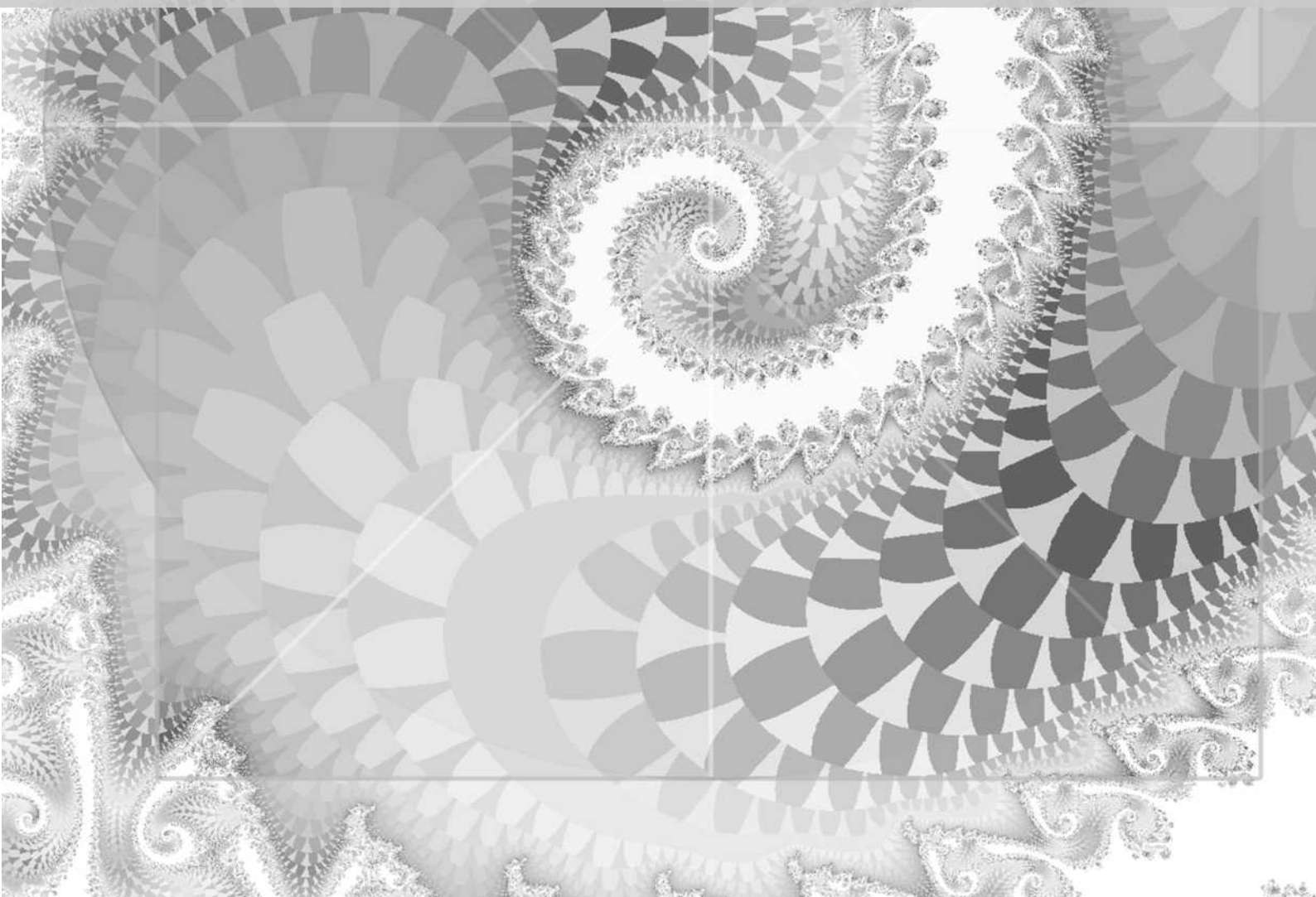
University of Babylon, College of science for women
Dept. of Computer science

Computer Architecture

Second year

Dr. Salah Al-Obaidi

Lecture #3: Instruction Set Architecture Spring 2024



Contents

3. Instruction Set Architecture

Instruction set architecture (ISA) is the portion of the computer that is visible to the programmer or compiler writer. The ISA serves as the boundary between the software and hardware (see Figure 3.1).

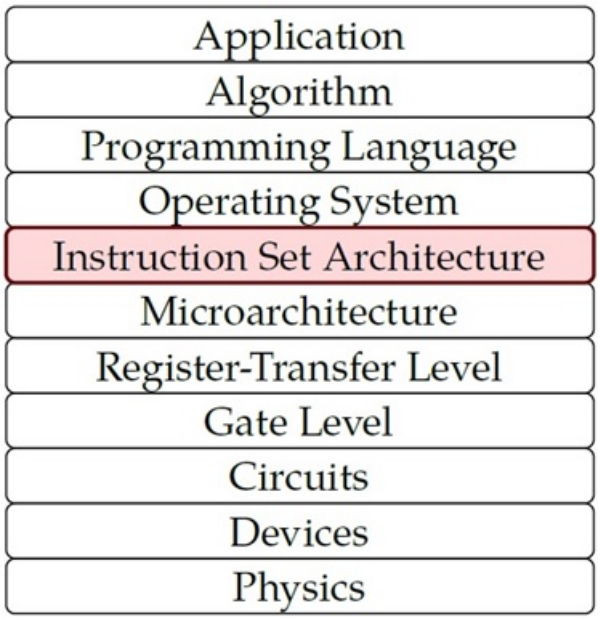


Figure 3.1: the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

This quick review of ISA will use examples from 80x86, ARM, and MIPS to illustrate the seven dimensions of an ISA.

1. **Class of ISA**: Nearly, all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating-point data,

while MIPS has 32 general-purpose and 32 floating-point registers. The two popular versions of this class are register-memory ISAs such as the 80x86, which can access memory as part of many instructions, and load-store ISAs such as MIPS, which can access memory only with load or store instructions. All recent ISAs are load-store.

2. **Memory addressing**: Virtually all desktop and server computers, including the 80x86 and MIPS, use byte addressing to access memory operands. Some architectures, like ARM and MIPS, require that objects must be aligned. An access to an object of size $\boldsymbol{s}$ bytes at byte address $\mathbf{A}$ is aligned if $\boldsymbol{A} \mod \boldsymbol{s} = \mathbf{0}$. The 80x86 does not require alignment, but accesses are generally faster if operands are aligned.

3. *Addressing modes*: In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), two registers where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement). It has more like the last three, minus the displacement field: register indirect, indexed, and based with scaled index.

4. *Types and sizes of operands*: Like most ISAs, MIPS and 80x86 support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).

5. *Operations*: The general categories of operations are data transfer, arithmetic logical, control, and floating point. MIPS is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures being used in 2006. Figure 3.2 summarizes the MIPS ISA. The 80x86 has a much richer and larger set of operations.

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LWU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits to/from FP registers from/to integer registers
Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract, signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
Floating point	*FP operations on DP and SP formats*
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers, and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D, C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Figure 3.2: **Subset of the instructions in MIPS64.** SP = single precision; DP = double precision.

6. *Control flow instructions*: Virtually all ISAs, including 80x86 and MIPS, support conditional branches, unconditional jumps, procedure calls, and returns. Both use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. There are some small differences. MIPS conditional branches (BE, BNE, etc.) test the contents of registers, while the 80x86 branches (JE, JNE, etc.) test condition code bits set as side effects of arithmetic/logic operations. MIPS procedure call (JAL) places the return address in a register, while the 80x86 call (CALLF) places the return address on a stack in memory.

7. *Encoding an ISA*: There are two basic choices on encoding: fixed length and variable length. All MIPS instructions are 32 bits long, which simplifies instruction decoding. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable length instructions can take less space than fixed-length instructions, so a program compiled for the 80x86 is usually smaller than the same program compiled for MIPS. Note that choices mentioned above will affect how the instructions are encoded into a binary representation. For example, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field can appear many times in a single instruction.

3.1 Classifying Instruction Set Architectures

The type of internal storage in a processor is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture. The major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack, and in an *accumulator architecture* one operand is implicitly the accumulator. The *general-purpose register* architectures have only explicit operands—either registers or memory locations. Figure 3.3 shows a block diagram of such architectures, and Figure 3.4 shows how the code sequence **C=A+B** would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may

need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called ***register-memory*** architecture, and the other can access memory only with load and store instructions, called ***load-store*** architecture. A third class, not found in computers shipping today, keeps all operands in memory and is called a ***memory-memory*** architecture. Some instruction set architectures have more registers than a single accumulator, but place restrictions on uses of these special registers. Such an architecture is sometimes called an ***extended accumulator*** or ***special-purpose register*** computer.

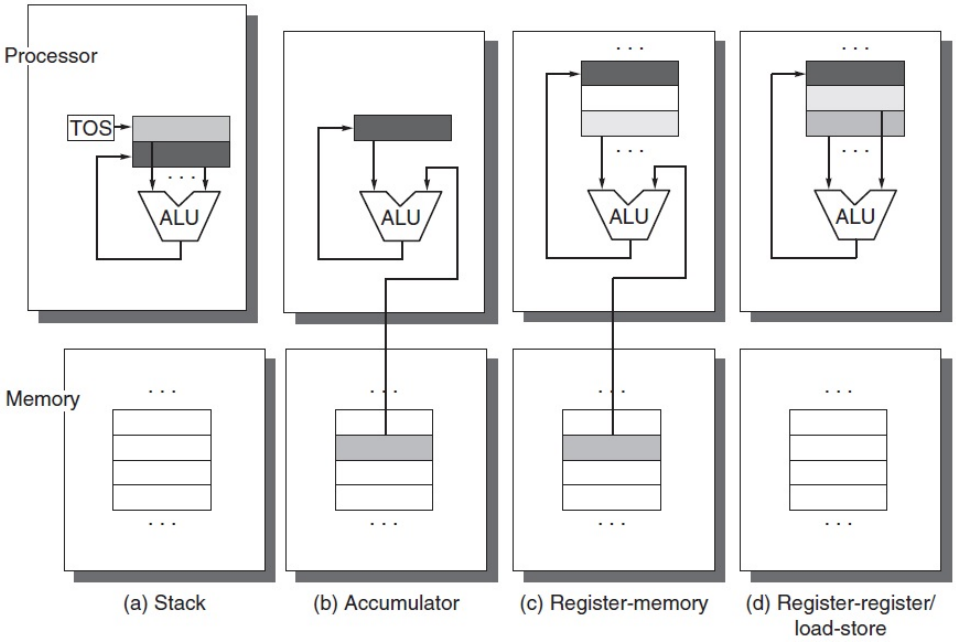


Figure 3.3: **Operand locations for four instruction set architecture classes**. The arrows indicate whether the operand is an input or the result of the ALU operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS), points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure 3.4: **The code sequence for C= A+B for four classes of instruction sets.** Note that the Add instruction has implicit operands for stack and accumulator architectures, and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure 3.3 shows the Add operation for each class of architecture.

Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. **First**, registers—like other forms of storage internal to the processor—are faster than memory. **Second**, registers are more efficient for a compiler to use than other forms of internal storage. For example, on a register computer the expression (A*B) – (B*C) – (A*D) may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns. Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times.

How many registers are sufficient? The answer, of course, depends on the effectiveness of the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. Modern compiler technology and its ability to effectively use larger number of registers has led to an increase in register counts in more recent architectures.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). **The first** concerns whether an ALU instruction has two or three operands.

In the three-operand format, the instruction contains one result operand and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. **The second** distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Figure 3.5 shows combinations of these two attributes with examples of computers. Although there are seven possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are **load-store** (also called register-register), **register-memory**, and **memory-memory**.

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure 3.5: **Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers.** Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

Figure 3.6 shows the advantages and disadvantages of each of these alternatives. Of course, these advantages and disadvantages are not absolutes: They are qualitative and their actual impact depends on the compiler and implementation strategy. A GPR computer with memory-memory operations could easily be ignored by the compiler and used as a load-store computer. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task.

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure 3.6: **Advantages and disadvantages of the three most common types of general-purpose register computers.** The notation (m, n) means m memory operands and n total operands. In general, computers with fewer alternatives simplify the compiler's task since there are fewer decisions for the compiler to make. Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size since you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes 3 extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

3.2 The Evolution of the Intel x86 Architecture

We rely on many concrete examples of computer design and implementation to illustrate concepts and to illuminate trade-offs. Numerous systems, both contemporary and historical, provide examples of important computer architecture design features. We relies principally on examples from two processor families: **the Intel x86** and **the ARM** architectures. The current x86 offerings represent the results of decades of design effort on **complex instruction set computers (CISCs)**. The x86 is considered an excellent example of CISC design. An alternative approach to processor design is the **reduced instruction set computer (RISC)**. The ARM architecture is used in a wide variety of embedded systems and is one of the most powerful and best-designed RISC-based systems on the market.

Interestingly, as microprocessors have grown faster and much more complex, Intel has actually picked up the pace. Intel used to develop microprocessors one after another,

every four years. But Intel hopes to keep rivals at bay by trimming a year or two off this development time, and has done so with the most recent x86 generations.

It is worthwhile to list some of the highlights of the evolution of the Intel product line:

- **8080:** The world's first general- purpose microprocessor. This was an 8-bit machine, with an 8-bit data path to memory. The 8080 was used in the first personal computer, the Altair.

- **8086:** A far more powerful, 16-bit machine. In addition to a wider data path and larger registers, the 8086 sported an instruction cache, or queue, that prefetches a few instructions before they are executed. A variant of this processor, the 8088, was used in IBM's first personal computer, securing the success of Intel. The 8086 is the first appearance of the x86 architecture.

- **80286:** This extension of the 8086 enabled addressing a 16-MB memory instead of just 1 MB.

- **80386:** Intel's first 32-bit machine, and a major overhaul of the product. With a 32-bit architecture, the 80386 rivaled the complexity and power of minicomputers and mainframes introduced just a few years earlier. This was the first Intel processor to support multitasking, meaning it could run multiple programs at the same time.

- **80486:** The 80486 introduced the use of much more sophisticated and powerful cache technology and sophisticated instruction pipelining. The 80486 also offered a built-in math coprocessor, offloading complex math operations from the main CPU.

- **Pentium:** With the Pentium, Intel introduced the use of superscalar techniques, which allow multiple instructions to execute in parallel.

- **Pentium Pro:** The Pentium Pro continued the move into superscalar organization begun with the Pentium, with aggressive use of register renaming, branch prediction, data flow analysis, and speculative execution.

- **Pentium II:** The Pentium II incorporated Intel MMX technology, which is designed specifically to process video, audio, and graphics data efficiently.

- **Pentium III:** The Pentium III incorporates additional floating- point instructions: The Streaming SIMD Extensions (SSE) instruction set extension added 70 new instructions designed to increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.

- **Pentium 4:** The Pentium 4 includes additional floating- point and other enhancements for multimedia.

- **Core:** This is the first Intel x86 microprocessor with a dual core, referring to the implementation of two cores on a single chip.

- **Core 2:** The Core 2 extends the Core architecture to 64 bits. The Core 2 Quad provides four cores on a single chip. More recent Core offerings have up to 10 cores per chip. An important addition to the architecture was the Advanced Vector Extensions instruction set that provided a set of 256-bit, and then 512- bit, instructions for efficient processing of vector data.

Almost 40 years after its introduction in 1978, the x86 architecture continues to dominate the processor market outside of embedded systems. Although the organization and technology of the x86 machines have changed dramatically over the decades, the instruction set architecture has evolved to remain backward compatible with earlier versions. Thus, any program written on an older version of the x86 architecture can execute on newer versions. The rate of change has been the addition of roughly one instruction per month added to the architecture, so that there are now thousands of instructions in the instruction set.

The x86 provides an excellent illustration of the advances in computer hardware over the past 35 years. The 1978 8086 was introduced with a clock speed of 5 MHz and had 29,000 transistors. A six-core Core i7 EE 4960X introduced in 2013 operates at 4 GHz, a

speedup of a factor of 800, and has 1.86 billion transistors, about 64,000 times as many as the 8086.

3.3 ARM Architecture

The ARM architecture refers to a processor architecture that has evolved from RISC design principles and is used in embedded systems.

ARM Evolution

ARM is a family of RISC-based microprocessors and microcontrollers designed by ARM Holdings, Cambridge, England. The company doesn't make processors but instead designs microprocessor and multicore architectures and licenses them to manufacturers.

ARM chips are high-speed processors that are known for their small die size and low power requirements. They are widely used in smartphones and other handheld devices, including game systems, as well as a large variety of consumer products. ARM chips are the processors in Apple's popular iPod and iPhone devices, and are used in virtually all Android smartphones as well. ARM is probably the most widely used embedded processor architecture.

The origins of ARM technology can be traced back to the British-based Acorn Computers company. In the early 1980s, Acorn was awarded a contract by the British Broadcasting Corporation (BBC) to develop a new microcomputer architecture for the BBC Computer Literacy Project. The success of this contract enabled Acorn to go on to develop the first commercial RISC processor, the **Acorn RISC Machine (ARM)**. The first version, **ARM1**, became operational in 1985 and was used for internal research and development as well as being used as a coprocessor in the BBC machine.

In this early stage, Acorn used the company VLSI Technology to do the actual fabrication of the processor chips. VLSI was licensed to market the chip on its own and had some success in getting other companies to use the ARM in their products, particularly as an embedded processor.

The ARM design matched a growing commercial need for a high-performance, low-power- consumption, small- size, and low- cost processor for embedded applications. But further development was beyond the scope of Acorn's capabilities. Accordingly, a new company was organized, with Acorn, VLSI, and Apple Computer as founding partners, known as ARM Ltd. The Acorn RISC Machine became Advanced RISC Machines.

Instruction Set Architecture

The ARM instruction set is highly regular, designed for efficient implementation of the processor and efficient execution. All instructions are 32 bits long and follow a regular format. This makes the ARM ISA suitable for implementation over a wide range of products.

Augmenting the basic ARM ISA is the Thumb instruction set, which is a reencoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus, and to allow better code density than provided by the ARM instruction set. The Thumb instruction set contains a subset of the ARM 32-bit instruction set recoded into 16-bit instructions.

ARM Products

ARM Holdings licenses a number of specialized microprocessors and related technologies, but the bulk of their product line is the **Cortex family** of microprocessor architectures. There are three Cortex architectures, conveniently labeled with the initials A, R, and M.

- **Cortex-A/Cortex-A50:** The Cortex-A and Cortex-A50 are application processors, intended for mobile devices such as smartphones and eBook readers, as well as consumer devices such as digital TV and home gateways (e.g., DSL and cable Internet modems). These processors run at higher clock frequency (over 1 GHz), and support a memory management unit (MMU), which is required for full feature OSs such as Linux, Android, MS Windows, and mobile OSs. The two architectures use both the ARM and Thumb-2 instruction sets; the principal difference is that the Cortex-A is a 32-bit machine, and the Cortex-A50 is a 64-bit machine.

- **Cortex-R:** The Cortex-R is designed to support real-time applications, in which the timing of events needs to be controlled with rapid response to events. They can run at a fairly high clock frequency (e.g., 200MHz to 800MHz) and have very low response latency. The Cortex-R includes enhancements both to the instruction set and to the processor organization to support deeply embedded real-time devices. Most of these processors do not have MMU; the limited data requirements and the limited number of simultaneous processes eliminate the need for elaborate hardware and software support for virtual memory. The Cortex- R does have a Memory Protection Unit (MPU), cache, and other memory features designed for industrial applications. Examples of embedded systems that would use the Cortex-R are automotive braking systems, mass storage controllers, and networking and printing devices.

- **Cortex-M:** Cortex-M series processors have been developed primarily for the microcontroller domain where the need for fast, highly deterministic interrupt management is coupled with the desire for extremely low gate count and lowest possible power consumption. As with the Cortex-R series, the Cortex-M architecture has an MPU but no MMU. The Cortex- M uses only the Thumb-2 instruction set. The market for the Cortex- M includes IoT devices, wireless sensor/actuator networks used in factories and other enterprises, automotive body electronics, and so on.