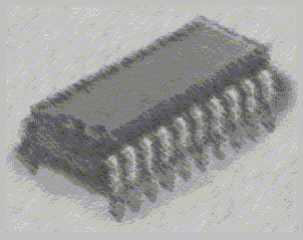
Lesson 2. Instruction set design

Computers Structure and Organization

Graduated in Computer Sciences
Graduated in Computers Engineering



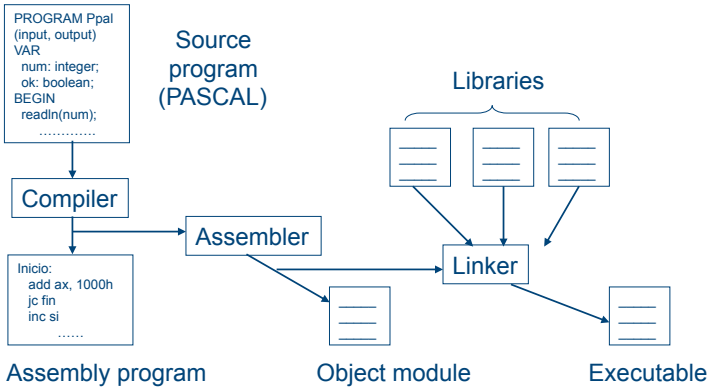
Contents

Basic concepts (I). Definitions

- **Instruction set**: is the set of all the operations the computer is able to perform.
- **Instruction**: is an operation represented in 0's and 1's which will be interpreted and executed by the computer
- **Machine instruction**: 0's and 1's string corresponding to a single instruction
- **Machine code**: is the representation of the instruction set as machine instructions
- **Assembly language**: is a set of mnemonics. These mnemonics correspond to machine instruction. Each mnemonic correspond to a single machine instruction

Basci concepts (II). Programming

- High level language programa creation process



High level languages and instr. set (I). Operands storage (I)

- How high level languages and operation system services have an influence on the instruction set design

```
TYPE vector = ARRAY [1..n] OF tipoelemento;
PROCEDURE IntercambioDirecto (VAR v: vector);
VAR i, j: integer;
      elemento: tipoelemento;
BEGIN
  FOR i:= 2 TO n DO
    FOR j:= n DOWNTO i DO
      IF v[j-i] > v[j] THEN
      BEGIN
        elemento := v[j-i];
        v[j-i] := v [j];
        v[j] := elemento
      END
END;
```

- Very often used variables storage in registers.
- How many operands are modified per instruction
- Data and addressing modes used
- Compare operation and conditional and unconditional jumps

High level languages and instr. set (&II). Operands storage (and II)

- How high level languages and operation system services have an influence on the instruction set design

```
PROGRAM Principal (input, output);
VAR   v: vector;

BEGIN
   LeerVector ( v );
   IntercambioDirecto ( v );
   MostrarVector ( V );
   writeln ('Progama finalizado')
END.
```

- Functions and procedures calls
- Operting systems service invocation

Instruction set parameters (I)

- What to be taken into account when designing instruction sets

What to take into account	Related to
CPU operands storage	Locations to storage operands: main memory, registers, …
Explicit operands per instruction	How many operands are designated in an instruction
Operand location	Can ALU operate with memory operands? How is a memory address specified?
Operations	What operations are considered by the instruction set?
Operand sizes and types	How is each operand specified? Which are operand sizes and types?

Instruction set parameters (II). Outside CPU storage alternatives

Temporal storage	Explicit operands per ALU instruction	Target result	Accessing to explicit operands
Stack machine	0	Stack	Push / Pop
Accumulator machine	1	Accumulator	Accumulator Load / Store
Register file machine	2 or 3	Registers or memory addresses	Register or memory addresses load / store

Machine type	Advantages	Drawbacks
Stack machine	Useful to evaluate expression Good density code moreover instructions are sort	Impossible random stack access Difficulty for efficient code generation and implementation
Accumulator machine 1-operand machine	Minimizes internal state machine Instructions are sort	A lot of traffic between accumulator and memory
Register file machine	General efficient code generation model	Long instructions to name registers

Instruction set parameters (III). Memory storage

- **General purpose register file machines advantages (GPR)**
  - Flexibility in evaluating expression and reordering code
  - Register can be used to storage often used variables to reduce traffic between memory and CPU
- **Compiler designers preffer non-dedicated registers**
- **The number of registers depend on how will be used by the compiler.**
  - Requiered registers for expression evaluations
  - Requiered register for parameter passing
  - Required register for variables storage

Instruction set parameters (IV). Memory storage

- **GPR classifiction:**
  - ALU number of oferands 2 or 3
  - ALU memory operands from 0 to 3

GPR type	Advantages	Drawbacks
Register-Register	• Fixed-length simple instruction coding • Simple code generation model • Similar cycle number in executing instructions	• More instruction per program than memory reference architectures • Excessive bits codification to get sort instructions
Register-Memory	• Data can be accessed from memory • Good code density because it's easy to codify instructions	• Operands are not equivalent. One operand is source and target of the operation • Codifying one register and one memory address per instruction may reduce the number of available registers • Cycles number execution varies when different addressing modes are used
Memory-Memory	• Compact memory model • Temporal register are not used	• Instruction sizes vary a lot • Instruction cycles required vary a lot • Memory becomes a bottleneck

Instruction set parameters (V). Memory addressing

- How is a memory address interpretated? Byte, half word, word, double word
- There two ways to storage the bytes of a word in memory depending on where the xx…xx00 byte address is located.
- 32 bits word 12345678h is to be stored from 100h memory address.

Addresses		100h	101h	102h	103h
Content	Little endian	78	56	34	12
	Big endian	12	34	56	78

Instruction set parameters (VI). Memory addressing

- Longer than a byte objects must be alligned in some architectures
- And object od ***s*** bytes is alligned on ***A*** address if A mod s = 0

A 32 bits memory allocation from 102h main memory address

Word 100h	Byte 100h	Byte 101h	Byte 102h	Byte 103h
Word 104h	Byte 104h	Byte 105h	Byte 106h	Byte 107h

102 mod 4 = 2 → word is not aligned → two memory accesses are required to be read

A 32 bits memory allocation from 104h main memory address

Word 100h	Byte 100h	Byte 101h	Byte 102h	Byte 103h
Word 104h	Byte 104h	Byte 105h	Byte 106h	Byte 107h

104 mod 4 = 0 → word is aligned → only one memory access is required to be read

Instruction set parameters (VII). Memory addressing

- A non-alligned memory access has several memory references
- Even non required allignement machines are faster in accessig alligned memory references
- Aligned and non-alligned different object sizes examples:

Addressed object	Byte aligned	Non byte aligned
Byte (8 bits)	0, 1, 2, 3, 4, 5, 6, 7	Never
Half word (16 bits)	0, 2, 4, 6	1, 3, 5, 7
Word (32 bits)	0, 4	1, 2, 3, 5, 6, 7
Double word (64 bits)	0	1, 2, 3, 4, 5, 6, 7

Instruction set parameters (VIII). Addressing modes

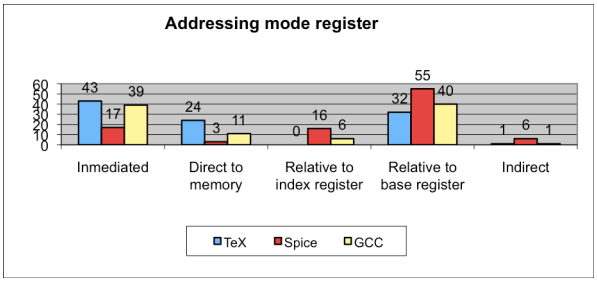
- **How an instruction can access to the operands**
  - **Inmediated**: operand is inside the machine code instruction.
  - **Direct to register**: machine code instruction contains the name of the register
  - **Direct to memory**: machine code instriction codifies the operand memory address.
  - **Relative**: an offset must be added to a register to access to the operand
  - **Indirect**: the coded address inside the machine code instruction is not the address itselft but its memory address
  - **Implícit**: there is not any reference to the operand in the machine code instruction because instruction works with a fixed operand
- **The name of the addressing mode may change from one vendors to others.**

Instruction set parameters (IX). Addressing modes

- **Addressing modes influence:**
  - Addressing mode instruction influences in the number of cycles execution
  - Cycle clock is influenced by addressing modes
  - Addressing mode set influences in hardware complexity
- **Addressing modes dependency:**
  - Addressing modes codification depends on the relationship between them and operation codes
  - Addressing modes codification depends on the allowed range for specified addreesing modes, e.g. offsets

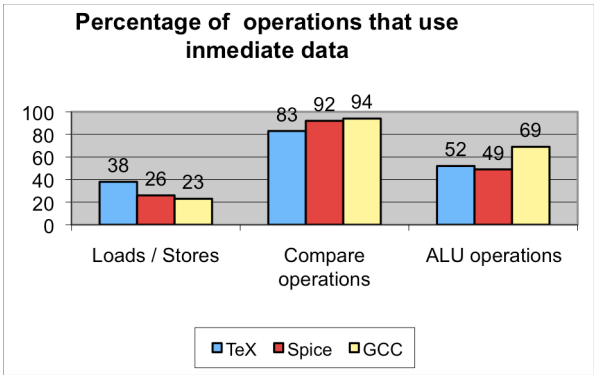
Instruction set parameters (X). Addressing modes use frequence

- VAX addressing mode use frequency



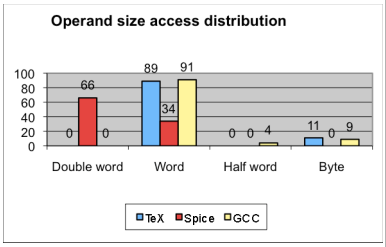
Instruction set parameters (XI). Addressing modes use frequence

- VAX immediate data use frequency



Instruction set parameters (XII). Operands size

- **How to codify an operand:**
  - Operand is codified in operation code
  - Tagged operands (operand type and operation)
- **Operand type shows its size**
- **Often operand sizez:**
  - Byte
  - Half word (16 bits)
  - Word (32 bits)
  - Single precision floating point (32 bits)
  - Double precision floating point (64 bits)
  - Character (byte) ASCII or EBCDIC
  - Packed or unpacked BCD



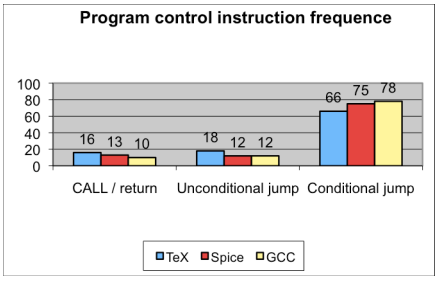
Instruction set parameters (XIII). Instruction set

Operator type	Examples
Logic or arithmetic	Logical and arithmetical integer operations
Data transference	Loads and stores
Control	Conditional and unconditional jumps, procedures calls and returns
System	Operating Systems Calls, virtual memory instructions
Floating point	Floating point operations
Decimal	Decimal addition and multiplication. Decimal rto character casting
Strings	String instructions

- Arithmetical-logical, control, and data transference instructions are supported by all microprocessors
- System instructions may vary a lot from one architecture to another
- Floating point, decimal and string instruction may, or may not, exist in different microprocessors

Instruction set parameters (XIV). Instruction set

- **Program control instructions:**
  - Modify program execution instruction sequence
  - Types of control instructions
    - Conditional jumps
    - Unconditional jumps
    - Procedures / interrupts calls
    - Procedures / interrupts returns
  - Jumping addresses must be always specified
  - Jumps are specified by using relative to program counter addressing mode

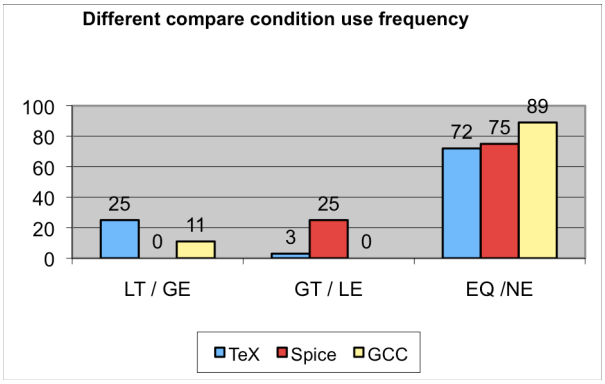


Instruction set parameters (XV). Instruction set

- **How to evaluate jump conditions:**
  - **Flag register**: one or more bits of the flag register record the result of the operaration, e.g., carry, signed, ...
    - Advantage: only one bit is used
    - Drawbacks: superescalar dependence problems
  - **One general purpose register**: one register is set to 0 or to 1 dependign on the result of the operation
    - Advantage: Dependence is simplified
    - Drawback: A full register is required
  - **By using a test and set instruction**: instruction for generating the condition and jumping is the same
    - Advantage: is solved in only one instruction
    - Drawback: CPI instruction may be increased

Instruction set parameters (XVI). Instruction set

- Conditional jump comparation frequency



Instruction set parameters (and XVII). To take into account when designing

- **Summary of parameters to be taken into account when designing an instruction set:**
  - **Parameter 1**: CPU operands storage
  - **Parameter 2**: number of explicit operand in each instruction
  - **Parameter 3**: addressing modes
  - **Parameter 4**: Operand types and sizes
  - **Parameter 5**. instruction types to be taken into account by the instruction set

Instruction formats (I). Design alternatives

- **CISC or RISC are available when designing**

- **CISC**: Complex Instruction Set Computer
  - A lot of addressing modes
  - A great deal of operation types
  - High complexity instruction rarely used
  - E.g. I80x86, M680x0

- **RISC**: Reduced Instruction Set Computer
  - A few addressing modes
  - Simple and few instruction types
  - Very regular instruction formats
  - E.g. ARM, MIPS R4000, SPARC

Instruction formats (II). Instruction coding

- **Criteria to be taken into account:**
  - Source code size is related to memory accesses and therefore to execution time.
  - Type and number of instructions to be decoded is related to execution time again
  - The computer must be easily programmed
- **Balance in design**:
  - Between number of registers and addressing modes
  - Instruction size must be in multiples of the word
  - Memory accesses must be in word or cache line size

Instruction formats (III). Instruction coding

- **Instruction code must contained:**
  - **Operation code**: identifies the operation to be performed
  - **Source operands**: the operands on which operation is performed
  - **Target operand**: where result will be stored
  - **Next instruction address**: where is next program instruction
- **To take into account:**
  - CISC, such as i80x86, often use one of the operands as source and target of the instruction
  - RISC usually employ three operands for ALU operations: two sources and one target

Instruction formats (IV). Instruction coding

- **General characteristics:**
  - **Systematic format**: fields use fixed positions.
  - **Operation code**: or its extension is the first field
  - **They are multiple of the computer word**: for memory access optimization
- **Instruction format alternatives:**
  - **Fixed format**: it's very difficult to implement it because it uses the same format for all instruction types
  - **Variable format**: operation code, extended operation codes, variable number of operands and different addressing modes are used
  - **Mixed format** : two or three fixed format are use to fit instructions
  - **Orthogonal format**: whatever instruction may use whatever operand and whatever addressing mode

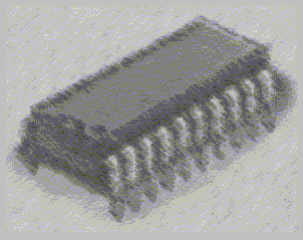
Instruction formats (and V). Instruction coding

- **Advices to be taken into account when designing instruction sets:**
  - **Balance**. Try to get number of registers and addressing modes balanced
  - **Instruction size must be multiple of computer word**. The best: the instruction size is only one word
  - **Source program code**. Depending on the addressing modes, number of registers and instruction types source program code may be longer or not and then execution time also depends on it
  - **Decoding**. Instruction decoding time increases when using complex instructions or addressing modes. Execution time depends on it too
  - **Programming**. Depending on the number of addressing modes and register it will be easy, or not, to program the computer
  - **Compiler**. Instruction formats must facilitate to compiler code generation

Lesson 2. Instruction set design

Computers Structure and Organization

Graduated in Computer Sciences
Graduated in Computers Engnieering



Computers Structure and Organization.
Graduate in Computer Sciences
Graduate in Computer Engineering

Achademic course 2011-2012
Automatic Department

Contents

- Basic definitions
- High level languages influence on instruction set design
- Instructions set parameter to be taken into account
- Intruction format
- VLIW technology
- Binary code compatibility
- Compilers
- Instruction set examples
- Bibliography

Contents

- Basic definitions
- High level languages influence on instruction set design
- Instructions set parameter to be taken into account
- Intruction format
- VLIW technology
- Binary compatibility
- Compilers
- Instruction set examples
- Bibliography

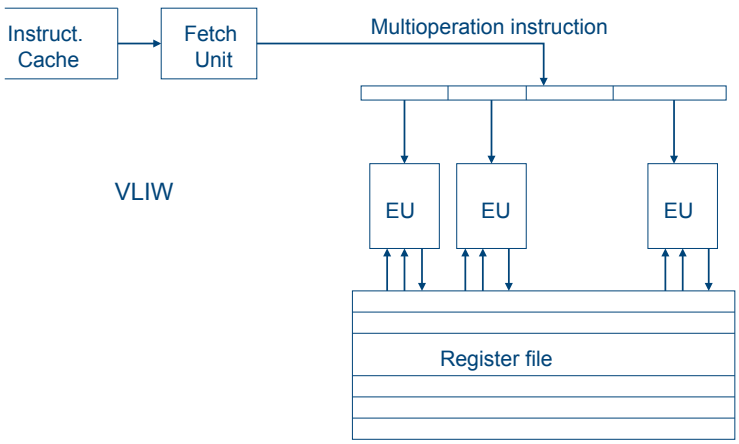
VLIW technology (I)

- **RISC technology easily allows microprocessor segmentation. In such way, several instruction may be executed at once**
- **Dependencies between instructions appear in segmentation. Two alternatives to face them:**
  - To detect dependencies and reorder the source code in execution time. Hardware is required
  - To detect dependecies and reorder the source code in compilation time
- **VLIW technology uses the second alternative**

VLIW technology (II)

- VLIW machines have a compiler that group several independent instruction in an only big instruction. They are splitted in independent operation and dispatched in different functional units when they are decoded by the control unit
- Compiler packs independent instruction in a big one
- Functional units and clock speed may be improved because no hardware is required to detect and correct dependecies in execution time
- Drawback: each compiler only works on an architecture (CISC or RISC)

VLIW technology (and III)



Binary code compatibility (I)

- Binary code translation is the used technique to change a designed program for an architecture and operating system into another program designed to run in a different archicture and operating system
- Using native compilers and source program files isthe best alternative but it's not frequently possible

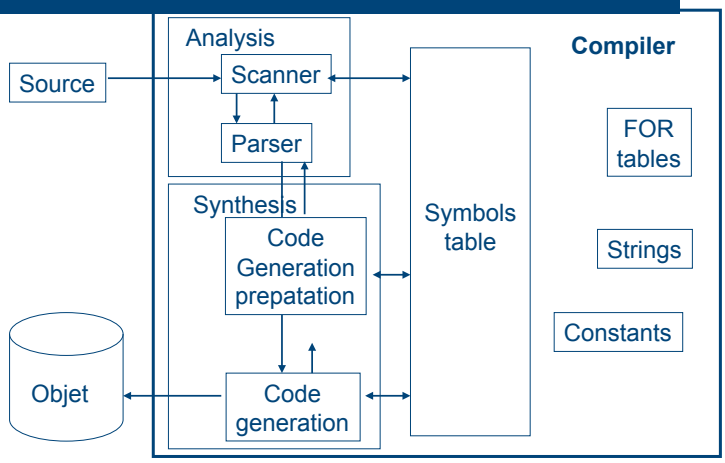
Binary code compatibility (and II)

- **Binary code translation techniques:**
  - **Software interpreter.** A program reads instruction by instruction the program of the old architecture and translate it to the new one. They are not very quickly
  - **Microcode emulator**. Similar to above technique but it only works in microcode machines by adding hardware to help in old instruction translation
  - **Binary code translator**. They are sequences of instructions of the new architecture that translate the old one sentences. Some of the flag register information is stored in the new one.
  - **Native compilers.** It's the quickest alternative by recompiling last source program

Compilers (I)

- **Compiler**: it's a program that translates a source program created in a high level language to a machine code object program ready to run in a computer with few or none additional information (PSP)
- **Compilers types**:
  - **Assembler**. Translates a mnemonic language to machine code language
  - **Crossed compiler**. Translates to a different computer and operating system that using for compilation it
  - **One or more passed compiler**. One or more passes must be taken to get object program
  - **Incremental compiler**. Compiles discovered fixed errors only
  - **Decompiler**. It's the opposite process of a compiler

Compilers (II). Compiler structure



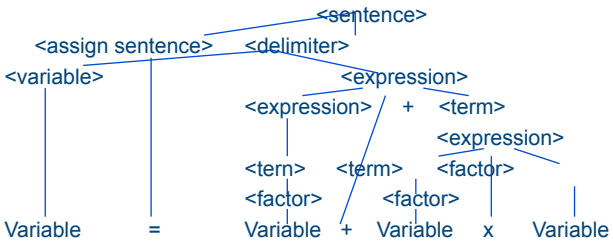
Compilers (III). Compilation process example

- **Pascal sentence to be compiled:**
  - Speed:= InitialSpeed+ Acceleration x Time;

Scanner output:

  - Variable = Variable + Variable x Variable Delimiter

- **Parser output:**



Syntax analysis result: it's a valid Pascal sentence

Compilers (IV). Compilation process example

- **Parser code generation preparation output:**

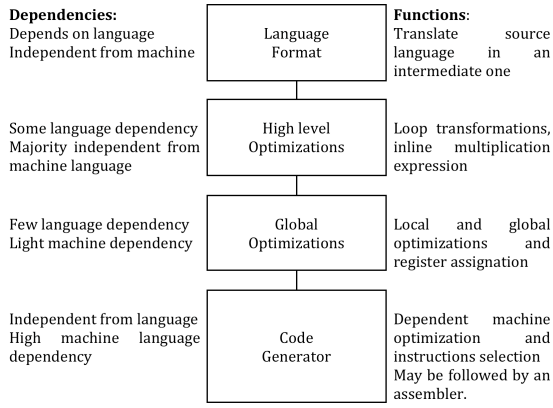
  - Variable Variable Variable Variable x + =

Polish inverse notation is used

- **I80x86 assembly sentences:**

```
MOV AL, Acceleration
MUL Time
ADD AL, InitialSpeed
MOV Speed, AL
```

Compilers (V). Compiler blocks dependency



Compilers (VI). Code optimizations

Optimization	Description
Inline procedures	To decide if a procedure is expanded inline or not
Common redundant sub expression removal	To substitute two or more copies of the same calculus by a temporal variable
Decrease size of the stack	To reorder operands to push in order of having less stack acceses
Code reordering	To remove a transfer instruction with the same value which may be inside a loop
Power decrease	To substitute multiplications by additions and divisions by subtractions
Segmentation planning	Source code reordering to improve performances and to reduce dependencies between instructions

- Performance can be improved between 60% and 80% by using global and local optimizations

Compilers (VII). Code optimizations

- **Inline procedures exameple:**

```
PROCEDURE EmptyTable (VAR T:
    Table)
BEGIN
    T.NumberOfItems := 0
END;
```

- **It's a very simple procedure to expand it**

With inline expansion:

```
    MOV T.NumberOfItems,0
```

Without inline expansion:

```
EmptyTablePROC
    MOV T.NumberOfItems, 0
    RET
EmptyTableENDP
                .
                .
                .
CALL EmptyTable
```


Compilers (VIII). Code optimizations

- **Common redundant expresions removal:**

```
                .
                .
Cost:= Quantity x PUnit;
Off:=
  Quantity x PUnit x Percent;
                .
                ..
Cost:= Quantity x PUnit;
                .
                .
```

```
                .
                .
Temp0 := Quantity x PUnit;
Off:= Temp0 x Percent;
                .
                .
                .
Temp0 := Quantity x PUnit;
                .
                .
```

Compilers (IX). Code optimizations

- **Decrease stack size**:

Energy := InitialEnergy+ Mass x Exp(Acceleration, 2);

```
Energy InitialEnergy Mass Acceleration Acceleration x x + =
```


Compilers (X). Code optimizations

- **Code reordering**:

```
MOV CX, 4
MOV SI, 0
Again:
    MOV AH, 2
    MOV DL, String[SI]
    INC SI
    LOOP Again
```

```
MOV CX, 4
MOV SI, 0
MOV AH, 2
Again:
    MOV DL, String[SI]
    INC SI
    LOOP Again
```

Compilers (XI). Code optimizations

- **Power decrease**

MOV AL, Pi
MUL Radius
MUL 2
MOV Perimetre, AL
	MOV AL, Pi
	MUL Radius
	SHL AL, 1
	MOV Perimetre, AL

Compilers (and XII). Code optimizations

- **Segmentation planning**:

ADD AL, BL
MOV CL, AL
XOR DX, DX
INC DL
	ADD AL, BL
	XOR DX, DX
	MOV CL, AL
	INC DL

Instruction sets examples (I). Alpha vs. i80x86

- **Alpha architecture: general specification**
  - 64 bits architecture
  - Parallel execution
  - Multiprocessor configuration
  - High speed clock
  - Non-oriented to a specific operating system
  - Non-oriented to a specific high level language
- **I80x86 architecture: general specification**
  - 16 bits architecture evolved to 32 bits one
  - Binary code compatibility
  - CISC instruction set
  - Difficulty in adapting to segmented execution
  - Difficulty in adapting to superscalar systems

Instruction sets examples (II). Alpha

- **Parameter 1: operands storage in cPU**
  - 32 integer general purpose registers
  - 32 floating point general purpose registers
- **Parameter 2: explicit operands per instruction**
  - 3 operands
- **Parameter 3: addressing modes**
  - Execution model: register-register
  - Alignment required
  - Little-endian by default. It's possible to change it to big-endian
  - Unique memory addressing mode: relative to register
  - One of the three operands may be an immediate datum

Instruction sets examples (III). Alpha

- **Parameter 4: operand types and sizes**
  - Process instruction work on 64 bits
  - Memory access instructions allow: byte, word (16), longword (32)
  - Sign extension required to work with fewer than 64 bits data
  - Allowed types are: unsigned, integer and IEEE and VAX floating point formats
- **Parameter 5: instructions set**
  - Memory access instructions
  - Control instructions
  - Process instructions
  - Floating point instructions
  - Miscellanea instructions: system calls, memory management, …
  - Multimedia instructions

Instruction sets examples (IV). i80x86

- **Parameter 1: operands storage in CPU**
  - 8 integer “quasi” general purpose registers
  - 8 floating point “quasi” general purpose registers with stack access
- **Parameter 2: explicit operands per instruction**
  - 2 operands. One of the is source and target of the instruction
- **Parameter 3: adrressing modes**
  - Execution model: register-memory
  - Alignment recommended but no requested
  - Little-endian
  - Addressing modes: immediate, relative to register, direct to memory, direct to register, indirect and implicit
  - Floating point operand are always in the stack

Instruction sets examples (and V). i80x86

- **Parameter 4: tipo y tamaño de los operandos**
  - Trabaja con tamaños de byte, palabra (16) y doble palabra (32)
  - El coprocesador matemático emplea además enteros de 64 y 80 bits y coma flotante de 32, 64 y 80 bits
  - Los tipos permitidos son entero con y sin signo y los formatos para coma flotante del estándar IEEE 754
- **Parameter 5: conjunto de instrucciones**
  - Instrucciones de transferencia de datos
  - Instrucciones de control
  - Instrucciones de proceso
  - Instrucciones de coma flotante
  - Instrucciones de manejo de cadenas
  - Instrucciones misceláneas: llamadas al sistema, gestión de memoria, …
  - Instrucciones multimedia: MMX, MMX2, SSE_{nn}, 3DNow, …

Bibliography

- Estructura y diseño de computadores
  David A. Patterson y John L. Hennessy. Reverté, 2000
  Capítulo 3
- Arquitectura de computadores. Un enfoque cuantitativo
  John L. Hennessy y David A. Patterson. Mc Graw Hill, 3ª ed, 2002
  Capítulos 3 y 4 y apéndices B, C, D y E
- Estructura de computadores.
  José Mª. Angulo. Paraninfo, 1996
  Capítulo 2 y 9