Chapter 4

From CISC to RISC

בפרק 2 ראינו שהמחשבים הקלאסיים שתכננו וייצרו בין 1950 עד 1990 היו מסוג CISC, ארכיטקטורות מורכבות עם מימושים מורכבים. בפרק 3 למדנו גישה לביצועי מעבד ושיטות לניתוח ביצועים. ממחקר על ביצועיי מחשבי CISC התברר שחלק מהיכולות של CISC לא בא לידי ביטוי בזמן ריצה ולכן המורכבות תרמה להאטה של המערכת ללא תמורה. בשנות ה- 1980 הציעו גישה חלופית בשם RISC על בסיס "חוק" אמדהל שמדגישה פעולות נפוצות על חשבון פעולות אחרות. בפרק הזה נציג תוצאות מחקר על מחשבי CISC כדי לבחור מרכיבים ל- ISA בסוג RISC — Reduced Instruction Set Computer.

Slides 2 – 3 **CISC Creates the Anti-CISC Revolution**

In 1977 the Digital Equipment Company (DEC) introduced the VAX, the first commercially successful 32-bit CISC minicomputer. Because minicomputers were much cheaper than large mainframes, serious computers became available to small organizations for the first time in the 1970s and 1980s. As CISC minicomputers were more widely used they opened new possibilities and led to new developments. The UNIX operating system was developed as a minicomputer operating system and TCP/IP protocols were developed to support networks of minicomputers. In the late 1970s computer science emerged as separate academic discipline, bringing together mathematicians, physicists and electrical engineers into new academic departments. CS students needed topics for final projects, theses and dissertations — some performed research on CISC performance.

The results of the performance studies on CISC machines were surprising. The complexity of the CISC ISA was driven by technical limitations, such as limited memory space and very simple compilers. But there was also a belief that a complex machine language, similar to natural language, would produce efficient code. Research revealed that most of the machine language instructions defined in the CISC ISA are never used. In order to implement the complex instruction set, CISC implementations give up speed in favor of generality. Thus, it was found that CISC machines run slowly in order to support unnecessary features.

In particular, CISC machines had three main **limitations**:

CISC instruction set requires microcode

CISC ISA contains many different instruction types and each instruction requires a somewhat different implementation. Also, many CISC operations are complex with many separate steps. These instructions require complex decoding and sequencing, which can best be implemented on a generic microarchitecture using microcode.

Central bus organization

The implementation of atomic microcode operations is most easily carried out using a central System Bus, which becomes a bottleneck in CPU operation. Because only one value can be transferred at a time on the bus, microcode operations must be performed sequentially. Therefore, the machine instructions made up of microcode operations are executed sequentially and each machine instruction executes in multiple clock cycles.

Memory access

The complexity of the CISC operations means that instructions are of non-uniform length, from 1 byte to 16 or more bytes per instruction. As a result, the process of instruction fetch is made complex — multiple clock cycles are required to fetch one instruction. Worse still, determining the number of bytes to load requires decoding of the first byte.

Slide 4 RISC "Philosophy"

By the mid-1980s, based on research that revealed the limitations and inefficiency of CISC, computer architects, including John **Hennessy** at Stanford and David **Patterson** at Berkeley, proposed a new approach called **RISC** (Reduced Instruction Set Computer). The basic idea is to apply Amdahl's "Law" to instruction set architecture — speed up the operations accounting for most of run time and ignore any performance degradation to other instructions, so that the overall run time will be improved.

The RISC proposal was based on significant technological developments since the VAX was introduced in the 1970s. The price of RAM fell from $5000 / MB in 1975 to $5 / MB in 1990 (today RAM can be ordered online for about $0.0035 / MB). As compilers became more powerful and efficient with extensive optimization procedures, and as UNIX, C, and TCP/IP made practical portable code widespread, only specialists programming critical path tasks used assembly language. Therefore, choices made by compilers became more important than the convenience of the assembly programmer.

The principal research result on **CISC performance** can be summarized as two rules of thumb:

About 90% of run time is accounted for by about 10% of the instructions in the VAX ISA

About 90% of the VAX instruction set accounts for about 10% of program run time

Beginning in 1984 the RISC model was proposed. The idea is to speed up the operations that account for most of run time and ignore performance degradation to other instructions that may come from the enhancements. The RISC ISA reduces the CISC ISA, keeping only the most important instructions. This means that CISC instructions not included in RISC will be implemented as multiple RISC instructions. The goal is to build a simpler hardware implementation that will permit faster execution.

Slides 5 – 6 RISC Microprocessors

Simpler ISA

The RISC ISA defines fewer machine instructions and all instructions are defined to be of the same length. This makes the task of microarchitecture design much simpler.

Simpler hardware design

The simpler ISA permits a microarchitecture without microcode — all instructions can be implemented in a similar way. As a result there is no dedicated system bus and the CPU can process several instructions at the same time. As a result the CPU has a lower CPI and can be driven at a higher clock speed. One instruction completes its execution on almost every clock cycle.

High level program compiled to RISC

Because some CISC instructions are implemented a several RISC instructions, programs have a larger IC. There are more machine instructions after compilation than in CISC. But the longer machine programs run more quickly than the same high level program on CISC.

All processors today use RISC technology

Pure RISC CPUs include the IBM Power, SPARC, MIPS, ARM, and others

RISC technology for CISC language includes Intel x86 (Pentium, Centrino, Core, Xeon)

Explicitly parallel RISC includes Intel IA-64 (Itanium) and IBM mainframe CPUs

A comparison of CISC and RISC features is shown in the table:

	CISC	RISC
Instruction Types	300	50
Addressing Modes	15	5
Data Types	10	2
Procedure Handling	Automated and complex	Must be hand coded
Implementations	Complex	Simple
Memory Organization	Complex	Simple

Analysis of x86 and SPEC programs on the VAX and MIPS RISC CPU lead to a significant **speedup**:

$$S = \frac{T}{T'} = \frac{CPI^{CISC} \times IC^{CISC} \times \tau^{CISC}}{CPI^{RISC} \times IC^{RISC} \times \tau^{RISC}} = \frac{CPI^{CISC}}{CPI^{RISC}} \times \frac{IC^{CISC}}{IC^{RISC}} \times \frac{\tau^{CISC}}{\tau^{RISC}} = (\approx 6) \times \left(\approx \frac{1}{2}\right) \times \frac{\tau^{CISC}}{\tau^{RISC}} \approx 3 \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

On average the compiled binary of a C program to RISC is about twice as large (IC) as for CISC. But the CISC CPU requires about 6 times as many clock cycles to execute a typical instruction as a RISC CPU, so the speedup is at least 3, even before raising the clock rate.

Clock rates were more easily raised in RISC implementations because the simpler design requires fewer and smaller digital circuits. In general, digital circuits consume power on transitions between 0 and 1 values, so that the faster the clock rate the higher the bit transition rate and thus the higher the power consumption (the so-called **thermal bound**). As circuits get physically larger it becomes more difficult to remove heat from the center of the CPU package and so smaller circuits can be more easily operated at higher clock rates. Companies like Intel and IBM that specialize in the solid state physics of silicon have succeeded in making individual circuit features so small (around 10^{-5} millimeter) that other factors have become limitations for large circuits.

Designing a RISC ISA

In the rest of this chapter we review research results on CISC machines in order to make intelligent choices for a RISC ISA.

Slide 8 **Considerations for a RISC ISA**

A RISC ISA should be:

Simple — no instruction should require more steps than others

Complete — the ISA should be able to perform any desired computation

Orthogonal — there should be only one way to encode any given computation

Design of the ISA requires the following choices

Computation model — register-register or register-memory

Range and type of operations — which operations to include from all CISC possibilities

Operands — which data types and sizes to include from all CISC possibilities

Addressing modes — which types and sizes of displacement parameters to include

Branches — which types (conditional, unconditional, call/return) and offset (jump length)

These choices can be made by studying their relative importance in the run time of CISC machines that allow many possible combinations of these features.

Slide 9 Instruction Types

In order to choose the instruction (operation) types for a RISC ISA, Hennessey and Patterson chose 5 programs from the SPEC CINT benchmark suite (1992 version) and compiled them for the Intel x86 CISC ISA. By counting the number of times each instruction ran (IC_i) and the number of clock cycles required to run each instruction (CPI_i) they found the relative proportion of run time for each instruction ($F_i = [CPI_i \times IC_i] / [CPI \times IC]$). The results appear in the table:

Instruction	Relative Proportion of Total Run Time
Load	22%
Conditional branch	20%
Compare	16%
Store	12%
Add	8%
And	6%
Sub	5%
Move reg-reg	4%
Call	1%
Return	1%
Other	5%
Total	**100%**

The main result is that the first 10 instructions (out of about 300 defined by the ISA) account for 95% of run time. A design that runs these instructions quickly will speed up 95% of run time.

The first 10 instructions do not form a complete set — additional operations are required, such as Boolean operations. Also, the call and return instructions (2% of run time) are complex and difficult to implement, so they will be eliminated. Any reasonable set of 30 to 50 simple instructions from the x86 ISA (excluding register-memory ALU instructions and call/return) will produce an efficient RISC ISA.

Slides 10 – 11 Addressing Modes

Slide 10 shows a graph of the frequency of use of addressing modes in three SPEC programs compiled for the VAX ISA. The numbers represent the relative proportion of instructions in the 3 programs that use a particular addressing mode. The programs are TeX (phototypesetting), spice (electronic circuit emulation) and gcc (GNU compiler).

The table on slide 11 summarizes the results. Of the 12 addressing modes available in the VAX ISA, 3 modes account for 99% (TeX), 75% (spice) and 90% (gcc) of all accesses to operands.

A RISC ISA can be restricted to these 3 addressing modes — **register deferred**, **immediate**, **displacement**. A further simplification can be obtained by implementing register deferred as a special case of displacement: implement `mem[ reg ]` as `mem[ reg + 0 ]`

Slide 12 Instruction Length

In order to achieve an efficient fetch and execute procedure, instructions should be of uniform length. This enables completion of instruction fetch in 1 clock cycle. It also simplifies instruction decoding because the instruction length is known and instruction fields are in the same place for most instruction types. Most RISC machines for servers and workstations (including 64-bit ISAs) use 32-bit instructions. Some special purpose RISC machines use longer instructions, and the Intel Itanium and some IBM mainframes use 128-bit instructions.

When the ISA defines uniform 32-bit instructions it is obvious that no single field in the instruction can be 32 bits long. This means that address displacements, immediate values, the length of branch jumps will be smaller than 32-bit, and we must **choose the size of the field** that specifies these values.

Slides 13 – 14 Length of Immediate Operand

An immediate value is a constant (literal) hard coded into the instructions of a program. Integer values represented by an n-bit field range in size from -2^{n-1} to $+2^{n-1} - 1$. But most constants are not large integers. Slide 13 shows a graph of the relative frequency of the **number of bits required** to represent actual immediate values used in TeX, spice and gcc compiled for VAX.

From the table on slide 14 it can be seen that **allocating 16 bits** in the 32-bit instruction for immediate operands covers 70% (TeX), 78% (spice) and 80% (gcc) of immediate values used.

Slides 15 – 16 Displacement Length

The displacement addressing mode **`mem[reg + displacement]`** requires a displacement value coded as an immediate in the instruction. The displacement generally represents an offset from the base of a local data frame and so most are not large integers. Slide 15 shows a graph of the relative frequency of the **number of bits required** to represent actual immediate values used in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 16 it can be seen that **allocating 16 bits** in the 32-bit instruction for displacements covers 100% of values used in INT and FP programs.

Slides 17 – 18 Branch Instructions

Slide 17 shows a graph of the relative use of branch instructions in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 18 it can be seen that the call and return instructions are used in more than 10% of branch instructions, typically for implementing function calls in high level languages. However these instructions are complex and difficult to implement in RISC architecture. The call instruction must store the return address and register values on a local stack and then jump to a target address. The return must recover the return address, jump to that address, recover register values and clean up the stack. In a RISC machine it is preferable to implement call and return in a different way.

Slide 18 also shows that the unconditional branch is used in fewer than 6% of branch instructions. But this instruction is required for completeness of the instruction set and must be included in the RISC ISA. Including conditional and unconditional branch instructions covers about 90% of branches used in INT and FP programs.

Slides 19 – 20 Branch Offset

In a CPU, the program counter (PC) is the register that holds the address of the next instruction to be fetched. Branch instructions change the PC as **`PC ← PC + offset`** and require an offset value coded as an immediate in the instruction. Slide 19 shows a graph of the relative frequency of the **number of bits required** to represent actual offset values used in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 20 it can be seen that **allocating 16 bits** in the 32-bit instruction for the offset covers 100% of values used in INT and FP programs.

Slide 21 Summary — RISC ISA By the Numbers

In this chapter we have seen research results on the actual usage at run time of the many complex features available in CISC ISA. Most of these features are rarely used. A RISC ISA is designed by including only the most widely used features from CISC and implementing the remaining features as combinations of the resulting reduced set. In summary, we expect a RISC ISA to include the following features:

Instruction Types

10 instructions account for 95% of run time in CISC machines. We may choose 30 – 50 of the most necessary and convenient instructions and operations for the RISC instruction set.

Addressing Modes

Three addressing modes — register, immediate and displacement account for over 75% of all operands accessed in CISC.

Instruction Length

The RISC ISA defines uniform 32-bit instructions.

Branch Instructions

Conditional and unconditional branch instructions account for over 75% of all branch instructions used in CISC.

Length of immediate values

It was seen that an immediate field of length 16-bits is adequate for over 70% of immediate operands and 100% of displacement and branch offset values.

An example of a RISC ISA is presented in the coming chapters.