Chapter 2

Instruction Set Architecture (ISA)

מטרת הפרק הזה היא הגדרת **ארכיטקטורה של קבוצת הפקודות** והבנת השיקולים שהדריכו תכנון מחשבים בין 1950 – 1990 ועדיין תקפים לגבי המחשב האישי. נסקור את האפשרויות שניתן לבחור מהן בהרכבת ISA והשיטות לממש אותן ב- microarchitecture.

Slide 2 **Overview of Chapter**

What is a processor?

- Von Neumann architecture
- Stages in the design of a processor

Instruction set

- Structure of instructions
- Operands and data
- Data storage and memory types
- Operations on data

Considerations in design of an instruction set

- Complex Instruction Set Computers (CISC)

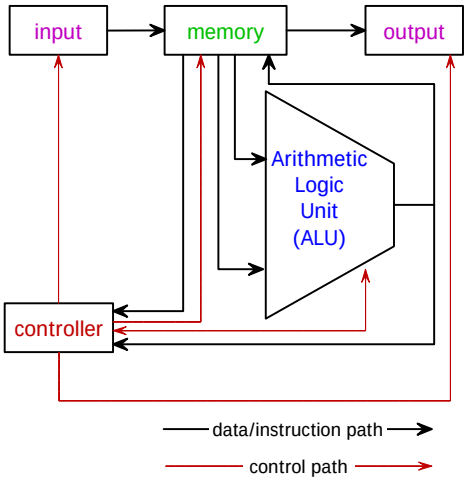
Implementing instructions in a microarchitecture

- Microcode

Slide 3 **Von Neumann Architecture**

In a 1947 paper, **John von Neumann** and others specified the features for an electronic digital computer:

- Digital computation in ALU
- Programmable via set of standard instructions
- Internal storage of data
- Internal storage of program
- Automatic Input/Output
- Automatic sequencing of instruction execution by decoder/controller



The activities in a digital are divided into a sequence of **instructions** — actions performed on data. Instructions move and manipulate data to produce new data according to a specific sequence.

Slide 4 Stages in Computer Design

Instruction Set Architecture (ISA)

The design of a computer begins with the specification of the ISA:

1. Look at the universe of problems to be solved and define the desired capabilities
2. Define a set of atomic operations at level of a system programmer (assembly language)
    - A set of small and orthogonal operations (each performs different task)
    - Instructions in the set can be combined to perform any desired operation
3. Specify the instruction set for the machine language
    - Choose a minimum set of basic operations from all the possibilities
    - Minimize the number of ways to solve the same problem

Implementation

1. Design the machine as a microarchitecture implementation of the ISA
2. Evaluate the machine's theoretical performance
3. Identify problem areas in the machine's performance
4. Improve processor efficiency by redefining operations

Slides 5 — 7 Instruction Set Architecture

Definitions

An **instruction** is a description of an **Operation** performed on **Operands**

An **Operation** is a specific action performed on data.

An **Operand** is a representation of data.

Source operands are the data **inputs** to an operation. **Destination** operands are the data **outputs** from an operation. Operands are specified by an **Addressing Mode** that determines the location of the data in the machine and by the **Data Type** that indicates whether the data is represented as an Integer, Long, Floating Point, Decimal, String, Constant, etc.

As an abstraction, a **general instruction** in an instance of the data structure

Operation	Operand	Operand	...	Operand

where the first field is taken from the set of legal (well-defined) actions on data and the remaining fields are instances of legal addressing modes.
A typical machine instruction has the form

```
ADD destination, source_1, source_2
```

which is interpreted to mean

```
destination ← source_1 + source_2
```

Two data operands are read from source operand locations and added. The sum is stored in the destination operand location.

General operations may act on any number of source operands.

A **unary** operation acts on one source operand.

A **binary** operation acts on two source operands.

An **n-ary** operation acts on n source operands.

An **address specifier** is a special field that describes the format of an operand. It may specify the addressing mode and the operation model (described on slides 13 – 14).

Various names are given to the width of an integer operand. In Intel documentation, an operand may be a byte, word (two bytes), dword (double word = 4 bytes), or quadword (8 bytes). In other architectures, a word is the standard integer length, 32 or 64 bits. **We will state the width of data operands explicitly**.

In slides 8 to 16 we define the basic aspects and features of an instruction set: **operands** (memory and registers), **operation models**, **addressing modes** and **operations**.

Slide 8 Memory Hierarchy

Memory is a basic feature of CPU operation. To maximize performance, memory is organized hierarchically into four levels.

Long-term storage (hard disk, DVD, flash drive, etc.) is least expensive (monetary cost per byte) with the longest access time (data read / write time). Hardware organization is complex, with most operations performed by the OS. This layer contains all stored data and programs.

Main memory (RAM) is more expensive with shorter access time. Each memory cell holds 1 byte of data and is addressed sequentially. This layer holds all data and instructions for currently running programs (except sections temporarily "swapped out" to disk storage by the OS paging system).

Cache is more expensive than RAM with shorter access time. Cache addressing is similar to RAM addressing — cache contains a copy of a small section of main memory. This layer holds data and instructions to be used in upcoming operations.

Registers are more expensive than cache with shorter access time. Addressing is by register name and defined in the ISA. This layer holds data and instructions to be used in the next few operations. Register widths are defined by the standard integer for the CPU.

In most modern CPUs, data is moved directly between the ALU and registers. The CPU loads data to registers from **cache** before ALU operations are performed.

Data is generally copied to cache from main memory as needed.

If a data location (the data contents identified by its address in main memory) is currently copied to Layer 1 cache (L1), that data can be copied to a register in one clock cycle. This condition is called a **cache hit**. When a required memory location is not currently in cache, it is called a **cache miss**.

The CPU stores values of intermediate results in **temporary registers** that cannot generally be accessed by the programmer. Registers that are directly visible to programs are called **architectural state**. **System state** consists of all resources visible to programs — architectural state and system memory.

When a system operation writes temporary values to system state, the write is called **commitment to state**.

Slide 9 Register Naming

The **registers** are part of the CPU design and are named in the design process. Information stored in registers is called **architectural state** and describes machine status and program status.

Registers are divided into **general purpose** and **special purpose**.

General Purpose (GP) registers hold data for instructions. The width of the data register is the width of the standard integer defined in the CPU architecture (usually 32 or 64 bits). Access to registers is by reference to names or numbers.

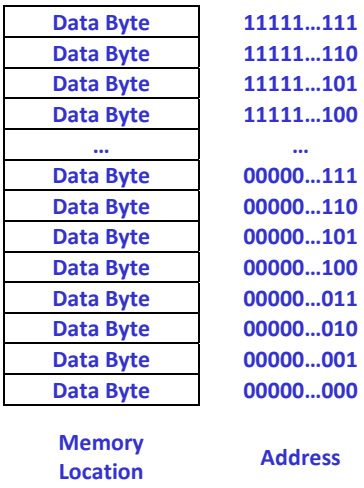
Intel x86 registers are named: **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP**

Registers in other ISAs are numbered: **R0, R1, … , R127**

Special Purpose (SP) registers include **machine status** registers and **Operating System** registers (reserved for use by the OS in supervisor mode).

Slides 10 – 11 Flat Memory Organization

Main memory is organized by an **N-bit physical address** $\mathbf{A = A_{N-1}\ A_{N-2}\ \ldots\ A_1\ A_0}$. The value of A runs from 0 to $2^N - 1$. Each address specifies the storage location of one byte of data. The CPU accesses data in main memory by sending the N-bit address of the first byte for the data. The CPU must contain an N-bit register to hold the physical address.



Since most integers are longer than one byte the ISA must specify the order in memory of the bytes that belong to the integer.

In a **little endian** ISA the least significant byte is stored at lowest address for the integer. The 32-bit integer **69 b3 36 7d** (in hexadecimal notation) is stored at address 0 as

stored byte					69	b3	36	7d
address	07	06	05	04	03	02	01	00

In a **big endian** ISA the most significant byte is stored at lowest address for the integer. The 32-bit integer **69 b3 36 7d** (in hexadecimal notation) is stored at address 0 as

stored byte					7d	36	b3	69
address	07	06	05	04	03	02	01	00

Intel x86 processors are little endian machines.

Slide 12 Specifying Operands

Access to operands is specified by **Addressing Modes**, which are formalized in the following rules:

An **immediate** value is specified as a literal (constant) coded into the instruction. It is referred to in an instruction definition as **`IMM`**.

A **register** value is specified by the name of the register that holds the value. It is referred to in an instruction definition as **`REGS[register name]`**.

A **memory** value is specified by an expression that evaluates to an address. It is referred to in an instruction definition as **`MEM[address]`**.

For example, the instruction **`ADDI reg1, reg2, #IMM`** can be specified as

`REGS[reg1] ← REGS[reg2] + IMM`

where **`reg1`** and **`reg2`** are registers defined in the ISA.

Pointer arithmetic is enabled by evaluating an expression. For example,

`LW reg1,IMM(reg2)`

is formalized as

`REGS[reg1] ← MEM[REGS[reg2] + IMM]`

where **`reg2`** holds a pointer to memory and the constant **`IMM`** is added to the pointer by the CPU before the memory access is performed. Slide 16 lists some common addressing modes.

Slides 13 – 14 Structured Operation Models

The operation model in an ISA is the system-level programming model. It specifies the type of ALU to be used in the implementation of defined instructions.

Stack

A stack-oriented ALU maintains a **stack pointer** and uses instructions that **auto-increment** or **auto-decrement** the pointer (add or subtract **`d`** = width of integer):

`Push`	**`Pointer ← Pointer – d`** **`Stack[Pointer] ← memory/register`**
`Pop`	**`memory/register ← Stack[Pointer]`** **`Pointer ← Pointer + d`**
`Binary Op`	**`Stack[Pointer + d] ← Stack[Pointer + d] Op Stack[Pointer]`** **`Pointer ← Pointer + d`**

The high level instruction **`Z = X + Y`** (where **`X, Y, Z`** are pointers to locations in main memory) is compiled to the sequence

```
push X
push Y
ADD        ; adds top two elements of stack to top of stack
pop Z
```

The stack programming model is used in Java byte code.

Accumulator

An accumulator-oriented ALU maintains a special register **A** which is one source and the destination of every ALU instruction. The high level instruction **Z = X + Y** is compiled to the sequence

```
load X    ; copies X to A
add Y     ; adds Y to A
store Z   ; copies A to Z
```

The accumulator programming model is used in hand held calculators.

Register-Memory Model

A Register-Memory ALU stores operands in general registers or in main memory. The high level instruction **Z = X + Y** is compiled to the sequence

```
load R1, X
add R1, R1, Y
store Z, R1
```

The Register-Memory programming model is used in the Intel x86 ISA.

Register-Register Model

A Register-Register ALU must load operands from main memory to general registers before an ALU operation. The high level instruction **Z = X + Y** is compiled to the sequence

```
load R1, X
load R2, Y
add R1, R1, R2
store Z, R1
```

The Register-Register programming model is less convenient for assembly language programming but permits faster implementation in hardware. It is used (at least in internal operations) in all modern CPUs.

Slide 15 **Typical Addressing Modes**

Addressing modes specify the location of an operand. Compilers use certain addressing modes as standard strategies to implement the programming models of high-level languages. Some addressing modes are:

Mode	Assembly Syntax	Operand Location Accessed	Use
Register	**R3**	**Regs[R3]**	Register data used in short-term ALU operations
Immediate	**#3**	**3**	Constant (literal) value. Encoded in instruction — cannot be changed at run time.
Direct (absolute)	**(1001)**	**Mem[1001]**	Static data — placed by OS at load time
Register deferred	**(R1)**	**Mem[Regs[R1]]**	Register R1 holds a pointer to a memory location
Displacement	**100(R1)**	**Mem[100+Regs[R1]]**	Local variables — R1 holds a pointer to the start of a local data frame and 100 is the offset to a named variable

Indexed	**(R1 + R2)**	**Mem[Regs[R1]+Regs[R2]]**	Array addressing — R1 points to the start of a data array and R2 holds the offset to an array element
Memory indirect	**@(R3)**	**Mem[Mem[Regs[R3]]]**	Pointer to pointer
Auto Increment	**(R2)+**	**Mem[Regs[R2]]** **Regs[R2] ← Regs[R2]+d**	Stack access (typically pop)
Auto Decrement	**-(R2)**	**Regs[R2] ← Regs[R2]-d** **Mem[Regs[R2]]**	Stack access (typically push)
Scaled	**100(R2)[R3]**	**Mem[100+Regs[R2]+Regs[R3]*d]**	Complex array indexing — R2 holds array base, 100 is an offset, and R3 is an index that is multiplied by the operand length **d**
PC-relative	**(PC)**	**Mem[PC+value]**	Store data relative to program counter (instruction address)
PC-relative deferred	**1001(PC)**	**Mem[PC+Mem[1001]]**	Store data relative to program counter (instruction address)

Slide 16 Typical Operations

An instruction set can define many types of operation on data, generally classified as:

Data transfer

Load (reg ← mem), store (mem ← reg), move (reg/mem ← reg/mem), convert data types

Arithmetic/Logical (ALU)

Integer arithmetic (+ – × ÷, compare shift) and logical (AND, OR, NOR, XOR)

Decimal

Integer arithmetic on decimal numbers

Floating point (FPU)

Floating point arithmetic (+ – × ÷, sqrt trig exp …)

String

String move, string compare, string search

Control

Conditional and unconditional branch, call/return, trap

Operating System

System calls, virtual memory management instructions

Graphics

Pixel operations, compression/decompression operations

Classic Computer Organization

In the previous section we saw examples of possible features for an instruction set. Given the various instruction formats, types of operands and addressing modes, possible programming models and instruction types, the next question is **what elements to chose and on what basis?**

In order to understand the choices made in contemporary CPUs, we will discuss the choices made historically in the order these strategies emerged. It will be seen that very few of these strategies have disappeared from modern instruction sets, and very little time will be wasted on "ancient history".

Slides 18 — 21 Considerations in Classic Computer Design

Before the mid-1970s all computers were large, expensive and typically owned by large businesses and institutions. By the late 1960s smaller computers were developed for special purposes. In the mid-1970s "minicomputers" were developed as general-purpose alternatives to large "mainframe" computers.

The first highly successful minicomputer was the VAX introduced in 1977 by the Digital Equipment Corporation (DEC). The VAX designers worked in a technical context that included:

Expensive memory

The wholesale price of RAM in 1977 was about $5000 per MB.

Poor compilers

Compilers were very simple with very limited error messaging and few optimization abilities. As a result, fast and efficient code was usually written, or optimized, in assembly language.

Semantic Gap Argument

The leading theoretical approach to programming language argued that an effective computer language must imitate natural language (spoken language). It should have a large vocabulary of operations and operands, and a high redundancy, meaning that it provides several different ways of programming the same task.

The result of these considerations was the development of powerful and **complex assembly languages**. The classic ISA defines many different types of instruction **syntax** with many **operations** and **addressing modes**. Although learning assembly language was a more difficult task, an experienced programmer could write efficient code easily, choosing the most appropriate methods from equivalent various alternatives. Because each instruction is **complex** and powerful (one instruction can perform many sub-operations), fewer instructions are necessary and program listings are shorter and occupy less memory.

An instruction set architecture designed under this approach is now called **CISC** (Complex Instruction Set Computer). A typical CISC ISA contains:

- More than 300 instruction types
- More than 15 addressing modes
- More than 10 data types
- Automated procedure handling — a single instruction to implement an function call
- Complex machine implementations — a consequence of the complexity of the instruction set. Each defined instruction must be implemented in dedicated hardware.

CISC machines were the conventional wisdom in the mainframe computers of the 1960s and 1970s. There was no other type of general-purpose computer and the term CISC did not yet exist (until there were alternatives in the 1990s). By 1980 all computers could be categorized as:

Mainframes

Mainframes are large and expensive computers, generally owned by big businesses and government agencies. In the 1980s the mainframe of an international bank occupied two entire floors in the World Trade Center. Some manufacturers in the 1970s were IBM, Control Data, Burrows, and Honeywell. Until the 1990s all mainframes were CISC machines.

Minicomputers

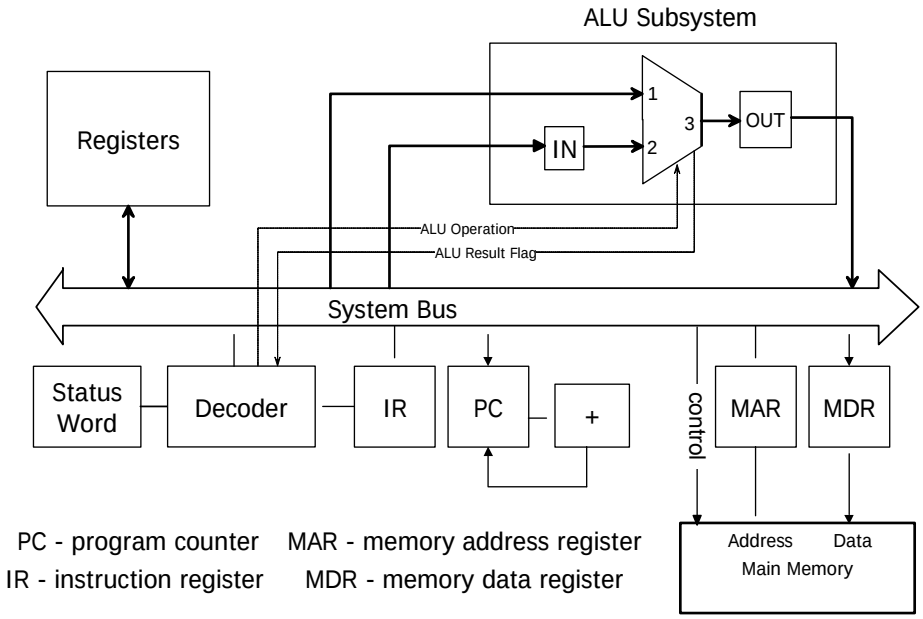
Minicomputers were smaller computers (about the size of a refrigerator) designed for smaller organizations. Unlike mainframes, they could typically run one OS at a time and serve up to about 30 users performing simple tasks. Two manufacturers were Digital (PDP/VAX) and Data General (Eclipse). Because a university department could own a minicomputer, this development promoted the emergence of academic computer science as a separate discipline (by mathematicians, physicists and electrical engineers). The smaller machines required smaller operating systems leading to the development of Unix. Because several small computers may be working on a single large task, it became important to connect them, leading to developments in computer networking such as TCP/IP.

Microcomputers

Microprocessors (a CPU on a single integrated circuit chip) were developed in the 1970s, based on the ISA of a minicomputer. Intel designed the 8086 and 8088 (1979) to operate like a tiny VAX. The Apple II personal computer and IBM's PC took advantage of these CISC type microprocessors. The Intel x86 family used in PCs and servers is the only CISC ISA still widely manufactured.

Slides 22 — 24 Physical Implementation

In order to implement the complex ISA of CISC, the microarchitecture was designed to be generic and easily expandable. Much like the workbench a medieval artisan's workshop, all work passes across the **System Bus** located at the center.



Attached to the System Bus are 5 subsystems:

Registers — the user accessible general-purpose registers defined in the ISA. These are generally numbered R_0 to R_{n-1} (for n registers).

ALU subsystem — the ALU with 2 temporary registers, IN and OUT.

The register IN can store a source operand. The other source operand is provided directly by the System Bus and must be held stable during the ALU operation.

The register OUT holds the ALU result until it can be moved to memory or a register.

Memory subsystem — 2 temporary registers, MAR and MDR

MAR is the memory data register. The MAR holds addresses sent to main memory (external to the CPU) for reading and writing.

MDR is the memory data register. The MDR holds data to be written to memory or data read from memory.

The **control** line specifies a read or write operation to the memory. On read data is transferred from the memory to the MDR. On write data is transferred to the memory from the MDR.

Program counter — the PC register holds the address of the next instruction to be fetched. After a fetch the PC is updated by adding the length of the fetched instruction.

Decoder subsystem — decoder, IR and status word

The **instruction register** (IR) holds the fetched instruction for the decoder.

The **status** word stores flags related to the results of ALU operations (zero, negative, overflow).

The **decoder** converts each instruction to a state machine sequence for performing the instruction.

The **state machine** provided by the decoder is a sequence of commands to the other subsystems. Each command tells the various subsystems to read from an input line or write to an output line. As shown on slide 24, this is accomplished through **Output Enable** (OE) and **Input Enable** (IE) connections on each device.

Device A is connected to Device B through two signal **amplifiers**. In order for data to transfer from Device A to Device B, the decoder will set the OE on Device A (enabling write) and the Input Enable on Device B (enabling read).

For Device 1 to transfer data to Device 2 but not Device 3, the decoder will set OE on Device 1, IE on Device 2 and leave all other IE and OE lines unset. This activates the output amplifier on Device 1, the input amplifier on Device 2, and leaves Device 3 electrically isolated from the system bus (so that it does not participate in the transfer).

A **decoder command** is a word of the type 100010000...00 where each bit sets or unsets the OE or IE of one device. A **machine language instruction** is interpreted (translated) by the decoder into a sequence of **decoder commands** to implement the instruction.

Slides 26 — 26 Instruction Fetch

Fetching an instruction requires a 4 step state machine controlled by the decoder. The steps are:

(1) MAR ← PC

The address of the instruction is transferred to the memory address register (MAR)

(2) READ

The instruction is transferred to the memory data register (MDR)

(3) IR ← MDR

The instruction address is transferred to the instruction register (IR) for the decoder

(4) PC ← PC + length(instruction)

The program counter is updated

These steps are detailed on slides 26 – 29.

Slides 30 — 39 Atomic Operations

The **fetched instruction** is stored in the **instruction register** (IR) and **decoded**. Decoding means translation from machine language to a sequence of atomic operations within the CPU. Each atomic operation includes write (OE) and reads (IE) controlled by the decoder.

As an example consider the machine instruction **`SUB R1, R2, 100(R3)`** defined in the ISA. The source operands are **R2** and **`100(R3)`**. The instruction is formally written:

`Regs[R1] ← Regs[R2] + Mem[ 100 + Regs[R3] ]`

and the sequence of atomic operations is:

`ALU_IN ← R3`	Copy **R3** to the temporary register **`IN`** in the ALU subsystem
`ALU ← 100`	Write **`100`** to the immediate input in the ALU subsystem
ADD	Perform **ADD** on **R3** and **`100`** in the ALU
`MAR ← OUT`	Copy the ALU result from the temporary register **OUT** to **MAR**
READ	Read the memory operand to **MDR**
`ALU_IN ← MDR`	Copy the memory operand from **MDR** to the **`IN`** register in the ALU
`ALU ← R2`	Write **R2** to the immediate input in the ALU subsystem
SUB	Perform **SUB** on **R2** and the memory operand in the ALU
`R1 ← OUT`	Copy the ALU result from the temporary register **OUT** to **R1**

These steps are detailed on slides 31 – 39.

Slides 40 — 42 Microcode

The sequence of atomic operations in the CPU is called a **microprogram** and is written in a syntax of primitives called **microcode**. The decoder **interprets** each machine instruction to a microprogram. The microcode sequence for each machine instruction is stored in the decoder in read only memory (ROM). This method was developed my Maurice V. Wilkes in 1951.

Each line of a microprogram is atomic — it must complete before the next line can begin. The primary reason for this requirement is that only one data value can be written on the system bus at one time.

The **clock cycle** for the CPU must be long enough so that the most complex microcode instruction can be completed in one clock cycle. Since each line of microcode executes in 1 clock cycle, the **number of clock cycles required to execute 1 machine language instruction** is just the number of lines of microcode plus the number of cycles to fetch the instruction.

For example, the instruction **`SUB R1, R2, 100(R3)`** shown above requires 4 CC to fetch and 9 CC to execute. Therefore, this instruction will execute in 13 clock cycles.

The Intel 8086 includes a special subsystem that prefetches instructions whenever the memory is not being used for data access. If the **SUB** instruction is prefetched then it will run in 9 CC instead of 13 CC, a significant optimization.

The **run time** for a program can now be calculated. The total number of clock cycles for the program is the sum of the required CC for each instruction. Many instructions can be divided into types that use system resources in the same way (for example **`ADD R1, R2, R3`** and **`SUB R1, R2, R3`**). So the total number of clock cycles is given by

$$\mathrm{CC}^{\text{program}} = \sum_{\substack{i=\text{instruction} \\ \text{types}}} \left(\text{Instructions of type } i\right) \times \left(\mathrm{CC}^{\text{Instruction of type } i}\right)$$

The **total program run time** is now just $\mathrm{CC}^{\text{program}}$ × seconds per CC = $\mathrm{CC}^{\text{program}}$ / clock rate.

Slide 43 CISC Creates Anti-CISC Revolution

The increased development of CISC-type minicomputers and microprocessor-based personal computers lead to the end of the CISC era. The first 32-bit minicomputer was the **Eclipse** introduced by Data General in 1974. Digital introduced the 32-bit **VAX** in 1977 and it became a major success in the market. Large institutions used the VAX to offload certain applications from the mainframe systems. Intel was still running the assembly line of their Jerusalem factory on VAX systems in the 1990s.

By 1990 minicomputers had turned into powerful servers and workstations, powering the development of UNIX as an operating system for small computers and TCP/IP to interconnect the growing number of machines. Computer Science emerged as separate academic discipline and students needed topics for projects, theses and dissertations. One area for academic research was the performance of small computers. The results were surprising.

Research on minicomputer performance showed that CISC machines use their resources inefficiently. As compilers improved, it turned out that most of the instruction types and addressing modes were never used in converting high level language to machine language. And because CISC machine were designed to be generic and complete they ran more slowly than necessary, carrying the weight of the need to support unnecessary features.