Presentation 1

Review of Basic Computer Architecture

תנאי הקדם לקורס **ארכיטקטורות מתקדמות** הוא קורס בארכיטקטורה בתואר ראשון. מטרת הפרק הזה היא סקירה קלה על הידע הבסיסי הדרוש לקורס מתקדם כפי שהחומר נלמד בקורס **ארכיטקטורות מחשבים** לתואר ראשון במכללת הדסה. מיותר לומר שאין אפשרות בפרק זה להעמיק בכל נושא שנלמד בסמסטר שלם — וכמו כן אין צורך. הפרק מיועד ליצור שפה משותפת בסיסית. מי שמבקש הסברים נוספים על תוכן המצגת הזאת מוזמן/ת לשאול בפורום הקורס או להסתכל באתר הקורס לתואר ראשון בכתובת:

cs.hadassah.ac.il/staff/martin/Adv_Architecture/BSC

Slides 2 – 4 **What is Computer Architecture**

Computer Architecture is a sub-discipline of Computer Science that includes:

1. The study of **performance** in computer systems
2. The practical science of applying **performance theory** to **specification** of the structure and relationship of the subcomponents of a computer. The theory includes **computability** and **performance** in digital systems.

The **goals** of a specific architectural design depend on the application **requirements**.

Specification of an architectural design is the choice of digital systems that can be predicted by performance theory to meet the system requirements.

Most processors are examples of a **von Neumann Architecture**, a digital system with the following features:

1. Digital computation in an ALU (arithmetic / logic unit)
2. Programmable via a set of standard instructions
3. Internal storage of data
4. Internal storage of programs
5. Automatic Input / Output (I/O)
6. Automatic sequencing of instruction execution by a decoder/controller

---

Slide 5 **Requirements — Relative to Application**

System performance is relative to the applications that run on the system. For example, three distinct application areas with different performance requirements are: **general purpose workstation** (including PC), **supercomputer**, **smartphone**.

Currently, the fastest general purpose CPU (processor) is the **Intel Xeon**, a high performance model in the x86 processor family, which also includes the CPUs found in mass market PCs. It conforms to Intel's IA-32 architectural standard and operates as an enhanced implementation of the P6 hardware model introduced in the 1990s for the Pentium Pro and Pentium II processors. The Xeon is used for high performance workstations and general application servers.

The fastest supercomputer is the IBM Sequoia, which uses a CPU from the **IBM POWER** architectural family. A single POWER processor has lower performance than Xeon, but is

designed to enable systems with the millions of processor nodes and huge quantities of memory required for supercomputing.

Most smartphones use an **ARM** processor that provides an optimized balance of high performance, a machine language with constructs familiar to Intel programmers, and low power consumption.

Slide 6 Fundamental Architectural Abstractions

Digital computer	Any machine that can be programmed to process symbols.
Data	A symbol with no intrinsic meaning to the machine, such as integer, float, string. The user imposes the meaning of the symbol.
Operation	A symbol describing the processing of data symbols. The machine interprets the meaning — transfer, ALU, control, OS, ...
Instruction	A symbol describing an operation on data. A machine language is a collection of legal instructions.
Addressing Mode	A system for specifying data locations as operands. A source operand is a data input to an operation. A destination operand is a data output from an operation

Slide 7 Stages in Computer Design

The design of a processor begins with the specification of its **instruction set** (machine language). The term **Instruction Set Architecture** (ISA) refers to the machine language and digital system components required to define instructions (registers, ALU types, etc.).

The ISA is related to the system requirements through the following steps:

1. Define the universe of problems to be solved
2. Study candidate operations at the level of the system programmer
3. Specify an instruction set for the machine language — a minimum set of orthogonal operations (leaving not too many ways to solve a given problem)

The CPU is designed by assembling abstract digital components (ALU, memory, registers, controllers, etc.) into a system that implements the ISA (it can perform the operations defined by all the instructions). The steps are:

1. Design the machine as an **implementation of ISA**
2. Evaluate theoretical performance
3. Identify performance problem areas
4. Improve processor efficiency and performance

A working model of a CPU is only built after theoretical performance evaluation.

The ingredients of the most general ISA are described on slides 8 – 13.

Slide 8 Typical Operations

Computational operations are performed in an **execution unit** (EU), such as arithmetic / logic unit (ALU), floating point unit (FPU), vector processor, graphics processing unit (GPU) and other types to be discussed later. Several types of operation are described on slide 8.

Slide 9 Memory Hierarchy

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

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

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

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

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

Slide 10 CPU and Memory Hierarchy

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

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

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

On a cache miss, Layer 1 cache (L1) requests the data location from the cache controller. The controller requests the data location from the Layer 2 cache (L2) and from main memory at the same time (L2 access time is much shorter than main memory access time). If the location is currently copied to L2, the controller copies it to L1 and cancels the request from main memory. If the location is not currently copied to L2, the controller copies it to L1 and L2 from main memory. The delay caused by updating L1 cache is called a cache penalty.

Slide 11 – 12 Specifying Operands

Access to operands is specified by **Addressing Modes**, which are formalized

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

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

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

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

`REGS[reg1] ← REGS[reg2] + IMM`

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

Pointer arithmetic is enabled by evaluating an expression: **`LW reg1,IMM(reg2)`**
`REGS[reg1] ← MEM[REGS[reg2] + IMM]`

Slide 12 lists some common addressing modes.

Slide 13 Commitment to State

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

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

Slide 14 – 15 Complex Instruction Set Computer (CISC)

Complex Instruction Set Computer (CISC) is a recent term describing the common form of ISA used for CPU designs from about 1950 to 2000. The complexity is seen in the large instruction set with many addressing modes and date types. CISC was the design pattern used for computers designed during this period:

1. Mainframe computers — large, expensive, centralized computers designed for big business and government.
2. Minicomputers — first small computers (the size of a refrigerator) supporting a few users and applications, and cheap enough for a large organization to own several.
3. PCs — developed from a single user desktop workstation (1981) to multi-user machines more powerful than earlier minicomputers (2000). Intel's x86 is the last CISC ISA still in common use.

The theory behind CISC was based on a general notion that computer language should imitate natural language (human languages). This means that a large vocabulary with high redundancy should insure flexibility and power. Several practical considerations were also important.

Compilers had limited power so that efficient code was always written in assembly language. An ISA with many features makes it easier to code in assembly language. Also RAM was very expensive — a powerful instruction set that accomplishes many tasks with short instructions will use less memory.

Slide 16 – 19 Physical Implementation of CISC

Older CISC implementations were based on a generic machine consisting of system components organized around a system bus that can carry one operand between components at a time. The components are the ALU subsystem (ALU with temporary data buffers), registers, memory subsystem (buffers for data and address), a program counter (PC) that stores the address of the next instruction in memory, an instruction register that stores the fetched instruction (machine code), and a decoder that interprets instructions, controls the sequence of steps involved in executing each instruction, and a status word that saves certain consequences of instructions (output is positive, negative, zero, etc.).

The simple assembly instruction **`SUB R1, R2, 100(R3)`** requires the following 9 steps to implement:

`ALU_IN ← R3`	Copy value in R3 to ALU input buffer
`ALU ← 100`	Copy value 100 from decoder to ALU
`ADD`	ALU adds 100 to value in R3 to form address
`MAR ← OUT`	Copy address from ALU output buffer to MAR
`READ`	Copy operand into MDR using address in MAR
`ALU_IN ← MDR`	Copy operand in MDR to ALU input buffer
`ALU ← R2`	Copy value in R3 to ALU
`SUB`	ALU subtracts memory operand from R2
`R1 ← OUT`	Copy result from ALU output buffer to R1

The 9 steps in this example are called a **microprogram** or **microcode**. Each line of microcode must be executed atomically (run start to finish, with no interruptions and no other microcode running), because only one operand may be transferred on the bus at any time.

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.

The clock period is determined by the execution time of the longest instruction in the microcode set. Every line of microcode takes exactly one clock cycle (even if it could be finished in less time). The microprogram in the example takes 9 clock cycles to execute. If the clock rate is 1 MHz = 10^6 cycles/second, then the clock period is 1 microsecond = 10^{-6} seconds. This instruction would run in 9 microseconds.

Despite its complexity, the CISC design is **highly generic**. Almost any possible instruction can be defined by a microcode sequence that specifies moving data across the system bus between the ALU, memory, registers and decoder. These sequences are stored in the decoder. The first byte of an instruction in the instruction register (IR) points to an entry in the decoder (in read-only memory, ROM) that contains the microcode sequence. A new instruction can be implemented by adding a new microcode sequence to the decoder ROM.

The complexity of a CISC processor can be seen in the simplified diagram for the Intel 386 (designed in Haifa). This CPU was the first 32-bit device conforming to IA-32 and was introduced with a full Unix operating system written for the ISA. The PC-AT based on this processor permitted Windows 3.1 to perform limited multitasking.

Slide 20 Basic Performance Measures

The most important performance characteristic is processing speed, measured by **Run Time**, which is the elapsed time T from start to finish of a defined program task. Run time includes 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 21 Basic Performance Measures

It is useful to divide the run time into more precise units.

Instructions can be divided into **groups** — in each group the instructions perform the same general kind of work (ADD and SUB) and require the same number of clock cycles. Groups can also be formed as a weighted average among instructions that run in different numbers of clock cycles (MULT and SQRT). Typical groups are the operation types (slide 8).

For each instruction group i, we can find the **run time** t_i for all the instructions in that group. For example, the total run time of all the integer ALU instructions ($+ - \times \div$) in a given program is written t_{ALU}. For a program that runs in T = 20 microseconds it could be that t_{ALU} = 9 microseconds.

The number of instructions (**Instruction Count**) in group i is written IC_i.

The performance of instruction group i is determined by the number of clock cycles (**CC**) required to run one instruction from the group. This performance measure is written CPI_i for **Cycles Per Instruction**. CPI_i is a weighted average over the group and gives the number of clock cycles required to run a representative member of the group. If all instructions of group i run in the same number of cycles, then the average is just the CPI_i for each instruction. From the microcode on slide 17, for instructions of the type ALU register, memory on the generic machine, $CPI_{ALU\text{-}reg\text{-}mem}$ = 9.

N_i is the number of clock cycles required to run all of the instructions in group i.

The **period** of one clock cycle (measured in seconds per cycle) is written τ (the Greek letter tau).

The **clock rate** (clock speed, clock frequency) $R = 1 / \tau$ measured in cycles per second.

The **total run time**, total number of clock cycles, and average Cycles Per Instruction is found from the contributions of each group.

Slide 22 – 23 CPU Equation

The total run time T, instruction count IC, and clock cycles N for the whole program are easily found by adding the contributions from all instruction groups:

$$T = \sum_{\text{all groups } i} t_i \qquad IC = \sum_{\text{all groups } i} IC_i \qquad N = \sum_{\text{all groups } i} N_i$$

It also follows from the definitions that

N_i = CC per instruction × number of instructions = $CPI_i \times IC_i$

t_i = number of CC for group i × seconds per CC = $N_i \times \tau = CPI_i \times IC_i \times \tau$

Therefore,

$$N = \sum_i N_i = \sum_i IC_i \times CPI_i$$

$$T = \sum_i t_i = \sum_i IC_i \times CPI_i \times \tau$$

Combining these expressions, the Cycles Per Instruction for the whole program is:

$$CPI = \frac{\text{total number of clock cycles to run program}}{\text{total number of instructions in program}} = \frac{N}{IC}$$

$$= \frac{1}{IC}\sum_i IC_i \times CPI_i = \sum_i CPI_i \times \frac{IC_i}{IC}$$

The ratio $\frac{IC_i}{IC}$ is proportion (weight) of instructions in group i. Since $\sum_i \frac{IC_i}{IC} = 1$, the CPI for the entire program is the weighted average of the CPI_i with each group contribution weighted by the relative number of instructions in the group.

Now the total run time T for the whole program can be written in terms of the average number of CC per instruction, the number of instructions, and the clock period.

$$T = \sum_i CPI_i \times IC_i \times \tau = \left(\sum_i CPI_i \times \frac{IC_i}{IC} \right) \times IC \times \tau = CPI \times IC \times \tau$$

Slide 24 Amdahl Equation

Amdahl's equation expresses the speedup following an enhancement in terms of the relative quantities

$$F_i = \frac{t_i}{T} = \text{relative run time of instructions in group i}$$

$$S_i = \frac{t_i}{t_i'} = \text{speedup for instructions in group i}$$

Now, by using $\sum_i F_i = \sum_i \frac{t_i}{T} = \frac{\sum t_i}{T} = 1$ the speedup becomes

$$S = \frac{T}{T'} = \frac{\sum_i t_i}{\sum_i t_i'} = \frac{\sum_i t_i}{\sum_i \frac{t_i}{S_i}} = \frac{\sum_i F_i T}{\sum_i \frac{F_i T}{S_i}} = \frac{\sum_i F_i}{\sum_i \frac{F_i}{S_i}} = \frac{1}{\sum_i \frac{F_i}{S_i}}$$

For a single enhancement to group e (enhanced) leaving the groups ē unchanged

$$S = \frac{1}{\frac{F_{\bar{e}}}{S_{\bar{e}}} + \frac{F_e}{S_e}} = \frac{1}{\frac{1-F_e}{S_{\bar{e}}} + \frac{F_e}{S_e}} = \frac{1}{1 - F_e + \frac{F_e}{S_e}}$$

Slide 25 Amdahl Equation in Parallel Processing

Amdahl's equation was developed to describe the speedup in **parallel processing**.

The assumption is that a program includes a part that can be run in parallel and a part that must run sequentially. The part that runs in parallel is divided among n processors, so that instead of N clock cycles, the parallel part will only require N/n clock cycles. The CPI for n processors is then N / IC $\rightarrow$ N / (n × IC).

Treating parallelization as an enhancement to the part of the program that can run in parallel (leaving the serial part unchanged),

$$\begin{aligned} CPI^{n-\text{processors}} &= \frac{CPI^{n=1}}{n} \times \text{Probability}\left(\text{work can be parallelized}\right) \\ &\quad + CPI^{n=1} \times \text{Probability}\left(\text{work cannot be parallelized}\right) \\ &= \frac{CPI^{n=1}}{n} \times F_P + CPI^{n=1} \times \left(1 - F_P\right) \end{aligned}$$

$F_P =$ Fraction of processing that can be performed in parallel

$n =$ Number of processing units

$$S = \frac{CPI^{n=1} \times IC \times \tau}{CPI^{n-\text{processors}} \times IC \times \tau} = \frac{CPI^{n=1}}{CPI^{n-\text{processors}}} = \frac{CPI^{n=1}}{\frac{CPI^{n=1}}{n} \times F_P + CPI^{n=1} \times \left(1 - F_P\right)} = \frac{1}{\left(1 - F_P\right) + \frac{F_P}{n}}$$

Slide 26 – 29 SPEC Benchmark

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.

The 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.

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 is S_A / S_B .

The report lists T_i^{ref}, T_i^{test}, and the ratio for the n programs. The tests are performed for a Base (standard) configuration and a Peak (optimized) configuration. The average is reported at the bottom of the ratio columns. Information regarding the system configuration is reported following the test results.

Slide 30 – 31 Representative CINT Results

The fastest CPU for integer instructions is the **Intel Core i7-5960X**, 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 8 running software threads. This chip also permits 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. The base score on Cint is 79.7

The **Intel Core i7-6700K** processor has a similar clock speed to the **Core i7-5960X** and with a score of 77.4 is almost as fast, running only 4 threads. The comparison suggests that automatic parallelization may not be effective for more than a few threads. This limitation is seen again in the **Xeon E5-2699**, which achieves a score of Cint = 74.0 (4% lower) while running 44 threads (11 times higher) at 2.2 GHz (half the speed). If performance scaled linearly with the number of threads, this processor would have a Cint score in the neighborhood of 420. Similarly, the **Core 2 Duo E6850** running with automatic parallelization is seen to be only 5% faster than the same chip without parallelization disabled.

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 — a machine with dual **Xeon E5-2699** processors (88 threads instead of

44) achieved a slightly lower score than the single processor machine. Optimized utilization of multicore systems requires the intervention of a skilled programmer knowledgeable in parallel algorithms.

Each core in a multicore processor is 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 o the above discussion, this doubling of speed is based on factors other than thread parallelization. These considerations will be discussed later in the course.

Similar results are seen in the list of Cfp results.

Slide 32 **Benchmarking a Processor Design**

The experts at SPEC identify programs whose **instruction content is representative** of typical program behavior. If Cfp of CPU 1 is twice the Cfp of CPU 2, we can be reasonable sure that a typical game will run twice as fast on CPU 1 than on CPU 2 (with the same amount of memory and same graphics card).

To **evaluate** a new CPU design (without building it):

1. The system ISA defines a machine language for each instruction.
2. The proposed system implementation provides CPI_i for each instruction group (the number of CC required to implement each instruction).
3. Compilers are written for C, C++ and Fortran to the proposed machine language.
4. The benchmark programs are compiled for the proposed machine from the sources provided by SPEC.
5. The compiler output (a list of machine language instructions) is sorted by instruction group. The number of instructions IC_i in each group is counted.
6. $CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$ is calculated using CPI_i (from the ISA) and IC_i (from SPEC sort).
7. The clock period τ is estimated (an electrical engineering problem).
8. The run time $T = CPI \times IC \times \tau$ is compared with the reference machine to compute the SPEC score (Cint and Cfp).

Slide 33 – 34 CISC Creates Anti-CISC Revolution

From 1950 – 1975 most computers were large **mainframes** designed to satisfy all computing requirements for large businesses and government organizations. For theoretical and practical reasons (slides 14 – 15), the ISA consisted of a complex mixture of instruction types, addressing modes and data types. Today this approach is called **CISC** (complex instruction set computer).

In the mid-1970s the first 32-bit minicomputers (machines that can process 32-bit integers) were introduced. Minicomputer architectures were also based on the CISC approach. Data General introduced the **Eclipse** in 1974 and in 1977 Digital (DEC) introduced the **VAX**, which became the most successful alternative to mainframes for 25 years. Large businesses deployed a VAX to run a specific application that was easily isolated from the mainframe. Also, the availability of inexpensive minicomputers meant that small organizations began using computers for the first time. The success of minicomputers led to further developments:

1. Unix was developed as an operating system for minicomputers.
2. TCP/IP was developed to support minicomputer networks.
3. University departments could afford a computer for academic research and teaching — Computer Science emerged as separate academic discipline.
4. Computer Science students needed topics for projects, theses, dissertations. Research in architecture and performance theory expanded.

By the 1980s researchers had found surprising results about CISC:

1. CISC architectures use available machine resources inefficiently.
2. Most machine instructions defined in the ISA for a given machine are rarely used in running programs. About 90% of program run time is devoted to about 10% of the instructions the CPU can perform.
3. CISC machines run slowly because they need to support features that are unnecessary for most programs.
4. CISC was based on assumptions that were no longer relevant. The price of RAM dropped from $5000/MB in 1977 to $5/MB in 1990. Compilers had become more powerful and efficient with extensive automated optimization. Portable code became practical with the development of the minicomputer, Unix, C, and TCP/IP.

In the mid-1980s, researchers developed the idea of **RISC** (reduced instruction set computer). This approach states defines a **reduced instruction set**: the 10% of CISC instructions that accounts for 90% of run time. While the structure of a CISC machine must be generic to handle instructions of many types, a RISC machine is simpler, specializing in running the reduced instruction set as fast as possible. The remaining 90% of CISC instructions are implemented by short routines in the reduced instruction set — they will run somewhat slower that on a CISC machine, but they are only 10% of run time.

RISC "philosophy" follows from Amdahl's equation. If 90% of run time is 3 times faster than CISC but 10% of run time is only 80% as fast, the total speedup is still good:

$$S = \left[\frac{F_e}{S_e} + \frac{F_{\bar{e}}}{S_{\bar{e}}} \right]^{-1} = \left[\frac{0.90}{3} + \frac{0.10}{0.80} \right]^{-1} = 2.35$$

Slide 35 Instruction Types

The SPEC benchmarks can be used to choose the **instructions and formats** from a CISC ISA to implement in RISC.

The Intel x86 instruction set is an example of CISC with about 300 instructions. Compiling five programs from SPECint92 (the 1992 version of Cint) for x86, the table shows that 10 instruction types account for 95% of run time. These instructions and a few others required to build a complete machine language are sufficient for RISC.

Similar compilations lead to choices for data types, addressing modes, and other ISA features.

Slide 36 RISC Microprocessors

Instructions in the x86 ISA have various lengths from 1 byte to about 12 bytes. One complication in processing CISC is that the CPU must read the first byte of an instruction and then determine how many additional bytes it must fetch to read the full instruction. Also, the location of operand information in the instruction depends on the length and type of instruction.

To make RISC processors simple and fast, all instructions are the same length, with operand information in just a few possible locations.

Because the instruction set is reduced, the RISC processor is designed to implement instructions in dedicated hardware, without the need for microcode or a system bus. The system bus is a serious **bottleneck** on CISC machines because only one transfer can take place at one time — all instructions must finish atomically. In a RISC machine, all instructions move through a small number of hardware stages in the same direction. Several instructions can be in some stage of execution at one time, so that at least one instruction can finish on each clock cycle. This is called **instruction pipelining**. In a pipeline, IC instructions finish in about IC clock cycles, so that the CPI is about 1.

All processors today use RISC technology, in one of three ways:

1. Pure RISC, such as PowerPC, Sparc, MIPS, ARM, are implementations of a RISC ISA. Most of these are "second generation" RISC and are more complex than the original RISC ideas.
2. Explicitly parallel RISC (Intel Itanium, IBM mainframes).
3. Since the Pentium II, Intel x86 processors are designed as a RISC core with an adaptation layer (in silicon) that interprets x86 CISC machine language into a RISC machine language for execution.

Slide 37 Typical RISC ISA

The typical RISC ISA contains only two **data types** — integer and float. Additional data types defined in high level code are handled by the CPU as operations on integers. Depending on the processor, data length is 32 or 64 bits.

Unlike the segmented memory model used in x86, RISC processors generally use a **flat memory model** — a single 32 or 64 bit address that locates a byte of data in memory. Only one addressing mode is used for memory access, the form **`disp(Rn)`** defined as

```
MEM[REGS[Rn] + disp]
```

where

`Rn` is a register,

`disp` is a displacement (an immediate literal added to the register content).

The operation model is called **load-store** or **register-register** — all ALU operations are performed on source operands located in registers and results are written to a destination operand in a register. The operation y = x + 2 requires three instructions:

Load the value of x from memory to a register,

Add 2 to this register value and store the result to a new register,

Store the result from the new register to location y in memory.

RISC processors contain between **32 and 128 registers** for each data type. There are also special registers for OS operations, result flags, a read only register (always contain the value 0) and a write only register (writes to NULL register are legal but have no effect).

RISC processors perform only a few general **instruction types**.

Slide 38 Typical Instruction Encoding

To simplify decoding and implementation of instructions, RISC ISA has a small number of very similar **instruction encodings**. For example the Alpha 64-bit RISC processor uses a 32-bit instruction of 4 similar types: PALcode, Branch, Memory, Operate. Bit locations are numbered (right to left) from 0 to 31. In all encodings, bits 31 to 26 are the opcode, which identifies the operation to the CPU. Bits 0 to 25 describe the operands for the operation:

PALcode (Privileged Architecture Library) — 25 bits identify a specific OS system call

Branch — 5 bits identify a register to test and 21 bits provide a displacement to add to the current program counter (to implement the jump operation)

Memory — 5 bits identify a register operand, 5 bits identify a register containing the base memory address, and 16 bits provide an immediate literal to add to the register base memory address

Operate — register-register operations consist of 2 source registers and a function (a binary operation on the sources), while register-immediate operations consist of a register source, an immediate source, and a function

Slide 39 Simple RISC Physical Implementation

The early **RISC implementations** consisted of a few hardware stages. Instructions enter at the left and complete on the right, in the style of an assembly line. Each stage is connected to the next, with no central system bus.

The stages of this RISC implementation can be summarized as follows (a more detailed description will be given in the next presentation):

In the first stage, instructions are fetched from memory (instruction cache). The **Instruction Fetch** (IF) logic provides the address (program counter) of the next instruction to cache and the cache provides the encoded instruction.

In the second stage, **Instruction Decode** (ID) logic identifies the opcode, register operands, and the immediate. For each register operand, the ID stage reads the 5 bit register identifier and copies the operand value from the corresponding register to a buffer. At the end of the ID stage, the numerical values of the operands and the opcode are available.

In the **Execute** (EX) stage, the opcode and operands are sent to an ALU for calculation.

In Stage 4, the ALU result is committed to state. For **load and store operations**, **Data Memory Access** (MEM) reads or writes **data cache** according to the address calculated in the ALU. For ALU operations, **Write Back** (WB) writes the result to the destination register.

Each stage completes in 1 clock cycle, so that a single instruction requires 4 clock cycles to complete.

Slide 40 – 41 Pipelining — The RISC Advantage

The main **advantage of RISC** over CISC is the independence of the execution stages. This allows the **pipelining** of instructions — when an instruction moves from IF to ID, a new instruction can enter IF. The table (slide 40) shows the contents of each stage on each clock cycle. After a few clock cycles, each hardware stage is operating on a different instruction. Then, in the last stage, one instruction finishes in the last hardware stage on every clock cycle.

Looking at the instruction list, the table (slide 41) shows the location of each instruction by clock cycle. With 4 stages, instruction 1 finishes on CC4, I_2 finishes on CC5 and so on. Instruction number IC (the last one) will finish on clock cycle number IC + 3.

For a general pipeline instruction IC will finish on CC number IC + pipeline_length – 1 so that the CPI = (IC + pipeline_length – 1) / IC $\approx$ 1. This is called the ideal CPI, the CPI under ideal conditions, and written CPI^{ideal}.

The total run time is then just IC times the length of a clock cycle.

Slide 42 – 43 Pipeline Imbalance and Superpipelining

The first proposals for 4-stage RISC processors suffered from **pipeline imbalance**.

The clock period τ must be greater than the time required to perform the longest operation in any stage. Each basic operation — fetch, decode, execute, memory access, write back — take about the same amount of time, $\tau_{minimum}$. But the load instruction performs two operations: memory read and register write back. So the longest operation takes $\tau = 2 \times \tau_{minimum}$ and this determines the clock period.

But most instructions do not access data memory in stage 4 — load is about 25% of instructions. The method called **superpipelining** splits one hardware stage into two, in order to raise the clock speed. Splitting stage 4 into separate MEM (new stage 4) and WB (new stage 5) stages has two results:

1. Only load and store operations perform useful work in the MEM stage. All other instructions simply pass through unchanged.
2. The clock period is now set by $\tau' = \tau_{minimum} = \tau /2$. Superpipelining can double the clock rate $R = 1 / \tau$.

Slide 44 – 45 Pipeline Hazards

The pipeline cannot always run under ideal conditions. When on instruction is dependent on the result of a previous instruction, there is a **hazard** (race condition).

In the example, the ADD instruction writes to R1 and the SUB instruction reads from R1. In the uncorrected pipeline, the ADD result will not be available when the SUB requires it.

Hazards are divided into 3 types (described in detail on slides 46 – 64):

1. Structural — two instructions compete for access to a hardware resource
2. Data — an instruction result is not ready when a dependent instruction needs it
3. Control — the result of a branch instruction (the decision for a conditional branch, and the new instruction address) is not ready when the IF stage fetches the next instruction

To avoid the hazard (and the error it would cause), the dependent instruction must wait for the result before executing. This is called a **wait state** or a **pipeline stall**. The stall must be included in the CPI, by adding the clock cycles for the stall to the clock cycles used to do useful work.

The performance degradation can be measured by the ratio of CPI^{stall} = (stall clock cycles) / IC to the total $CPI = CPI^{ideal} + CPI^{stall} = 1 + CPI^{stall}$.

The main activity of hardware architects is to find ways to eliminate the effects of pipeline stalls on performance.

Slide 46 – 48 Structural Hazards

Structural stalls are not generally a problem in RISC. But they are simple to understand and are a useful illustration.

As an example, suppose that the cache were not **split** into separate instruction and data memories. This is called **unified cache** and is not used in level 1 cache (closest to the ALU).

The IF stage accesses cache on every clock cycle to fetch a new instruction. The MEM stage accesses memory whenever a load or store instruction reaches that stage. In unified cache the IF stage and the MEM stage would compete for access to the single memory chip on every data access.

The memory access cannot be delayed. If it were, it could not accept the next instruction from EX, and EX could not accept the next instruction from ID, and so on. So whenever a load or store reached MEM, the IF stage must be stopped for one clock cycle — no new instruction is fetched.

The first table shows I_1 running normally through the 5 stages of the pipeline. The next instruction is LW (load word). The LW and instructions I_2 and I_3 also run normally through the pipeline. But when LW reaches MEM in CC5, the stage IF must be stalled (no operation). This is symbolized by the letter ϕ. It means that no operation (**NOP**) is performed in IF in CC5. This creates a **bubble** — there is no new instruction to pass to ID on CC6, no new instruction to pass to EX on CC7, and so on. Notice that on CC9, no instruction completes — this is the real effect of the stall.

In the second table, the progress of the program is presented according to the program listing. Up to I_3 the program runs ideally. CC5 is shaded gray to indicate that no instruction enters IF. Instruction I_4 enters IF on CC6.

We calculate CPI^{stall} by writing the definition and analyzing the pieces. This requires statistics on the average program content. For RISC CPUs the usual instruction distribution is:

ALU	40%
Load	25%
Store	15%
Branch	20%

From the definition

$$CPI^{stall} = \frac{\textbf{stall cycles}}{\textbf{instructions}}$$

we count stall cycles as the number of cycles for one stall multiplied by the number of stalls.

$$CPI^{stall} = \frac{\textbf{stall cycles}}{\textbf{instructions}} = \frac{\textbf{stall cycles}}{\textbf{stall}} \times \frac{\textbf{stalls}}{\textbf{instruction}}$$

Since different instruction types can cause different stalls with a different number of stall cycles, we specify the types as

$$CPI^{stall} = \sum_{i=types} \left(\frac{\textbf{stall cycles}}{\textbf{stall}} \right)_i \times \left(\frac{\textbf{stalls}}{\textbf{instruction}} \right)_i \times \frac{\textbf{instructions}_i}{\textbf{instruction}}$$

This is now a sum over instruction types i of stall cycles per stall caused by type i, multiplied by the number of stalls per instruction caused by type i, multiplied by the number of instructions of type i per instruction in the program.

For the case of unified cache, types = {load, store}, each type causes exactly 1 stall, and each stall lasts exactly 1 clock cycle. So,

$$\mathrm{CPI}^{\mathrm{stall}} = \frac{1\ \text{stall cycle}}{\text{stall}} \times \frac{1\ \text{stall}}{\text{data memory load}} \times \frac{\mathrm{IC}^{\mathrm{load}}}{\mathrm{IC}} + \frac{1\ \text{cycle}}{\text{stall}} \times \frac{1\ \text{stall}}{\text{data memory store}} \times \frac{\mathrm{IC}^{\mathrm{store}}}{\mathrm{IC}}$$

Writing both load and store as memory access, this factors to

$$\mathrm{CPI}^{\mathrm{stall}} = \frac{1\ \text{stall cycle}}{\text{stall}} \times \frac{1\ \text{stall}}{\text{data memory access}} \times \left(\frac{\mathrm{IC}^{\mathrm{load}}}{\mathrm{IC}} + \frac{\mathrm{IC}^{\mathrm{store}}}{\mathrm{IC}} \right)$$

The factors IC_i / IC are the relative proportion of instruction in the program, so we can use the general statistics for RISC to write

$$\mathrm{CPI}^{\mathrm{stall}} = \frac{1\ \text{stall cycle}}{\text{stall}} \times \frac{1\ \text{stall}}{\text{data memory access}} \times \left(\frac{0.25\ \text{loads}}{\text{instruction}} + \frac{0.15\ \text{stores}}{\text{instruction}} \right) = 0.40 \frac{\text{stall cycles}}{\text{instruction}}$$

This means that 0.40 stall cycles must be counted for every instruction that runs, besides the cycles for useful work. Since $\mathrm{CPI}^{\mathrm{ideal}}$ = 1 for useful work,

$$\mathrm{CPI} = 1 + \mathrm{CPI}^{\mathrm{stall}} = 1.40$$

Compared to the ideal conditions, unified cache would cause

degradation = 0.40 / 1.4 = 29%

To prevent this performance degradation, most RISC CPUs use split cache.

Slide 49 **Data Hazards**

Data hazards can be described as hazards to the order of reading and writing. There are 4 possible orders for reading and writing. The hazards are classified according to their correct operation:

Read After Write (RAW)

Correct I_2 Reads register after I_1 Writes to it

Hazard I_2 Reads register before I_1 Writes to it
I_2 uses incorrect value

Write After Write (WAW)

Correct I_2 Writes to register after I_1 Writes to it

Hazard I_2 Writes to register before I_1 Writes to it
Incorrect value stays in register

Write After Read (WAR)

Correct I_2 Writes to register after I_1 Reads it

Hazard I_2 Writes to register before Reads I_1 it
I_1 uses incorrect value

Read After Read (RAR)

No hazard — Reads do not affect registers

RISC pipelines are designed to prevent WAW and WAR hazards. Instructions are processed in order and writing to registers is the last stage in instruction processing. Therefore, I_2 can never write before I_1 writes. Since I_1 reads before it writes, I_2 cannot write before I_1 reads.

RISC pipelines can suffer RAW hazards (I_2 reaches stage ID for reading before I_1 reaches stage WB to write its result). This hazard is prevented in various ways shown in presentation 2.

Slide 50 Control Hazards

Control hazards are caused by branch and interrupt conditions. The program counter (PC) holds the address of the next instruction. A branch instruction performs a jump by adding a displacement to the PC. The result of a branch instruction is the **target address**.

For a **conditional branch**, the instruction tests a certain condition.

If the condition is true, the branch is **taken**: target = PC + disp

If the condition is false, the branch is **not taken**: target = PC

Evaluation of the condition and the target address takes at least 1 clock cycle after the branch leaves stage IF. The control hazard is present if the IF stage fetches a new instruction before the branch evaluation is complete — IF may fetch the wrong instruction (the default next instruction in the program list).

The simplest fix is to flush (delete) the default next instruction from IF until the branch evaluation is complete. The more modern technique is to attempt to predict the branch result (discussed in presentation 4). The delay in processing the branch is called a **branch penalty**.

Slide 51 – 64 Exception Hazards

A related hazard is the **exception hazard**. An exception is any condition in which an instruction requires special handling, usually by an interrupt that triggers an interrupt service routine (**ISR**). Some interrupts are caused by a **trap**, a software instruction that produces a result similar to a **hardware interrupt**.

Just as in a branch instruction, an interrupt or trap changes the execution order by writing a new value (the address of the ISR) to the PC. When the ISR completes, the CPU must return to the instruction that was interrupted. The address of this interrupted instruction is the **return point** — it must be saved before jumping to the ISR and restored to the PC when the ISR completes.

Because several instructions are in various stages of execution in the pipeline, the exception can cause a hazard. How / where / when should the system interrupt the pipeline? Where is return-point, the PC for continuing the program after the exception?

A **precise exception** prevents the exception hazard by specifying which instructions complete atomically before jumping to the ISR, which instructions are flushed from the pipeline, and where is the return point.

Exceptions are raised at different stages of execution, depending on type. For example, an illegal instruction would raise an exception in ID when the opcode is not recognized. An overflow / underflow error would occur in EX. A page fault would occur in stage MEM when memory access points to an address not in main memory.

The table on slide 53 shows that I_2 causes a memory exception in stage MEM. This exception can be made precise by the following method:

1. Each instruction entering IF receives a status field and the PC from which it was fetched (extra bits for status and address are defined for the register holding the instruction).

2. When I_2 reaches stage MEM it raises an exception. A code identifying the exception is written to the status field.
3. The instruction (with status field and source PC = return point) is saved in a buffer, and I_2 to I_5 are flushed from the pipeline.
4. I_1 is allowed to complete.
5. The CPU jumps to the ISR associated with status code stored in I_2.
6. When the ISR completes the CPU jumps back to the return point by loading PC with the address stored with I_2.