Chapter 1

Introduction to Computer Architecture

מטרת הפרק הזה היא הגדרת המקצוע **ארכיטקטורות מחשבים** וסקירה קלה על מושגים בסיסיים.

Slides 2 – 4 What is Computer Architecture

Computer Architecture is a sub-discipline of Computer Science defined in Wikipedia as:

> In computer engineering, computer architecture is a set of **rules** and **methods** that describe the **functionality**, **organization**, and **implementation** of computer systems. Some definitions of architecture define it as describing the capabilities and **programming model** of a computer but not a particular implementation. In other definitions computer architecture involves **instruction set architecture** design, **microarchitecture** design, **logic design**, and **implementation**.

The **rules** and **methods** are determined by theoretical considerations based primarily on performance goals and market factors.

Functionality refers to the actions that a computer can perform and the ways in which it can perform them. This includes the **programming model** for the system and the **Instruction Set Architecture (ISA)**.

Organization refers to the division of the computing system into functional subsystems and the relationships among the subsystems. Another term for the organized interaction among subsystems is **microarchitecture**. An ISA can be organized in many different ways.

Implementation refers to the choice and **design** of digital and analog devices used to make each subsystem operate as required. A microarchitecture can be implemented in various different ways. The word implementation is often used in less specific ways, referring to converting a set of goals into a detailed design.

Instruction Set Architecture (ISA)

The term ISA is used in two different but related ways:

1. The set of considerations involved in choosing a particular set of instructions
2. A "family" of processors using the same instruction set, for example the Intel x86 processor family including the Pentium, Centrino, Core, i5, i7, Xeon and Atom.

Processor design proceeds in 4 steps:

1. Determination of the ISA,
2. A proposed microarchitecture,
3. Theoretical determination of predicted performance for the microarchitecture,
4. Changes to the microarchitecture (and possibly the ISA) to improve performance.

In computer architecture **performance** generally refers to processing speed, which will be defined in detail in chapter 3. The most important performance goals are:

1. Short run time — each defined task should run quickly with low latency (waiting time between programs and operations).
2. Low energy consumption — the system should minimize the cost of electricity, permit long battery life and prevent systems from overheating.

Important **market factors** are:

1. Low cost in relation to realistic demand for devices — it is not practical to manufacture computer systems that are too expensive for their potential application.
2. Reliable manufacture and delivery — it must be possible to deliver systems and subsystems to market within a timeframe required by the customer. The inability to deliver systems on a predictable schedule has often prevented high performance equipment from sustaining a successful market.
3. Profitability — the computing industry is accustomed to much higher profit margins than most business sectors. High performance computing systems can become uninteresting to reliable manufacturers when their profitability falls into levels that would be quite welcome in other industries.

Slides 5 — 6 Computing Platform by Application

The performance requirements for a computer system depend upon its applications and uses. Typical applications of computer systems include:

Workstation applications

A workstation is a single-user general-purpose computing environment used primarily for office applications, basic number crunching, graphics, gaming and programming. Typical programs consist of a few sequential loop-oriented threads. The typical CPU is a member of the Intel x86 family with 2 to 16 cores.

Mobile applications

Mobile applications — telephones and tablets — can be described as a low power version of the workstation. Because these devices require reasonable speed at very low energy consumption, the typical CPU is an ARM processor with 1 to 4 cores.

Online Transaction Processing (OLTP)

OLTP applications involve relatively simple code that performs database access for banking operations, credit card and order processing, inventory management systems, student information systems and so on. Such systems handle thousands of independent SQL transactions, with low code complexity but significant memory latency. Transactions are managed by concurrent threads so that CPUs remain active during the memory I/O wait. The typical CPU is a SPARC processor with 64 to 256 cores. The SPARC architecture was developed by Sun Microsystems which was purchased by Oracle, an industry leader in OLTP software.

An alternative to a specialized OLTP-optimized system is an x86 **general-purpose server**. In this usage, a server is similar to a workstation with less support for the graphic user interface (GUI) but more memory and increased hardware support for Input/Output (**I/O**) operations.

Supercomputer applications

A supercomputer is designed for heavy number crunching (mathematical programming), data mining and so on. The programming model is known as **massively parallel computing** because large tasks are divided among many processors. Typical applications run thousands of separable and sequential loop-oriented threads. A typical supercomputer is built from tens of thousands of IBM Power CPUs with 256 cores per CPU.

Mainframe

A mainframe computer is a business-oriented system designed to perform the work of 10 to 1000 server systems. A typical mainframe contains 120 CPU cores with 3840 GB RAM and is capable of maintaining 8 GB/s of continuous I/O. A unified, top-down system design provides very **high reliability** with a mean time between failures (MTBF) measured in years. Maintenance is simplified by centralized control, power, cooling and monitoring. The system permits complex partitioning of resources, so that a single mainframe can provide many separate instances of multiple independent operating systems, each serving many users. The mainframe can reallocate hardware subsystems as needed for load balancing.

Server **virtualization** is an alternative to mainframe partitioning designed to run on conventional server systems. System software running above or below the primary OS can partition the hardware resources among multiple guest OSs.

Cloud computing

Cloud computing is not strictly speaking a computer architecture, but rather a **business model**. A provider of cloud service sells access to a standard-definition system interface as a service. Interface types are classified as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) or Storage as a Service (StaaS). The cloud customer sees a virtual system specified in a contract called a service level agreement (SLA) and can refer its own customers to the resources it purchases from the cloud provider. A major economic advantage is that the cloud provider handles OAM tasks, (Operations, Administration and Maintenance).

Slide 8 **Basic Definitions**

The most important performance characteristic is processing speed, measured as **Response Time**, which is the elapsed time T from the start to finish of a defined program task. **Run time** is the response time for a complete program, and includes both useful work performed by the CPU and time spent waiting for external events.

Latency is the time spent waiting. Which part of run time is useful work and which part is latency depends on the situation.

Throughput is the number of defined tasks performed per unit time, that is, the task rate including latency between tasks (such as OS scheduling).

Any change to system is called an **enhancement** (even if the change degrades performance). After an enhancement, the new run time is written T '.

The **speedup** is the ratio $S = \frac{T}{T'}$ which is greater than 1 when the enhancement improves run time, in which case $T' < T$.

Slide 9 Run Time and Clock Cycles

Timing in the CPU is determined by a periodic **Clock** signal. The **clock period** is measured in seconds per cycle. The **clock rate** is measured in cycles per second.

$$\text{Clock Rate} = \frac{\text{cycles}}{\text{second}} = \frac{1}{\frac{\text{second}}{\text{cycles}}} = \frac{1}{\text{Clock Period}}$$

Run time can be measured in seconds or by counting clock cycles (CC):

$$\begin{aligned}\textbf{Run time} &= \text{clock cycles to run program} \times \text{seconds per clock cycles} \\ &= \frac{\text{clock cycles to run program}}{\text{clock cycles per second}} = \frac{\text{program clock cycles}}{\text{clock rate}}\end{aligned}$$

If the clock rate is higher, then the run time is shorter. But also, if the program requires fewer CC to run then the run time is shorter.

Slide 10 Speedup and Clock Rate

Speedup is the old (longer) run time divided by the new (shorter) run time. It can be expressed in terms of the clock cycles speedup and the clock rate speedup:

$$\begin{aligned}S &= \frac{T^{old}}{T^{new}} \\ &= \frac{\text{program clock cycles}^{old} \times \text{seconds per clock cycle}^{old}}{\text{program clock cycles}^{new} \times \text{seconds per clock cycle}^{new}} \\ &= \frac{\text{program clock cycles}^{old} \,/\, \text{clock rate}^{old}}{\text{program clock cycles}^{new} \,/\, \text{clock rate}^{new}} \\ &= \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{\text{clock rate}^{new}}{\text{clock rate}^{old}}\end{aligned}$$

If the clock rate doubles with no change in CPU structure, then S = 2. But if a structural change reduces by one half the number of clock cycles required to run a program then again S = 2.

Slide 11 Factors Affecting Run Time

CPU hardware

The CPU design determines the average number of clock cycles (CC) required to perform an instruction. A more efficient CPU performs a given instruction in fewer CC.

Memory

Computer memory includes long-term storage, main memory (RAM) and cache memory internal to the CPU. The quantity and organization of memory affects the availability of data to the CPU. Generally, increasing memory size and organizing it efficiently decreases the time the CPU must wait for data.

Internal communication and I/O

The speed and organization of the I/O system affects data availability related to long-term storage and networks.

Operating system efficiency

The CPU alternates between running application code and the OS code that performs services for the application required and keeps the system functioning. If the OS code is dense (few instructions per task) then the CPU devotes less time to the OS and more time to applications. The OS also manages tasks and threads to keep the CPU and I/O hardware busy on long waits (such as page faults and program loads). Efficient scheduling insures that applications run more quickly.

Compiler

The compiler converts code from a high-level language to machine code. Optimized machine code is designed to run faster on specific hardware, taking advantage of its specific capabilities.

Special hardware

Most computer systems have dedicated special-purpose processors to handle graphics operations, memory management, I/O and more specialized features such as encryption, virtualization, load balancing and so on.

Application code

Application code will run more quickly when the programmer uses efficient algorithms, data structures, parallelization techniques, and so on. Generally, PC programming is intended for code mobility — it must run on various types of processor and different operating systems. This mobility involves compromises in choosing parameters for instruction types, data structures, cache sizes and other considerations. Game designers programming for a specific target platform (Xbox, PlayStation, etc) can optimize for specific hardware to achieve higher performance.

Slides 13 — 16 CPU Hardware and Memory Examples

The most obvious example today of the influence of the CPU hardware on system performance is the number of **CPU cores** in the main processor. Most personal computers and telephones now include a CPU with 2 to 8 cores in a configuration called **Symmetric Multiprocessor (SMP)**. The configuration is called symmetric because all of the cores perform the same kind of work.

Each core can be thought of as roughly equivalent to a single Intel Pentium 4 processor manufactured in 2001. Each processor core includes its own **cache memory**, **execution units** (**EU**s), and a complete set of **registers**. Executions units include the **arithmetic logic unit** (ALU) that performs operations on integers, the **floating point unit** (FPU) that performs operations of float and double operands and a **vector processor** (slide 14) that operates on long vector-type registers.

In the **ideal case**, all cores are busy working in **parallel** and so an N core CPU performs N times the work of a single core (we will see that this does not deliver ideal linear speedup). If code is written in a way that permits separable threads to run independently and if the data structures are not too entangled (including different types of data mixed together) then the threads can be run in parallel on separate cores.

Slide 14 shows an example of an operation for a **vector processor**. In this example, each of two 64-bit registers holds eight byte-long operands — byte 0 to byte 7. The **Parallel Addition** adds byte 0 from the source (SRC) register to byte 0 from the destination (DEST) register and stores the result in byte 0 of DEST. At the same time it adds byte 1 (SRC) to byte 1 (DEST) and so on.

There is no carry from one byte to another — these are eight separate byte-oriented additions. Some vector processors can perform parallel operations on eight 64-bit operands at once.

The vector processor can perform in one clock cycle the work an ALU performs in eight sequential iterations of a loop. But the code must be written in a way that permits access to eight sequential operands. For example, if a graphics structure for a figure or 200 points is defined as

```
struct { float x, y, z, r, g, b ; } H [200] ;
```

then each operand in memory is of a different type from the next — there is no way to operate in parallel. Slide 15 shows a **hybrid structure** designed to allow **parallel operation** in the vector processor. Two structures are defined as

```
struct { float x[8], y[8], z[8] ; } H_xyz[25] ;
struct { float r[8], g[8], b[8] ; } H_rgb[25] ;
```

The operands appear in memory as 8 x-coordinates followed by 8 y-coordinates, and then 8 z-coordinates. On each iteration, the vector processor fetches 8 x-coordinates and performs 8 x-operations in parallel, then fetches 8 y-coordinates and performs the y-operations in parallel, and then fetches 8 z-coordinates and performs the z-operations in parallel. On the next iteration, it fetches another 8 x-coordinates and so on. A separate thread running on a separate processor core operates on the color points.

Slide 16 shows another interaction between CPU structure and memory allocation. Most 32-bit processors save complexity by using 30 lines for the memory address — the lowest two bits of every address sent from the CPU to memory are always 0. This is called a 32-bit **aligned address** and it means that the address A sent to memory is always a multiple of 4. When the CPU sends address A, it receives the 4 bytes from addresses A, A+1, A+2, A+3.

The human eye cannot distinguish between 2^{24} different colors and so **24-bit color** (1 red byte, 1 green byte, and 1 blue byte) covers all the colors we can see. But a **pixel** (short for picture element) written as a 3-byte (24-bit) color word is not convenient to store in memory. Half of all 3-byte combinations are split across 4-byte aligned memory accesses — two read cycles are required to fetch all bytes for a pixel.

To make graphic memory access more efficient, pixels are stored as 4 bytes = **32-bit color**. The use of the 4^{th} byte for additional pixel information began to take advantage of the extra space.

Slides 17 — 22 Compiler Efficiency Example

Slide 17 shows a short C program compiled for an Intel 8086 processor. The **assembly language** code uses memory locations to store **variable operands** in the manner of the C language. For example, line 0000 copies the 16-bit value 0 to the memory location [BP-02] — this memory address is formed by subtracting 2 from the address value stored in the base pointer register BP. Similar operations appear in lines 05, 0B, 10, 13.

On 8086 CPUs, **memory access** operations are much slower than register accesses. Slide 18 shows a page from the **Intel 80186 manual** — for each instruction type the right column shows the **number of clock cycles** required to perform one instruction of that type.

The first row is a **move instruction** from a register to either another register or memory. We see that a **register-to-register** move requires 2 CC, but a **register-to-memory** move requires 12 CC.

Slide 19 shows the assembly program along with the instruction type and required CC. The first and last instructions (light green) run only once at the start and end of the program. Four

instructions (dark green) form the loop defined in the C program — compare, increment, branch. Three instructions (light blue) are the ALU action in each loop iteration.

Slide 20 shows the program with the specific number of CCs required for each line. Notice that the conditional branch JGE requires 4 CC on all but the last loop iteration because its condition is not met and it does not jump — on the last loop iteration, its condition is met and it jumps to stop, which requires 13 CC.

For N loop iterations we can add the number of CCs for all the instructions. The first instruction runs once in 13 CC. The second instruction runs N times for 10 CC each. The next six instructions run on all but the last iteration (N – 1 times). Finally we add the 13 CC for JGE on the last iteration and the 16 CC of RET at the very end. This adds up to **66N – 14** clock cycles. Taking i = 10, there are N = 11 iterations and so the program runs in 712 clock cycles.

Slide 21 shows a **compiler optimization** that is possible for this program. Since the registers SI and DI were not used in the program, it is possible to store the **variables** i and j in these **registers**. Since register access is much faster than memory access, there is an enhancement in several lines — the new CC numbers appear in the right column. The program now runs in 30N + 6 CC and for N = 11 the new total is 377 clock cycles for this implementation of the same C program.

Since the clock speed is not changed the **speedup** is just the ratio of CC^{old} to CC^{new} which comes to 712 CC / 377 CC= 2.11. The optimized program runs in about half the CCs and should run about twice as fast as the original program.

Slide 22 shows another optimization that is not available in the compiler. For critical path routines — motion in computer games or calculation in Matlab libraries — programmers write optimized routines in assembly language. By **rebuilding the loop** using the Intel LOOP instruction, it is possible to reduce the number of CCs from 377 to 261. The speedup relative to the original program is now 2.73.

Slide 24 **Benchmarks**

Benchmark programs are used to test system performance. Many benchmarks exist for measurement of various system characteristics under various types of load and application. Any benchmark must meet the basic requirements of a scientific measurement:

1. Standard and repeatable: any test on a given system under fixed conditions should provide the same results.
2. Realistic: the benchmark must test the system under conditions similar to those in which it will be used, including application type and system load.
3. Comparison: the benchmark must provide easy comparison between results on different systems.

Slides 25 – 31 **SPEC Benchmark**

SPEC

A widely accepted benchmark is **SPEC**, a product of the **S**ystem **P**erformance **E**valuation **C**orporation. Results for many systems are posted at **www.spec.org**.

SPEC sells testing suites for various types of performance, including CPU integer (int) operations, CPU floating point (fp) operations, system as a file server, web server, mail server, graphics

processor. The suites are updated every few years to reflect realistic conditions. The current versions for CPUint and CPUfp were released in 2006.

How SPEC Works (slide 26)

The performance measure in these tests is speedup compared to a reference system. Run times T_i^{ref} for n programs on the reference system (i = 1, 2, ... , n) are provided by SPEC. Run times T_i^{test} on the tested system are measured by the user. The ratio is the speedup $S_i = T_i^{ref} / T_i^{test}$ and the final score S is the geometric average of the S_i . Given scores S_A and S_B for systems A and B the score of A compared to B will be S_A / S_B .

The programs included in SPEC 2017 and their reference run times are listed on **slide 27**. The reference machine is a Sun Fire V490, a computer that was considered extremely fast when it was introduced in 2006. The CINT suite includes programs with integer-intensive calculations and the CFP suite includes programs with float-intensive calculations. The run times are very long — up to 6,200 seconds (nearly 2 hours) — so that runtime can be measured precisely on a standard clock.

Typical SPEC Report

The first part of the report lists **number of Threads**, $\mathbf{T_i^{test}}$, and the speedup **ratio** $\mathbf{S = T_i^{ref} / T_i^{test}}$ for each of the n programs. The tests are performed for a **Base** (standard) configuration and then a **Peak** (optimized) configuration. The average is reported at the bottom of the ratio columns. **Slides 28 – 29** show a report for the ASUS RS700-E9, a server system based on 2 Intel Xeon processor chips containing 18 cores per chip and 2 threads per core. The report shows that the runtimes are, on average, about 9% of the runtimes on the reference machine — the base **SPECint score** is about 8.87. Notice that one program runs 22 times as fast while another runs only 4 times as fast. These differences represent the structural improvements in the newer system that are advantageous for some programs but not for others.

Information regarding the system configuration is reported following the test results. **Slide 29** shows technical details for the system under test: a Xeon Gold 6150 processor running at 2700 MHz clock rate (2.7 GHz), with 32 KB of instruction L1 cache (I) and 32 KB of L1 data cache (D) per core (total of 4.6 MB cache), 1 MB of unified L2 cache per core (cache is covered in presentation 8), 24.75 unified L3 cache per chip, and 64 GB of main memory (RAM) and a 240 GB SSD (solid state drive). The OS is Red Hat Linux.

Slides 30 – 32 Representative Results

The tables on **slides 30** and **32** show some representative results for CPUs on Cint2017 (SPECint) and Cfp2017 (SPECfp). The fastest CPU for integer instructions is the **Intel Xeon E-2278G** running at 3.4 GHZ (in the first row), a processor designed for powerful servers, workstations and powerful gaming machines. The tested machine contains 1 processor chip with 8 cores per chip, for a total of 16 running software threads. The base score on Cint2017 is 13.2. All processors on this table permit automatic parallelization of software. The SPEC software sources are written sequentially with no parallel algorithms — the compiler can automatically recognize opportunities to run parallel threads on multiple cores. But notice that a similar machine with 2 chips (48 threads) scores slightly lower than the machine running 16 threads. This may be an example of a known problem: the communication required for multithreading creates additional load on the system and running more threads can actually slow down the run time.

The **Intel i9** running 6 threads at 3.6 GHz has a score of 11.2. Similarly the **Intel i7** running 8 threads at 3.6 GHz has a score of 10.8. The increased speed of the **Xeon** processors is the result of internal system optimization, not simply clock speed or number of threads.

Slide 32 shows some results on Cfp2017. Because programs performing floating-point operations are more structured than integer programs and easier to parallelize, most systems tested on Cfp2017 have 8 or more multicore chips. We see that the 8280L system running 224 threads is only 5% faster than the system running 112 threads. The comparable 8165 processor running 8 threads has a score of 53.8. Although the 8280L running 112 threads has 14 times as many threads at the 8165, its score in only 4.3 times as high. We also notice that the 8280L and 8156, which achieve high scores on Cfp2017 have significantly lower scores on Cint2017 than the E-2278G. This suggests that integer-oriented programs and floating-point-oriented programs may require different hardware optimizations.

Because the multithreading is performed automatically by the compiler, only the most obvious parallelization opportunities are recognized. Even so, parallelization does not guarantee improved performance. Optimized utilization of multicore systems requires the intervention of a skilled programmer knowledgeable in parallel algorithms.

Each core in a multicore processor is approximately comparable to a single **Pentium M** CPU (Pentium M was the last single core upgrade of the Pentium 4). The early dual core processors (Core 2 Duo) were about twice as fast as a single core Pentium 4. Based on the above discussion, this doubling of speed is based on factors other than thread parallelization.

Slide 33 **Actual Sources of Performance Improvement**

The Intel **8086** processor manufactured in 1978 ran with a clock speed of 4 MHz. A program written for the 8086 can be run on an Intel **Xeon** processor manufactured in 2008 running with a clock speed of 4 GHz. It will run about 100,000 times as fast. Since

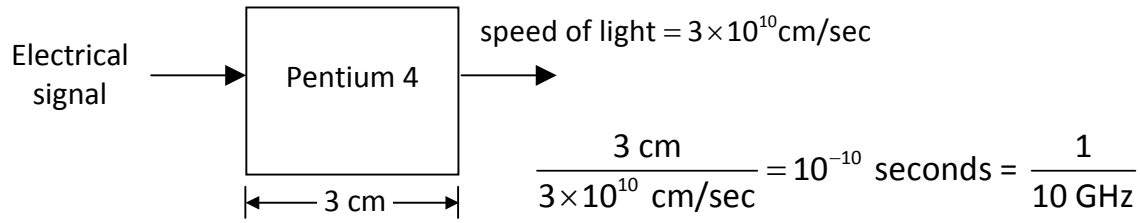
$$S = \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{\text{clock rate}^{new}}{\text{clock rate}^{old}} = \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{4\text{ GHz}}{4\text{ MHz}}$$

$$100{,}000 = \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times 1000$$

we see that structural improvements — fewer clock cycles per operation — provide a speedup of about 100.

In the past, **increasing clock speed** was the basic strategy for improving performance. For example, the Intel Pentium III ran at 1 GHz while the Pentium 4 can run at 4 GHz. Applying a 4 GHz clock to the Pentium III would have damaged it, causing it to overheat and melt. To design the Pentium 4 to run at 4 GHz required increasing the clock cycles per instruction. Thus, a 1 GHz Pentium 4 was actually slower than a 1 GHz Pentium III. But the 4 GHz Pentium 4 was still more than 3 times faster than the 1 GHz Pentium III and this was a reasonable compromise.

The increase in clock speed will not continue, because clock speeds are approaching a fundamental physical limit. At 10 GHz the time it takes for an electrical signal to cross a CPU at the speed of light is about one clock cycle (1 / 10 GHz = 10^{-10} seconds).



At 10 GHz it would be difficult to maintain data consistency across the CPU because signals would appear in waves rather than binary 0 or 1.

Future speedup must come from structural improvements — more cores, better parallel programming and better architectures.