Lesson 2. Instruction set design

Contents

- Basic definitions
- High level languages influence on instruction set design
- Instructions set parameter to be taken into account
- Instruction format
- VLIW technology
- Binary compatibility
- Compilers
- Instruction set examples
- Bibliography
Lesson 2:
Instruction set design

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.

Lesson 2:
Instruction set design

Basic concepts (II). Programming

- **High level language program creation process**

  ![Diagram](image)

  **Source program (PASCAL)**

  ![Diagram](image)

  **Assembly program**

  ![Diagram](image)

  **Object module**

  ![Diagram](image)

  **Executable**

  ![Diagram](image)
Lesson 2: Instruction set design

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

```pascal
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;
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

Lesson 2: Instruction set design

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

```pascal
PROGRAM Principal (input, output);
VAR v: vector;
BEGIN
  LeerVector ( v );
  IntercambioDirecto ( v );
  MostrarVector ( v );
  writeln ('Programa finalizado')
END.
```

- Functions and procedures calls
- Operating systems service invocation
Lesson 2: Instruction set design

Instruction set parameters (I)

- What to be taken into account when designing instruction sets

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

Lesson 2: Instruction set design

Instruction set parameters (II)

Outside CPU storage alternatives

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

<table>
<thead>
<tr>
<th>Machine type</th>
<th>Advantages</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack machine</td>
<td>Useful to evaluate expression</td>
<td>Impossible random stack access</td>
</tr>
<tr>
<td></td>
<td>Good density code more memory instructions are sort</td>
<td>Difficulty for efficient code generation and implementation</td>
</tr>
<tr>
<td>Accumulator machine 1-operand machine</td>
<td>Minimizes internal state machine Instructions are sort</td>
<td>A lot of traffic between accumulator and memory</td>
</tr>
<tr>
<td>Register file machine</td>
<td>General efficient code generation model</td>
<td>Long instructions to name registers</td>
</tr>
</tbody>
</table>
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 store often used variables to reduce traffic between memory and CPU
- Compiler designers prefer non-dedicated registers
- The number of registers depend on how will be used by the compiler.
  - Required registers for expression evaluations
  - Required register for parameter passing
  - Required register for variables storage

Instruction set parameters (IV). Memory storage

- GPR classification:
  - ALU number of operands: 2 or 3
  - ALU memory operands from 0 to 3
Instruction set parameters (V).
Memory addressing

- How is a memory address interpreted? Byte, half word, word, double word
- There are two ways to store 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.

<table>
<thead>
<tr>
<th>Addresses</th>
<th>100h</th>
<th>101h</th>
<th>102h</th>
<th>103h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little endian</td>
<td>78</td>
<td>56</td>
<td>34</td>
<td>12</td>
</tr>
<tr>
<td>Big endian</td>
<td>12</td>
<td>34</td>
<td>56</td>
<td>78</td>
</tr>
</tbody>
</table>

Instruction set parameters (VI).
Memory addressing

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

A 32 bits memory allocation from 102h main memory address

<table>
<thead>
<tr>
<th>Word 100h</th>
<th>Byte 100h</th>
<th>Byte 101h</th>
<th>Byte 102h</th>
<th>Byte 103h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 104h</td>
<td>Byte 104h</td>
<td>Byte 105h</td>
<td>Byte 106h</td>
<td>Byte 107h</td>
</tr>
</tbody>
</table>

102 mod 4 = 2 \(
\rightarrow
\) word is not aligned \(
\rightarrow
\) two memory accesses are required to be read

A 32 bits memory allocation from 104h main memory address

<table>
<thead>
<tr>
<th>Word 100h</th>
<th>Byte 100h</th>
<th>Byte 101h</th>
<th>Byte 102h</th>
<th>Byte 103h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 104h</td>
<td>Byte 104h</td>
<td>Byte 105h</td>
<td>Byte 106h</td>
<td>Byte 107h</td>
</tr>
</tbody>
</table>

104 mod 4 = 0 \(
\rightarrow
\) word is aligned \(
\rightarrow
\) only one memory access is required to be read
Lesson 2: Instruction set design

Instruction set parameters (VII). Memory addressing

- A non-aligned memory access has several memory references
- Even non required alignment machines are faster in accessing aligned memory references
- Aligned and non-aligned different object sizes examples:

<table>
<thead>
<tr>
<th>Addressed object</th>
<th>Byte aligned</th>
<th>Non byte aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte (8 bits)</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
<td>Never</td>
</tr>
<tr>
<td>Half word (16 bits)</td>
<td>0, 2, 4, 6</td>
<td>1, 3, 5, 7</td>
</tr>
<tr>
<td>Word (32 bits)</td>
<td>0, 4</td>
<td>1, 2, 3, 5, 6, 7</td>
</tr>
<tr>
<td>Double word (64 bits)</td>
<td>0</td>
<td>1, 2, 3, 4, 5, 6, 7</td>
</tr>
</tbody>
</table>

Instruction set parameters (VIII). Addressing modes

- How an instruction can access to the operands
  - Inmediate: operand is inside the machine code instruction.
  - Direct to register: machine code instruction contains the name of the register.
  - Direct to memory: machine code instruction 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 itself but its memory address.
  - Implicit: 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 design

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 addressing modes, e.g. offsets

Lesson 2:

Instruction set design

Instruction set parameters (X). Addressing modes use frequency

- VAX addressing mode use frequency

![Addressing mode register chart]

- Immediate
- Direct to memory
- Relative to index register
- Relative to base register
- Indirect

<table>
<thead>
<tr>
<th>Mode</th>
<th>TaX</th>
<th>Spice</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>63</td>
<td>39</td>
<td>24</td>
</tr>
<tr>
<td>Direct</td>
<td>24</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Relative</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Indirect</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>
Lesson 2: Instruction set design

Instruction set parameters (XI). Addressing modes use frequency

- VAX immediate data use frequency

![Percentage of operations that use immediate data](chart.png)

Lesson 2: Instruction set design

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 sizes:
  - 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

![Operand size access distribution](chart.png)
Lesson 2: Instruction set design

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 to 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

Lesson 2: Instruction set design

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

![Program control instruction frequency](chart.png)

<table>
<thead>
<tr>
<th>CALL / return</th>
<th>Unconditional jump</th>
<th>Conditional jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>68</td>
<td>75</td>
<td>78</td>
</tr>
</tbody>
</table>

Computers Structure and Organization.
Automatic Department Graduate in Computer Sciences / Graduate in Computer Engineering
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 operation, e.g., carry, signed, ...
    - Advantage: only one bit is used
    - Drawbacks: superscalar dependence problems
  - **One general purpose register**: one register is set to 0 or to 1 depending 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 comparison frequency

![Different compare condition use frequency chart](image_url)

- LT / GE
- GT / LE
- EQ / NE

- TeX
- Spice
- GCC

Computers Structure and Organization.
Automatic Department
Graduate in Computer Sciences / Graduate in Computer Engineering
Lesson 2: Instruction set design

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

Lesson 2: Instruction set design

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 set design

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 contain:
  - 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.
Lesson 2: Instruction set design

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

Lesson 2: Instruction set design

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

Contents

- Basic definitions
- High level languages influence on instruction set design
- Instructions set parameter to be taken into account
- Instruction format
- VLIW technology
- Binary code compatibility
- Compilers
- Instruction set examples
- Bibliography
Lesson 2: Instruction set design

Contents

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

Lesson 2: Instruction set design

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 dependencies and reorder the source code in compilation time
- VLIW technology uses the second alternative
VLIW technology (II)

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

VLIW technology (and III)

The diagram shows the relationship between the instruction cache, fetch unit, multioperation instruction, and the execution units (EU) connected to the register file.
Lesson 2: Instruction set design

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 architecture and operating system.

- Using native compilers and source program files is the best alternative but it's not frequently possible.

Lesson 2: Instruction set design

Binary code compatibility (and II)

- Binary code translation techniques:
  - **Software interpreter.** A program reads instruction by instruction the program of the old architecture and translates 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

- Source
- Analysis
  - Scanner
  - Parser
- Synthesis
  - Code Generation preparation
- Symbols table
- Compilation
  - FOR tables
  - Strings
  - Constants
Lesson 2: Instruction set design

Compilers (III).
Compilation process example

- Pascal sentence to be compiled:
  - Speed := InitialSpeed + Acceleration \times Time;

Scanner output:
- Variable = Variable + Variable \times Variable Delimiter

Parser output:
\[
\begin{align*}
\langle \text{assign sentence} \rangle & \quad \langle \text{delimiter} \rangle \\
\langle \text{variable} \rangle & = \langle \text{variable} \rangle + \langle \text{variable} \rangle \times \langle \text{variable} \rangle
\end{align*}
\]

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
Lesson 2: Instruction set design

Compilers (V).
Compiler blocks dependency

<table>
<thead>
<tr>
<th>Dependencies:</th>
<th>Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depends on language</td>
<td>Language Format</td>
</tr>
<tr>
<td>Independent from machine</td>
<td>Translate source language in an intermediate one</td>
</tr>
<tr>
<td></td>
<td>High level Optimizations</td>
</tr>
<tr>
<td>Some language dependency</td>
<td>Loop transformations, inline multiplication expression</td>
</tr>
<tr>
<td>Majority independent from machine language</td>
<td></td>
</tr>
<tr>
<td>Few language dependency</td>
<td>Global Optimizations</td>
</tr>
<tr>
<td>Light machine dependency</td>
<td>Local and global optimizations and register assignment</td>
</tr>
<tr>
<td>Independent from language</td>
<td>Code Generator</td>
</tr>
<tr>
<td>High machine language</td>
<td>Dependent machine optimization and instructions selection May be followed by an assembler.</td>
</tr>
<tr>
<td>dependency</td>
<td></td>
</tr>
</tbody>
</table>

Computers Structure and Organization.
Automatic Department Graduate in Computer Sciences / Graduate in Computer Engineering

Lesson 2: Instruction set design

Compilers (VI).
Code optimizations

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

- Performance can be improved between 60% and 80% by using global and local optimizations
Lesson 2:

Instruction set design

Compilers (VII).
Code optimizations

- Inline procedures example:

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

- It's a very simple procedure to expand it

  Without inline expansion:

  EmptyTablePROC
  MOV T.NumberOfItems, 0
  RET
  EmptyTableENDP

  With inline expansion:

  MOV T.NumberOfItems, 0

  It's a very simple procedure to expand it.

Lesson 2:

Instruction set design

Compilers (VIII).
Code optimizations

- Common redundant expressions 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;
Lesson 2: Instruction set design

Compilers (IX).
Code optimizations

- Decrease stack size:

\[
\text{Energy} := \text{InitialEnergy} + \text{Mass} \times \exp(\text{Acceleration}, 2);
\]

Lesson 2: Instruction set design

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
```
Lesson 2: Instruction set design

Compilers (XI). Code optimizations

- Power decrease

\[
\begin{align*}
\text{MOV AL, Pi} & \\
\text{MUL Radius} & \\
\text{MUL 2} & \\
\text{MOV Perimetre, AL} & \\
\text{MOV AL, Pi} & \\
\text{MUL Radius} & \\
\text{SHL AL, 1} & \\
\text{MOV Perimetre, AL} & \\
\end{align*}
\]

Lesson 2: Instruction set design

Compilers (and XII). Code optimizations

- Segmentation planning:

\[
\begin{align*}
\text{ADD AL, BL} & \\
\text{MOV CL, AL} & \\
\text{XOR DX, DX} & \\
\text{INC DL} & \\
\text{ADD AL, BL} & \\
\text{XOR DX, DX} & \\
\text{MOV CL, AL} & \\
\text{INC DL} & \\
\end{align*}
\]
Lesson 2: Instruction set design

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

Lesson 2: Instruction set design

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
Lesson 2: Instruction set design

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
  - Miscellaneous instructions: system calls, memory management, ...
  - Multimedia instructions

Lesson 2: Instruction set design

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: addressing 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, 3DNow, ...

---

**Bibliography**

- Estructura y diseño de computadores
  David A. Patterson y John L. Hennessy. Reverté, 2000
  Capítulo 3

- Arquitectura de computadores. Un enfoque cuantitativo
  Capítulos 3 y 4 y apéndices B, C, D y E

- Estructura de computadores.
  José Mª. Angulo. Paraninfo, 1996
  Capítulo 2 y 9