing on the screen). The difference is that, in addition to the parameters of the printf function, the fprintf function has the file descriptor as the first argument.

* ***Arguments of the fprintf function***
* First argument of the function represents the file descriptor (identifier) returned by the *fopen* function call. The next argument of the function is a character string containing the format for printing, followed by the same number of arguments (constant values or variable names) as specified in the format. Similar to the printf function, the character string representing the format can contain certain formatting markups, starting with the character ’%’, which will be replaced by the values given in the following arguments.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Type** | **Example** | **Value representation dimension** |
| c | Character | a | byte |
| d or i | Signed decimal integer | 392 | dword |
| u | Unsigned decimal integer | 7235 | dword |
| x | Unsigned hexadecimal integer | 7fa | dword |
| s | String (terminated with a 0) | example | string of bytes terminated with 0 |

The fread function respects the *cdecl* convention and it can be found in the *msvcrt.dll* library.

***Arguments of the fread function***

The first argument of the fread functions represents the address of a string where the data read from file is stored. The second argument represent the size of one element that will be read from the file. The third argument represents the maximum number of elements to be read. The last argument of the function represents the file descriptor (identifier) returned by the *fopen* function call. When reading from a text file, the first argument of the fread function is a byte string and the second argument is 1 (= dimension of one byte). The third argument is the size of the byte string (number of elements).

***The value returned by the fread function::***

The fread function stores in EAX the number of elements read. If this number is below the value of the *count*argument, it means that either there was an error, or that the function got to the end of the file.

***Observations:***

If text files have large sizes, one cannot read the whole content of the file with one function call. In that case multiple fread function calls are necessary, until the whole content of the file is read. In the „Examples” section we will present such an example. In order to check whether we got to the end of the file, we can check if the value returned by fread is 0.

MULTIMODUL PROGRAMMING:

Modularization = split of subproblems (for reusability)

Separete compilation = the possibility offered by a programming language to compile separete files at different moments in time and link them together sometimes in the future as the step for obtaining the executable program

Global = export; extern = import

* Everything is exported in C;

**1. Modularization and Separate Compilation**

* **Modularization**:  
  Modularization is the process of dividing a program into smaller, self-contained submodules or functions. This promotes **reusability**, **maintainability**, and **ease of debugging**. Each module can perform a specific task and can be independently tested or updated.
* **Separate Compilation**:  
  Separate compilation refers to the ability to compile program modules (written in separate files) independently. The linker combines these compiled modules into a single executable during the final build phase. This allows for:
  + Faster compilation of individual modules.
  + Code reuse across different projects.
  + Better management of large codebases.

**2. Communication Between Modules**

In **assembly language**, two different modules can communicate through:

1. **Import/Export Mechanism**:
   * Use of global and external declarations:
     + global: Makes a symbol available for other modules to use.
     + extern: Declares a symbol defined in another module.

Example:

asm

CopiazăEditează

; Module 1

global myFunction

myFunction:

; Function implementation

; Module 2

extern myFunction

call myFunction

1. **General Registers**:  
   Registers like EAX, EBX, ECX, and EDX can hold values shared between modules. However, volatile registers (e.g., EAX, ECX, EDX) should be saved if modified.
2. **The Stack**:
   * Used to pass arguments, save return addresses, and manage local variables.
   * Each module can use the stack for temporary storage, adhering to calling conventions to ensure proper cleanup.

**3. ASM + High-Level Programming Integration**

**Phases of Function/Procedure Call**

The lifecycle of a procedure call includes three main phases:

1. **Call Phase**:
   * Prepares the stack and registers for the procedure call.
   * Saves volatile resources if needed.
   * Executes the actual call (e.g., call instruction in ASM).
2. **Entry Phase**:
   * Sets up the stack frame for the called function.
   * Initializes local variables and saves non-volatile registers.
3. **Exit Phase**:
   * Cleans up the stack and restores saved registers.
   * Returns control to the calling function.

**4. Calling Conventions**

**CDECL Convention**

* Default calling convention for C programs.
* Characteristics:
  + Parameters are passed on the stack, **from right to left**.
  + Caller cleans up the stack after the call.
  + Return values are typically stored in the EAX register.
  + Volatile registers (EAX, ECX, EDX) may be modified.

**STDCALL Convention**

* Standard calling convention used in Windows API.
* Characteristics:
  + Parameters are passed on the stack, **from right to left**.
  + Callee cleans up the stack after execution.
  + Return values are stored in EAX.

Example in STDCALL:

asm

CopiazăEditează

push param2

push param1

call myFunction

; No cleanup required by the caller

**5. Subroutine Call**

**Steps in Subroutine Call**

1. **Call Code**:
   * Prepares the stack by saving volatile registers and arguments.
   * Transfers control to the subroutine.
2. **Entry Code**:
   * Creates a new stack frame (e.g., push ebp and mov ebp, esp).
   * Saves non-volatile registers and initializes local variables.
3. **Exit Code**:
   * Frees up stack space used by the subroutine.
   * Restores registers and returns to the caller.

**Call Code Tasks**

1. Save volatile resources in use (e.g., push registers like EAX).
2. Ensure compliance with constraints (e.g., alignment).
3. Push arguments onto the stack.
4. Execute the call using:
   * **Statically linked subroutines**: Directly call the address of the function.
   * **Dynamically linked subroutines**: Use indirect calls via a table.

Volatile vs. non-volatile resources

- volatile resources are represented by those registers that the calling

convention is defining them as belonging to the called subroutine, thus,

the caller being responsible as part of the call code to save their values (if

the called subroutine is using them) and after that, at the end of the call to

restore the initial (old) values. So: who is saving the volatile resources ? The

caller (as part of the call code) . Who is restoring in the end those values ?

Also the caller but NOT as part of a certain call/entry or exit code. Just restore

them after the call in the regular code as a mandatory responsibility.

- non-volatile resources are any memory addresses or registers which do

not belong explicitly to the called subroutine, but if this one needs to modify

those resources, it is necessary that the called subroutine to save them at the

entry as part of the entry code and restore them back at exit, as part of the

exit code. So: who is saving the non-volatile resources ? The callee (apelatul =

the called subroutine, as part of the entry code) . Who is restoring in the end

these values ? Also the callee (as part of the exit code).