University of Babylon, College of science for women
Dept. of Computer science

Computer Architecture

Second year

Dr. Salah Al-Obaidi

Lecture #2: Primary Microprocessor Architectures

Spring 2024

Contents

Contents **i**

2 Primary Microprocessor Architectures **13**

2.1 Complex Instruction Set Computing (CISC) 13

2.1.1 CISC Background 13

2.1.2 CISC Instruction Set Organization 13

2.1.3 CISC Hardware Implementation 14

2.1.4 CISC Applications 15

2.2 Reduced Instruction Set Computing (RISC) 16

2.2.1 RISC Background 16

2.2.2 Characteristics of Reduced Instruction Set Architectures 17

2.2.3 RISC Instruction Set Organization 18

2.2.4 RISC Hardware Implementation 19

2.2.5 RISC PIPELINING 19

2.2.6 Multiple Registers and Register Windowing 21

2.2.7 RISC Applications 22

2. Primary Microprocessor Architectures

2.1 Complex Instruction Set Computing (CISC)

2.1.1 CISC Background

Because software compilers were in their infancy during the 1970's, computer instruction sets evolved around the needs of the human programmer, not the software. Ease of programming motivated rich instruction set development, and microcode updates enabled quick updates for new instructions. In these early years, memory capacity and price were at a premium, and program runtime footprints kept understandably small. This required that instruction sets be highly encoded, thus minimizing processor-memory bandwidth requirements. These restrictions led to the development of complex encoded instruction sets interpreted by microcode engines. The term CISC encompasses this approach. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance.

2.1.2 CISC Instruction Set Organization

In the early days of the Intel x86 series, designers attempted to simplify and reduce the "semantic gap", which is the difference between the operations provided in **high-level languages (HLLs)** and those provided in computer architecture, between software and CPU instruction sets. Symptoms of this gap are alleged to include execution inefficiency, excessive machine program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key features include large instruction sets, dozens of addressing modes, and various HLLs statements implemented in hardware.

This resulted in a large, richly typed instruction set to lower software development costs. To facilitate reuse and market proliferation, commercial CPU hardware required compatibility with existing computers of the same family. Hence, the creation of supersets and expansion of features to ensure hardware portability and accommodate high-level programming languages such as C, C++, and Java. Such complex instruction sets are intended to:

1. Ease the task of the compiler writer.
2. Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
3. Provide support for even more complex and sophisticated HLLs.

As a result, the mix of instruction types for a CISC machine can number in the hundreds. CISC instructions tend to vary in width (8/16/32/64-bit) and can specify individual or complete sequences of operations. As forward and backward compatibility requirements dictate, developers simply add more instructions to the CISC microcode ROM. See Table 2.1 below for an overview of typical CISC instructions.

Table 2.1: Typical CISC Instructions

Operator type	**Examples**
Arithmetic and logic	Integer arithmetic and logical operations: ADD, SUB, AND, OR
Data transfer	Load/store (move data:register/external memory, addressing modes)
Control	Branch, jump, procedure calls, taps
System	Operating system call, virtual memory management
Floating point	Floating point operations: FADD, FMULT, FDIV
Decimal	ADD, MULT, decimal → char conversations
String	String move, compare, search

2.1.3 CISC Hardware Implementation

At power-up, a CISC CPU accesses the microcode ROM, executing a bootstrap sequence of instructions to “enable” the processor into a startup state. During normal operation, the CPU microsequencer accesses microcode ROM as needed to execute instructions and active the appropriate control signals in a logical sequence. Due to the large number of clock states, a CISC instruction set must support microprogrammed control to implement the control unit logic as opposed to hardwired (hardware implemented) control. In most cases, hardware changes are not required as more instructions expand the microcode. In operation, the microcode program is stored inside an internal lookup ROM.

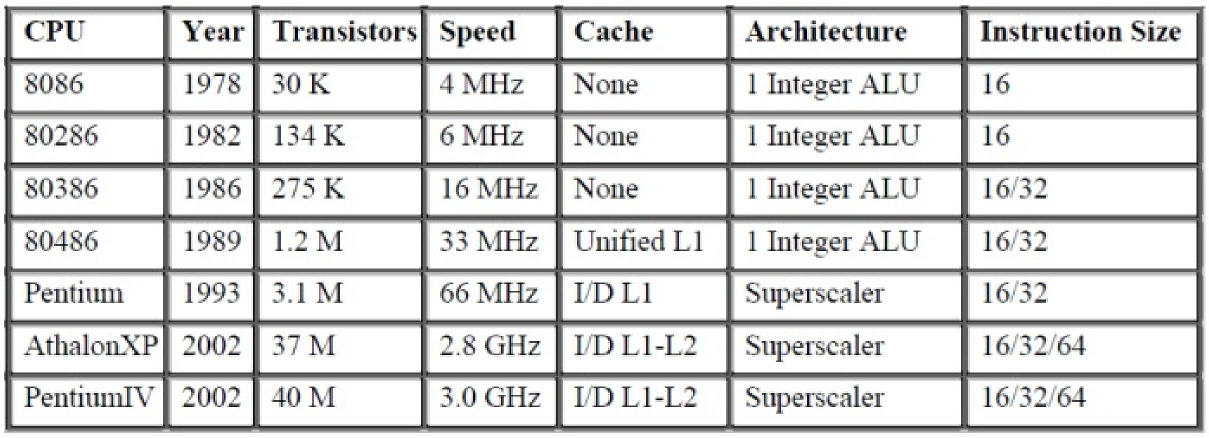
Because of shrinking transistor geometries, CISC developers were quickly able to boost performance by adding hardware resources. For example, the Intel 80386 was the first commercially available CISC to offer a complete 32-bit architecture. The 80386 included a 32-bit instruction set, a 32-bit data bus, and a 32-bit virtual address space while still maintaining object code compatibility with prior 16-bit Intel x86 processors.

In addition, the 80386 and follow-on CISC processors expanded integer data and address storage by simply adding more internal registers. Because a CISC microsequencer directly accesses local registers, memory I/O operations decrease, thus improving instruction execution efficiency. The 80386 was the first CISC to include an internal MMU (Memory Management Unit) supporting segmentation, paging, and protection through the full 32-bit virtual address space.

Beyond the 80386, the Intel Pentium series of processors built upon the earlier x86 designs, significantly enhancing the architecture's performance by adding internal instruction and data caches, hardware branch prediction, and superscaler execution units. Superscaler CPUs fetch, decode, and execute multiple instructions during each clock cycle. Multiple instruction execution is made possible by implementing deep parallel instruction pipelines with multiple (ALU) Arithmetic Logic Units. Figure 2.1 shows the Key CISC Microprocessors.

2.1.4 CISC Applications

One does not require statistics to see the impact of CISC in general, and the Intel x86 architecture in particular on the computing world. In 1994, the early Pentium designs offered little advantage over their RISC contemporaries. Not to be left behind, Intel's aggressive pricing, marketing, and rapid increase in clock speed enabled Pentium derivatives to dominate the desktop market by 1995, and the notebook market by 1996.



CPU	Year	Transistors	Speed	Cache	Architecture	Instruction Size
8086	1978	30 K	4 MHz	None	1 Integer ALU	16
80286	1982	134 K	6 MHz	None	1 Integer ALU	16
80386	1986	275 K	16 MHz	None	1 Integer ALU	16/32
80486	1989	1.2 M	33 MHz	Unified L1	1 Integer ALU	16/32
Pentium	1993	3.1 M	66 MHz	I/D L1	Superscaler	16/32
AthalonXP	2002	37 M	2.8 GHz	I/D L1-L2	Superscaler	16/32/64
PentiumIV	2002	40 M	3.0 GHz	I/D L1-L2	Superscaler	16/32/64

Figure 2.1: Key CISC Microprocessors.

By adding incremental superscaler features and improving the silicon process technology, Intel expanded the original Pentium design into the Pentium II-IV series and beyond.

Intel is not the only manufacturer of x86 CISC CPUs. AMD leveraged the x86 architecture with the K5 – K9 series, the K7 being the first commercially available microprocessor to hit 1GHz.

2.2 Reduced Instruction Set Computing (RISC)

2.2.1 RISC Background

During the late 1970's, IBM began project 801 to create a load/store oriented instruction set architecture that would be two-five times faster than other CPU architectures. In 1980, David A. Patterson at UC Berkeley began a similar project and built two machines called "RISC-I" and "RISC-II". The Berkeley architecture included simple load/store instructions and a new concept termed "register windowing". This means the use of a large set of registers should decrease the need to access memory.

On a similar quest, John L. Hennessy at Stanford University published a description of an efficient pipelining and compiler-assisted scheduling machine termed the "MIPS" architecture. It was hoped that the new RISC architectures would have a lower Clock Per Instruction (CPI) rate than existing CISC architectures such as the Intel 80XXX and DEC VAX-11/780 architectures. This meant that **RISC machines could execute more instructions within the same clock rate**.

The initial premise of RISC vs. CISC was that 90% of clock cycles execute only 10% of the instruction set. To realize a high-performance single-chip CPU, the design would need to be simple, with just enough logic to perform basic tasks such as load-store of internal/external locations and arithmetic operations. With instruction sets growing more complex to support high-level compilers, CISC CPUs required more logic to implement extra functionality—thereby increasing transistor counts and die sizes. Larger die sizes translated to higher development/production costs and frequencies.

Patterson and Ditzel concluded that instructions added for a given compiler generally were useless for other compilers. They also discovered that replacing complex instructions with a small number of lower-level instructions yielded minimal loss in performance. Furthermore, compiler development for a CISC instruction set was more time-consuming and bug-prone than compiler development for the simpler RISC architecture.

2.2.2 Characteristics of Reduced Instruction Set Architectures

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them:

1. One instruction per cycle.
2. Register-to-register operations.
3. Simple addressing modes.
4. Simple instruction formats.

The first characteristic listed is that there is **one machine instruction per machine cycle**. A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register. Thus, RISC machine instructions should be no more complicated than, and execute about as fast as, microinstructions on CISC machines. Such instructions should execute faster than comparable machine instructions on other machines.

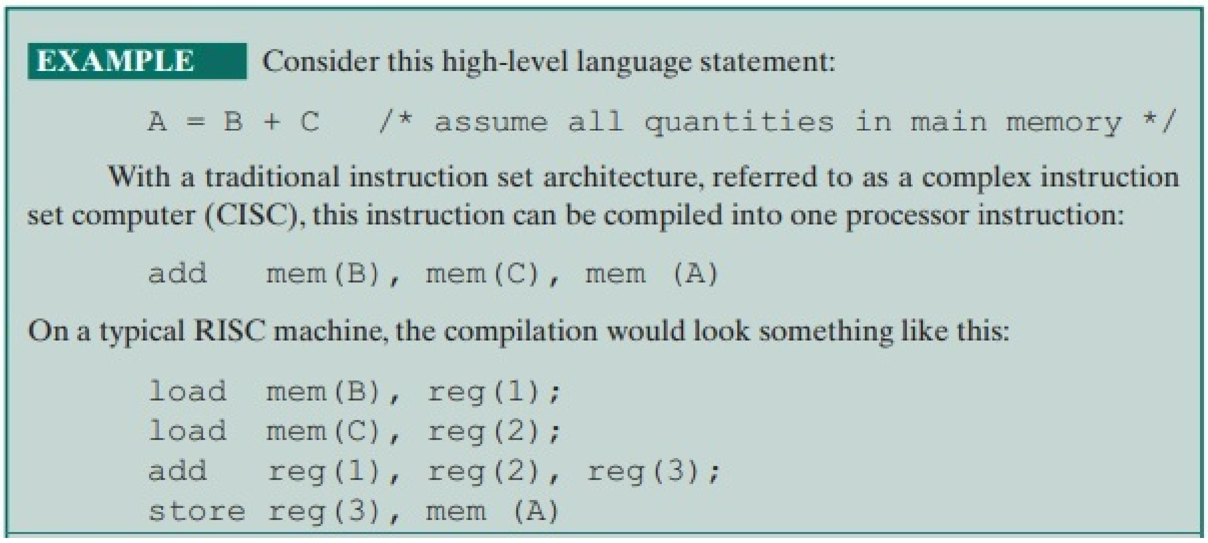
A second characteristic is that most operations should be **register to register**, with only simple LOAD and STORE operations accessing memory. This design feature simplifies the instruction set and therefore the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); the VAX has 25 different ADD instructions. Another benefit is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

A third characteristic is the use of **simple addressing modes**. Almost all RISC instructions use simple register addressing. This design feature simplifies the instruction set and the control unit.

A final common characteristic is the use of **simple instruction formats**. Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

2.2.3 RISC Instruction Set Organization

RISC microprocessors implement simple operations with fixed-width instructions (32/64). In general, most RISC processors include fewer than 100 instructions and execute at least one instruction per primary clock cycle. Even though the instruction set is small, RISC includes basic operations to accomplish system tasks. These tasks include data movement (internal/external load/store, internal register data movement), arithmetic, logic, shift operations, and simple branch instructions. Today, the RISC/CISC barrier is blurring, and many processors use hardware implementations derived from both classes of instruction sets.



EXAMPLE Consider this high-level language statement:

```
A = B + C   /* assume all quantities in main memory */
```

With a traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

```
add   mem(B), mem(C), mem (A)
```

On a typical RISC machine, the compilation would look something like this:

```
load  mem(B), reg(1);
load  mem(C), reg(2);
add   reg(1), reg(2), reg(3);
store reg(3), mem (A)
```

2.2.4 RISC Hardware Implementation

Because a smaller instruction set yields simpler logic requirements, RISC uses hardwired control logic as opposed to the more complex CISC microcode implementation. Due to the reduced logic complexity, RISC processors can incorporate several performance enhancing concepts such as pipelining, multiple internal registers, and register windowing while still maintaining relatively low transistor counts.

2.2.5 RISC PIPELINING

Instruction pipelining is often used to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the following two stages:

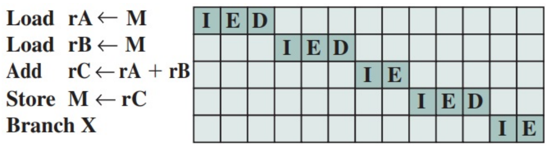
- I: Instruction fetch.
- E: Execute. Performs an ALU operation with register input and output.

For load and store operations, three stages are required:

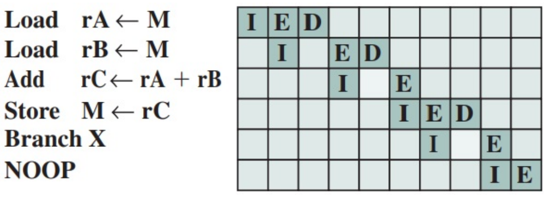
- I: Instruction fetch.
- E: Execute. Calculates memory address.

- D: Memory. Register-to-memory or memory-to-register operation.

Figure 2.2a depicts the timing of a sequence of instructions using no pipelining. Clearly, this is a wasteful process. Even very simple pipelining can substantially improve performance. Figure 2.2b shows a two-stage pipelining scheme, in which the I and E stages of two different instructions are performed simultaneously. The two stages of the pipeline are an instruction fetch stage, and an execute/memory stage that executes the instruction, including register-to-memory and memory-to-register operations. Thus we see that the instruction fetch stage of the second instruction can be performed in parallel with the first part of the execute/ memory stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This scheme can yield up to twice the execution rate of a serial scheme. Two problems prevent the maximum speedup from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per stage. This requires the insertion of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler.

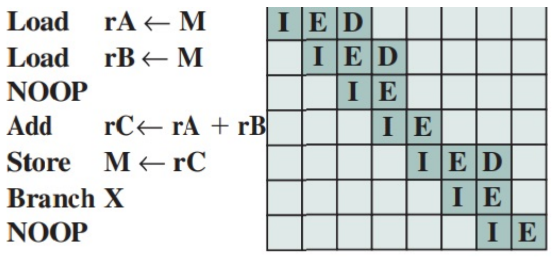


(a) Sequential execution

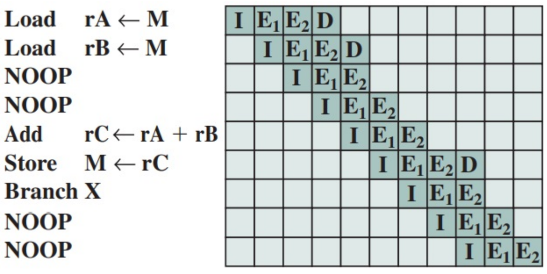


(b) Two-stage pipelined timing

1



(c) Three-stage pipelined timing



(d) Four-stage pipelined timing

Figure 2.2: The Effects of Pipelining

Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 2.2c. Now, up to three instructions can be overlapped, and the improvement is as much as a factor of 3. Again, branch instructions cause the speedup to fall short of the maximum possible. Also, note that data dependencies have an effect. If an instruction needs an operand that is altered by the preceding instruction, a delay is required. Again, this can be accomplished by a NOOP.

The pipelining discussed so far works best if the three stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two substages:

- E1: Register file read
- E2: ALU operation and register write

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. Figure 2.2c shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

2.2.6 Multiple Registers and Register Windowing

Most RISC and CISC designs use large numbers of general and special purpose registers (GPR/SPR) to store intermediate values. Extra registers enable fast data access since the CPU does not require external load/store to access required information. Reducing external memory referencing improves performance for applications requiring frequent calls to/from subroutines.

To boost performance, the SPARC (Scalable Processor ARChitecture) CPU from Sun Microsystems uses "**register windowing**" to pass parameters between subroutines. This concept utilizes *fully accessible global registers* and *window pointer/window mask registers.* **Window pointer registers** contain the address of active registers while **window mask registers** contain a bit flagging all registers containing valid data. As a result, the SPARC processor can pass parameters to subroutines through registers that overlap windows.

Without this feature, parameters would be passed through external memory, consuming precious CPU and I/O cycles.

2.2.7 RISC Applications

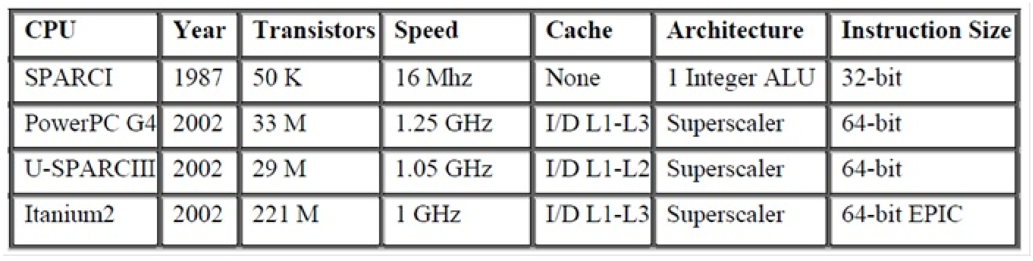
The growth of RISC in the commercial market began in the late 1980's with the SPARC CPU from Sun Microsystems. Prior to this, Sun relied upon CISC architectures such as the Motorola 680x0 series of CPU's to power Sun's UNIX workstations. The UNIX operating system enabled multi-user/multi-tasking, as opposed to the single-tasked limitation of the original Microsoft Windows platforms.

Sun's quest for higher performance and lower CPU costs fueled initial SPARC development. Within a few years, SPARC replaced Sun's CISC-based machines. During this period, RISC-based UNIX workstations became the platform of choice for EDA (Electronic Design Automation) software.

Contemporary EDA tasks such as RTL simulation, RTL-gate synthesis, cell APR typically ran on HP/PA-RISC, MIPS RISC, and SPARC-based platforms. The raw compute power of RISC vs. contemporary CISC machines (80486 DOS/Windows PC's) allowed RISC to dominate the scientific and engineering world until the mid-1990's.

Following on the heals of Sun, the joint IBM/Motorola/Apple development of PowerPC RISC microprocessors from 1992 onward allowed Apple Computer to replace its line of 680x0 CISC machines. Apple's PowerPC machines target contemporary desktop users and professionals requiring high performance for tasks such as video editing, graphics design, and 3D-gaming applications. PowerPC established Apple as the only manufacturer of RISC-based systems with a measurable share of the desktop market.

Figure 2.3 shows the Key RISC Microprocessors.



CPU	Year	Transistors	Speed	Cache	Architecture	Instruction Size
SPARCI	1987	50 K	16 Mhz	None	1 Integer ALU	32-bit
PowerPC G4	2002	33 M	1.25 GHz	I/D L1-L3	Superscaler	64-bit
U-SPARCIII	2002	29 M	1.05 GHz	I/D L1-L2	Superscaler	64-bit
Itanium2	2002	221 M	1 GHz	I/D L1-L3	Superscaler	64-bit EPIC

Figure 2.3: Key RISC Microprocessors.