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 הנדרש 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:

http://cs.hac.ac.il/staff/martin/Architecture

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, first introduced in the 1990s for the Pentium Pro and Pentium II processors. Over the years, Intel has introduced enhanced versions of the P6 micro-architecture: Netburst → Ivy Bridge → Haswell → Broadwell → Skylake → … → Golden Cove. The Xeon is used for high performance workstations and general application servers.

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.

The fastest supercomputer is the FRONTIER - HPE CRAY EX235A, which uses a 64 core AMD CPU.

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 / τ 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),

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

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

$n =$ Number of processing units

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

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 2017. (Results from the 2006 version must be used for older CPUs never tested on the 2017 version).

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 Some Cint2017 Results

The table on **slide 30** shows some representative results for CPUs on Cint2017 (SPECint). 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.

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 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.

Presentation 2

DLX: A Simplified RISC Model

באמצע שנות ה- 1980 החוקרים John L. Hennessy (סטנפורד) ו- David A. Patterson (ברקלי) הובילו את הפיתוח של גישת RISC בארכיטקטורה. אחד המעבדים הראשונים בגישה הזאת היה מוצר בשם MIPS2000. החוקרים גם שיתפו פעולה בספר Computer Architecture: A Quantitative Approach ולצורך כך כתבו ISA בשם DLX ככלי הוראה מבוסס על משפחת MIPS. המודל מהווה בסיס מסודר ללימוד מושגי RISC, ומתאים להצגת פיתוחים עכשוויים שלא קיימים במודל המקורי. מטרת המצגת היא לתאר את ה- ISA והמימוש הבסיסי של DLX. למבוא יותר הדרגתי ומפורט:

http://cs.hac.ac.il/staff/martin/Architecture

DLX Architecture
Speeding-up DLX

Slide 2 – 4 **DLX Pipeline**

The DLX pipeline contains the "classic" 5 stages of first generation RISC: IF, ID, EX, MEM, WB. Instructions are executed in an integer ALU or floating point unit (FPU).

DLX is a 32-bit machine using 32-bit memory addresses and operating on 32-bit integers and floats (single precision).

The CPU contains 32 integer registers R0 – R31 and 32 FP registers F0 – F31. Double precision FP numbers are 64 bits and stored in a pair of FP registers. All integer registers are identical except R0 which is read-only (contains 0). A write to R0 is legal but has no effect.

The blue lines in the pipeline diagram perform **forwarding** (also called bypass) — intermediate results of can be fed backward to instructions that need them to prevent RAW hazards (explained in detail on slides 16 – 29).

Slide 5 **DLX Instruction Formats**

The DLX **instruction format** is very similar to the Alpha instruction (presentation 1 slide 38). Bits are numbered from left to right.

Bits 0 – 5 are the opcode (identify the operation).

For **Type J** instructions (unconditional jump), bits 6 – 31 provide an offset (displacement) to add to PC to implement the branch.

In **Type R** instructions (reg-reg ALU), 3 registers are named in bits, 6 – 10, 11 – 15, and 16 – 20. With 5 bits per register, this method can name 32 registers, 0 – 31. The first registers are sources and the third is the destination.

The **Type I** format is used for instructions requiring an **immediate** literal value: load, store, ALU with immediate, and branch.

Slide 6 – 9 **DLX Instruction Set**

The DLX has a small instruction set, divided into **transfers**, **integer ALU**, **FP**, and **control**. Each instruction is defined in the formal language described in presentation 1. For example, the first instruction is the Load Word instruction:

$$\mathbf{LW\ R1,30(R2)} \qquad \mathbf{R1} \leftarrow_{32} \mathbf{MEM[30+REGS[R2]}$$

The operation adds 30 to the 32-bit value in register **R2**, uses that value as an address in memory, and loads the 32-bit value at that address (4 consecutive bytes) to **R1**. Store Word is similar but copies the value from register to memory. Load Float loads the value to a floating point register instead of an integer register.

A subscript after a value refers to a single bit from the value (numbered right to left).

A superscript after a value indicates repetition of the bit value.

The symbol ## is concatenation (שרשור).

The instruction Load Byte reads one byte from a memory address, and copies the high order bit (the sign of a signed integer) 24 times to the left of the byte. The result is a 32-bit representation of the signed value of the byte.

For example, the signed value –2 (decimal) = 0xFE (hexadecimal byte) is expanded to the 32-bit value 0xFFFFFFFE representing –2.

The instruction LBU is Load Byte Unsigned — instead of the upper bit, the instruction fills in 24 leading zeros for the unsigned valued.

The instructions MOVFP2I and MOVI2FP transfer literals unchanged between register sets. There is no conversion of float to int or int to float.

The **integer ALU** register-register instructions are ADD, SUB, MULT, DIV, AND, OR, and XOR. The forms ADDI, SUBI, ANDI, ORI, XORI take one immediate source operand. There are 6 comparison instructions that set a register based on the comparison of two other registers.

The **floating point** instructions are ADD, SUB, MULT, DIV, and the 6 comparisons. The register StatFP is a 1-bit flag used to hold the comparison results.

There are 7 **control instructions**:

J offset is an unconditional jump — it adds the offset to the PC.

JAL is similar but it saves the default PC (a return point) in register R31.

The saved return point can be accessed using the **JL reg** instruction that loads PC with the value saved in the register.

JALR reg, offset is similar to JAL but allows the choice of register for saving the return point.

BEQZ reg, offset is **B**ranch on **Eq**ual **Z**ero — it adds the offset to PC is the register contains 0. **BNEZ** is similar but jumps when the register is non-zero. These instructions implement the **conditional branches** used in high level control blocks — if, else, for, while, case, ...

TRAP N is a software interrupt. In some implementations, N is an index to a table of ISR addresses, while in others it is the address of the ISR.

Slide 10 Programming in DLX Assembly

The simple C program

```
main()
{
      int i,j;
      for (i = 0; i < 10; i++){
            j = 2 * i;
      }
}
```

written for DLX appears

```
          ADDI R1, R0, #0        ; i = R1 <-- 0
          ADDI R10, R0, #0A      ; R10 <-- 10
start:    SGE R11, R1, R10       ; R11 <-- 1 iff R1 >= R10 = 10
          BNEZ R11, stop         ; jump to label stop if R1 >= 10
          ADD R2, R1, R1         ; R2 <-- R1 * 2
          ADDI R1, R1, #1        ; R1++
          J start                ; jump to start
stop:     SW -2(R13), R2         ; store j <-- R2
                                 ; R13 = base pointer for variables
          JR R31                 ; return to calling function
```

Slide 11 – 14 DLX Implementation (Integer Pipeline)

The pipeline consists of 5 stages **IF**, **ID**, **EX**, **MEM**, **WB** separated by **4 stage buffers**.

The buffers are named for the stages they separate: **IF/ID** (between IF and ID), **ID/EX**, **EX/MEM**, **MEM/WB**. The stage buffers contain temporary registers that hold intermediate values during instruction execution. The temporary registers are defined on slide 12. It is useful to think of the stage buffer as a **struct** in a C program, with each register as one member. The formal specification on slides 13 – 14 describes the operations in each stage.

Each stage buffer operates as an **edge triggered latch** (דלגלג מדורבן מעבר). The input to the latch does not affect the contents except on a falling transition of the clock signal **CLK**. At the precise moment that CLK goes high-to-low, the flip-flop samples and stores the input, and provides the new stored value as an output. This produces **synchronous transfer** of data (synchronized with the clock signal CLK).

IF stage

On the CLK transition, **PC** updates. The cond flag is only 1 if there is a conditional branch taken (the branch condition is true). If cond = 1 then the next **PC** is the computed address taken from the adder in stage ID (this case is described in more detail below). Otherwise the **PC** receives the default address of the **fall-through instruction** (the next instruction in the listing) found by adding 4 bytes per instruction.

$$\mathbf{PC} \leftarrow \begin{cases} \mathtt{PC + 4,\ cond = 0} \\ \mathtt{ID/EX.NNPC,\ cond = 1} \end{cases}$$

This means that when **PC** updates, it holds its new value for a full clock cycle τ with the value of **PC + 4** waiting as a new input. The new input becomes the updated **PC** on the following CLK transition.

The same value loaded to **PC** is saved in the register **NPC** (next **PC**) in the stage buffer IF/ID.

$$\mathbf{IF/ID.NPC} \leftarrow \begin{cases} \mathbf{PC + 4,\ cond = 0} \\ \mathbf{ID/EX.NNPC,\ cond = 1} \end{cases}$$

The current **PC** (the value stored in the latch) is used as a memory address. The 32-bit instruction at that address is loaded into the instruction register IR.

$$\mathbf{IF/ID.IR} \leftarrow \mathbf{Mem[PC]}$$

ID stage

In the decode stage several temporary registers receive values:

A receives the content of rs1 (type R) or rs (type I):

$$\mathbf{ID/EX.A} \leftarrow \mathbf{Reg[IF/ID.IR_{6\text{-}10}]}$$

B receives the content of rs2 (type R) or rd (type I):

$$\mathbf{ID/EX.B} \leftarrow \mathbf{Reg[IF/ID.IR_{11\text{-}15}]}$$

I receives the last 16 bits in the instruction, extended as a signed number to 32 bits. This is either meaningless (type R) or is the immediate literal in the instruction (type I):

$$\mathbf{ID/EX.I} \leftarrow \mathbf{(IR_{16})^{16}\ \#\#\ IF/ID.IR_{16\text{-}31}}$$

IR receives the instruction encoding:

$$\mathbf{ID/EX.IR} \leftarrow \mathbf{IF/ID.IR}$$

NNPC receives the value NPC + immediate. If there is a taken branch, NNPC holds the value of the computed address for the target:

$$\mathbf{ID/EX.NNPC} \leftarrow \mathbf{IF/ID.NPC + (IR_{16})^{16}\ \#\#\ IF/ID.IR_{16\text{-}31}}$$

cond receives the condition flag, which is 1 if the value in register A is 0:

$$\mathbf{ID/EX.cond} \leftarrow \mathbf{(Reg[IF/ID.IR_{6\text{-}10}] == 0)}$$

EX stage

In EX the ALU receives two operands from two multiplexors:

$$\mathbf{EX/MEM.ALU_{OUT}} \leftarrow \begin{cases} \mathbf{ID/EX.A\ function\ ID/EX.B} & \mathbf{(R\text{-}ALU)} \\ \mathbf{ID/EX.A\ op\ ID/EX.I} & \mathbf{(I\text{-}ALU,\ Memory)} \end{cases}$$

$$\mathbf{Forwarding:\ EX/MEM.ALU_{OUT}\ or\ MEM/WB.ALU_{OUT}\ or\ MEM/WB.LMD}$$
$$\mathbf{substituted\ for\ A\ or\ B}$$

For type R instructions, the top multiplexor provides the value from temporary register **A** and the bottom multiplexor provides the value from temporary register **B**.

For type I instructions, the top multiplexor provides the value from temporary register **A** and the bottom multiplexor provides the value from temporary register **I**.

The forwarding mechanism permits substitutions for either **A** or **B**. This mechanism is explained in slides 21 – 25.

The ALU performs the instruction specified in **ID/EX.IR** and writes the result to $\mathbf{ALU_{out}}$.

Also, the values in registers **B** and **IR** are copied from the stage buffer at left to the stage buffer at right:

$$EX/MEM.B \leftarrow ID/EX.B$$
$$EX/MEM.IR \leftarrow ID/EX.IR$$

MEM stage

On load operations, the value in ALU_{out} is used as the memory read address — the data from memory is written to temporary register **LMD**.

On store operations, the value of register **B** is written to memory at address ALU_{out} . The forwarding mechanism permits substitution of $MEM/WB.ALU_{out}$ for **B**.

$$MEM/WB.LMD \leftarrow Mem[EX/MEM.ALU_{OUT}] \ (Load)$$
$$Mem[EX/MEM.ALU_{OUT}] \leftarrow EX/MEM.B \ (Store)$$
$$Fowarding: MEM/WB.ALU_{OUT} \ substituted \ for \ B$$

For other instruction types, there is no memory access.

The values in ALU_{out} and IR are copied from the stage buffer at left to the stage buffer at right.

$$MEM/WB.ALU_{OUT} \leftarrow EX/MEM.ALU_{OUT}$$
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

WB stage

ALU and load operations write results to registers according to the destination location **rd** (bits 11 – 15 for type I, or bits 16 – 20 for type R).

$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} \ (ALU\text{-}I) \\ MEM/WB.LMD \ (Load) \end{cases}$$
$$Reg[MEM/WB.IR_{16\text{-}20}] \leftarrow MEM/WB.ALU_{OUT} \ (ALU\text{-}R)$$

General features of the implementation

The operations in stages IF and ID are identical for every instruction, simplifying the pipeline. This method involves some unnecessary work: For type R instructions, writing to register **I** has no meaning. For type I ALU instructions, writing to register **B** in ID is useless (it receives the old value of **rd**). But this work requires no extra processing time and is harmless. It would require more effort to prevent these actions than to ignore them.

The choice of source operands and operation for the ALU in stage EX permits the CPU to perform any defined operation. Comparing the definitions on slide 5, the implementations are:

Type R ALU:
```
rd ← ALU_function (rs1, rs2)
rd ← ALU_function (A, B)
```

Type I ALU:
```
rd ← ALU_operation (rs, imm)
rd ← ALU_operation (A, I)
```

Load:
```
rd ← imm(rs)
rd ← MEM[A+I]
```

Store:
```
imm(rs) ← rd
MEM[A+I] ← B
```

Branch:
```
if (rs == 0) {PC ← PC + imm}
if (A == 0) {PC ← NPC + I}
```

Slide 15 – 19 Implementation Examples

Each example illustrates:

1. The **instruction** format in assembly language.
2. The **operation** performed by the instruction as a formal specification.
3. The **encoding** of the instruction. The opcodes are entered by name — in the actual instruction these are 6-bit binary numbers representing the operation.
4. The operations performed in **Hardware Stage 1**. These are identical for every instruction.
5. The operations performed in **Hardware Stage 2**. These are identical for every instruction. Some operations are not used by stages 3 – 5.
6. The operations performed in **Hardware Stage 3**. These actions depend on the particular instruction.
7. The operations performed in **Hardware Stage 4**. These actions depend on the particular instruction — some instructions do no work in this stage.
8. The operations performed in **Hardware Stage 5**. These actions depend on the particular instruction — some instructions do no work in this stage.

Slide 20 DLX Integer Pipeline Statistics

The **instruction distribution** is found by compiling the programs in SPEC Cint into DLX assembly language and sorting the instructions. The result is

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

Data dependencies between instructions determine RAW hazard statistics. If I_N is an ALU instruction, then in 50% of cases,

One **source** operand of I_N is the **destination** operand of instruction I_{N-1}

The instruction I_{N-1} could be an ALU or load instruction

The DLX must treat RAW hazards for 50% of 40% = 20% of all instructions.

Slide 21 Data Hazards in DLX Integer Pipeline

The DLX must treat RAW hazards for 50% of 40% = 20% of all instructions.

WAW and WAR hazards cannot occur in the DLX

Slide 22 – 23 ALU-ALU RAW Dependencies

The program on slide 22 has 3 data dependencies. Register R1 is the destination of I_1 and a source for $I_2 – I_4$.

I_1 updates R1 in stage WB in CC5. Any instruction that enters stage ID to read R1 before CC5 will read the old value and cause an error.

Unless the hazard is prevented, the table shows that:

I_2 will enter ID in CC3 causing an error

I_3 will enter ID in CC4 causing an error

I_4 enters stage ID in CC5. This does not cause an error, because I_1 in stage WB updates R1 at the **beginning of CC5**, but the reading of R1 for I_4 is not latched into the temporary register **A** or **B** until the next CLK transition at the **end of CC5**. (For those interested, this is shown in detail in slide 23.)

Slide 24 – 25 Pipeline Stall to Avoid RAW Hazard

The hazard can be prevented by stalling the pipeline. I_2 is held in stage IF for 2 clock cycles and enters ID when I_1 performs the WB in CC5. Meanwhile 2 NOP bubbles are placed in the pipeline (as explained in presentation 1 slide 47).

In the instruction view, the red clock cycles indicate the 2 CC pipeline stall, with I_2 held in IF.

The effect on performance is found from

CPI^{stall} = (CC per stall) × (stalls per ALU instruction) × (ALU instructions per instruction)

Since the stall is 2 clock cycles, 50% of ALU instructions (on average) have a data dependency, and 40% of instructions are ALU instructions, CPI^{stall} = 0.40 causing a 29% performance degradation.

Slide 26 – 27 Forwarding or Bypass

Without the pipeline stall, I_2 requires the new value of R1 in stage EX in CC4. During CC4, the new value has not been written to R1, but is **saved** in the stage buffer **`EX/MEM.ALUout`**.

The method called **forwarding** (bypass) is:

1. Allow I_2 to read the old value of R1 in stage ID in CC3 (no pipeline stall).
2. When I_2 reaches stage EX in CC4, do not use the old value. Instead, use the temporary value from **`EX/MEM.ALUout`**.

The green line fed back from EX/MEM to the EX stage represents this forwarding.

Instruction I_3 receives similar treatment. I_3 enters ID in CC4 and reads the old value of R1. When I_3 enters EX in CC6, the temporary value (not yet saved to R1) is saved in stage buffer **`MEM/WB.ALUout`** and can be fed back to the EX stage. The purple line fed back from MEM /WB to the EX stage represents this forwarding.

On the table, the forwarding is indicated by the arrows showing transfer of the temporary result directly to the instruction in stage EX.

In the instruction view, arrows again show transfer of the temporary result directly to the instruction in stage EX.

This way, forwarding removes the need for a pipeline stall.

Slide 28 – 30 Load-ALU RAW Dependencies

The program on slide 28 is similar to the program on slide 22, except that now I_1 is a **load** instruction. The forwarding method can be applied again, but it does not solve the whole problem. The **LW** instruction calculates the memory address in CC3 but only reads from memory in CC4. So, instruct3.ion I_2 must be held in stage ID for 1 extra CC — it enters ID on CC5 and the old value of R1 (that it read in ID) is replaced by the new temporary value saved in register **`MEM/WB.LMD`**.

Comparing the tables on slides 26 and 29,

> The ALU-ALU dependency is handled by copying the value for R1 from EX in CC3 down to EX in CC4.
>
> The Load-ALU dependency is handled by copying the value for R1 from MEM in CC4 down and back to EX in CC5.

The effect on performance is found from

$$CPI^{stall} = (\text{CC per stall}) \times (\text{stalls per ALU instruction}) \times (\text{ALU instructions per instruction})$$

Since the stall is 1 clock cycles, 50% of ALU instructions (on average) have a data dependency, and 25% of instructions are load instructions, $CPI^{stall} = 0.125$ causing an 11% performance degradation. The performance in this situation can be improved in the compiler (slide 35).

Slide 31 ALU-Store RAW Dependencies

The program on slide 30 has another data dependency — the value of R1 is not updated in CC3 when the store instruction SW reads it for writing to memory.

Forwarding can also be used in this case. The temporary result saved in **`MEM/WB.ALU`** replaces the register **B** that holds the old value of R1 read in stage ID. This prevents the hazard without a stall.

Slide 32 – 33 DLX Control Hazard

In order to minimize the control hazard, the DLX uses a policy called **predict not taken**:

1. The branch instruction is evaluated in stage ID. In the same clock cycle, the CPU automatically fetches the default instruction — the next instruction in the program listing (known as the **fall-through** instruction). In the example on slide 31 this occurs on CC2.
2. If the branch instruction evaluates as **not taken** (the condition is false and there is no jump), then the fall-through instruction continues. There is no stall.
3. If the branch instruction evaluates as **taken** (the condition is true and program control jumps), then the fall-through instruction is cancelled and the **target** instruction is loaded from the address calculated from the branch instruction. There is a stall of 1 clock cycle, because the cancelled fall-through moves through the pipeline as a NOP bubble.

The effect on performance is found from

$$CPI^{stall} = (\text{CC per stall}) \times (\text{stalls per ALU instruction}) \times (\text{ALU instructions per instruction})$$

Statistics show that of all branch instructions (20% of all instructions), 2/3 are taken and 1/3 are not taken.

Since the stall is 1 clock cycles, 67% of branch instructions are taken (causing a stall), and 20% of instructions are branch instructions, $CPI^{stall} = 0.13$ causing a 12% performance degradation. The control hazard can be improved by branch prediction (presentation 4).

Slide 34 Other Stalls

Some data dependency stalls are too complex to repair with forwarding:

1. ALU followed by a branch instruction conditioned on the ALU result
2. ALU followed by an independent instruction followed by a store dependent on the ALU result.

In these cases, a stall must occur until the dependent instruction can read the register directly in stage ID while the prior instruction updates the register in WB.

Slide 35 – 36 Rescheduling

Rescheduling is a compiler optimization that can improve performance by preventing hazards.

The program on the left side of slide 35 suffers 3 Load-ALU stalls (1 CC each) and an ALU-Branch stall (2 CC).

Without affecting the program outcome, the compiler can move instructions in the listing so that instruction results are ready when the next dependent instruction needs them. This is called **hiding the latency**.

On the left, the instruction in row 10 is **`SUBI R1, R1, #4`**. In this program R1 is an index for loop iterations and is checked in row 11 (**BNEZ**). On the right, this index is updated in row 2 instead of row 11. Since R1 is also used to index memory accesses (LW and SW), the addresses must be adjusted to account for the change in program order.

On the left, the program performs 2 loads and then an ALU operation. On the right all the loads are performed first, using additional registers for all the loaded data. Now, all the loads are available in registers when the ALU instructions need them. For example, in row 3, LW R2 writes R2 in CC7 and is first used by ADD in row 7 that enters ID in CC8.

The execution of the two programs is shown on slide 36. Each iteration of the loop suffers 5 stall cycles in the original version (plus the control stall after BNEZ). The rescheduled version has no stalls (except the control stall). The loop runs 256 times (100 hexadecimal) and so rescheduling saves 5 × 256 = 1280 clock cycles.

Rescheduling techniques are discussed in detail in presentation 3.

Slide 37 DLX Memory Hierarchy

Slide 37 shows the relationships among memory units in the DLX. The **Instruction Memory** in stage IF and the **Data Memory** in stage MEM of the pipeline are **level 1 (L1) cache**. The L1 cache memories are connected to the cache controller. When a memory address (PC or data address formed in the ALU) is not located in L1 (L1 cache miss), the controller simultaneously accesses level 2 (L2) cache in the DLX package and Main Memory (through the external I/O controller). If the memory location is found in L2 cache (L2 cache hit), the Main Memory access is cancelled. If the location is not found in L2 cache (L2 cache miss), then the location is copied to L2 and to L1. Cache organization is discussed in detail in presentation 4.

Slide 38 – 42 MIPS Architecture

The DLX is a pedagogical abstraction of the commercial **MIPS ISA**. The ISA defines registers and the instruction set for a family of MIPS implementations. Each MIPS core defines the device-dependent implementation details, including pipeline organization, I/O organization, control registers and so on. The **MIPS32** ISA defines a 32-bit RISC CPU similar to the DLX. The **MIPS64** ISA defines a 64-bit RISC CPU. The 32 and 64 bit versions have the same instruction set, with binary compatible machine instructions of 32-bit length. MIPS core designs are typically licensed to OEMs (original equipment manufacturers) that implement the design in an embedded system (a microprocessor based device that is not a general purpose computer).

The register set and instruction format for MIPS is very similar to DLX. An important difference is the set of coprocessors defined for MIPS. For simplicity, certain basic functions expected in a general purpose computer are moved to **coprocessors**, which are not required on simple embedded systems. The ISA defines special instructions that access the coprocessors through the following hardware interfaces:

> CP0 is reserved for virtual memory support and exception handling. It is used to translate virtual addresses (as seen by software) into physical addresses (required by the memory I/O system). It also controls the cache subsystem, handles switches between kernel, supervisor, and user states, and manages exceptions, diagnostic control, and error recovery.
>
> CP1 is used for the interface with an external FPU on older MIPS32 cores.
>
> CP2 is used to interface specialized, device-specific hardware.
>
> CP3 is used for the interface with an external FPU on MIPS644 and newer MIPS32 cores.

Slide 41 shows some of the **MIPS instructions** not defined for the DLX. In addition to the coprocessor instructions, the MIPS defines shift and rotate operations, sync operations to support parallel programming (discussed in presentation 7), predefined systems calls (similar to Alpha PALcode, presentation 1, slide 38), and cache prefetch (discussed in presentation 4). Other instructions include convenient variations, such as Set on Less Than Immediate, Branch on greater or equal zero, and Branch on less than or equal zero.

Presentation 3

Directions for Improving Performance

מעבדים בסגנון RISC הצליחו לשפר ביצועים בצורה משמעותית ביחס למעבדים ישנים יותר. אם בשנת 1985 מעבד CISC פעל בקצב שעון של 20 MHz עם CPI בסביבות 8, בשנת 2000 מעבד RISC פעל בקצב שעון של 4 GHz עם CPI בסביבות 1. ביחד זה שיפור של פי 1600 בזמן ריצה. כדי להמשיך לשפר את הביצועים חייבים להשיג מידה משמעותית של מקבילות, גם ברמת הפקודות וגם ברמת הפתיל (thread). הפרק הזה עוסק בשיטות להשיג מקבילות רבה ברמת הפקודות.

Slide 2 **General Outlook**

The **fundamental parameters** of performance follow from the equation $T = CPI \times IC \times \tau$. Reducing any of the three factors improves run time. But this desire confronts certain technological realities:

1. IC grows as software becomes more complex. Not only are programs getting larger, but software development favors fast development over efficient run time. The use of advanced programming techniques, such as middleware, OOP, design patterns, and virtualization, introduce additional layers of code translation that contribute to IC at run time.
2. The clock period τ is reaching physical limits. The time required for an electrical signal to cross a 3 cm wide chip is 3 cm / (3×10^{10} cm/sec) = 10^{-10} seconds, where 3×10^{10} cm/sec is the speed of light. This is the period of a 10 GHz clock, since $(10\ GHz)^{-1} = 10^{-10}$ seconds.
3. For a RISC processor, $CPI^{ideal} = 1$. A sequential processor cannot improve on this.
4. CPI^{stall} is affected by data dependencies, memory latency such as cache miss, and control hazards in branches.

The areas for possible improvement are:

1. Achieving $CPI^{ideal} < 1$ with instruction and thread level parallelism
2. Reducing instruction dependency stalls to lower $CPI^{stall}_{data\ dependency}$
3. Reducing cache latency to lower $CPI^{stall}_{cache\ miss}$
4. Reducing branch stalls to lower $CPI^{stall}_{branch\ penalty}$

This chapter discusses **Instruction Level Parallelism** (ILP) and reduction of $CPI^{stall}_{data\ dependency}$.

Slide 3 **Processing Parallelism**

Processing can be made parallel in two ways:

1. **Instruction Level Parallelism** (ILP) — parallel execution of instructions in a single thread
2. **Thread Level Parallelism** (TLP) — parallel execution of a single program by multithreading

ILP includes

1. **Pipelining** — multiple sequential instructions from a single thread in various stages of execution at any given time (as in the DLX)

2. **Superscalar** processing — multiple sequential instructions from a single thread executing in parallel on multiple execution units of a single CPU

Multithreading does not imply parallelism. **Concurrency** describes multiple threads that are defined in software to run independently. Concurrent threads can run in **parallel** (on separate cores) or by **interleaving** (threads executing by taking alternate turns on the same CPU core).

TLP refers to **multiprocessing** — multiple independent processor cores running independent threads.

Slide 4 – 7 Flynn Taxonomy

The **Flynn Taxonomy** enumerates the 4 possible ways of combining data and instruction streams. Each stream may come from a single source or multiple sources:

1. Single Instruction Single Data (SISD) — a single operation on a single pair of operands
2. Single Instruction Multiple Data (SIMD) — single operation on multiple operands, such as a vector processor (slide 8)
3. Multiple Instruction Single Data (MISD) — multiple operations on single operand, such as the IBM Watson IA that applies multiple algorithms to the same data
4. Multiple Instruction Multiple Data (MIMD) — multiple operations on multiple operands, such as implemented in a multiprocessor, multithreaded processor, and cluster computer.

The DLX discussed in presentation 2 is an example of SISD. This chapter focuses on ILP, which includes SIMD and superscalar methods. Approaches to MIMD will be discussed in presentations 7 – 9.

Slide 8 – 11 SIMD — Vector ALU

A **vector processor** receives multiple pairs of operands per clock cycle and performs the same operation on each pair. In the example on slide 8, a pair of vector registers (each 4 operands wide) receives 4 pairs of operands, and performs 4 addition operations in parallel.

Slide 9 shows the DLX updated to include a vector ALU and parallel registers P1, P2, ... , P31. The code loads 4 operands in parallel, loads another 4 operands in parallel, and then adds 4 pairs of operands. The execution follows the regular pattern for DLX (including the Load-ALU stall).

Some practical examples of a vector ALU include the Intel MMX, SSE, and AVX ALUs, and the AltiVec ALU in the IBM Power ISA. Compilers for these ISAs support vectorization for appropriate data structures.

The sample program on slide 11 shows vectorization of a routine that adds the 100 elements of an array. The standard SISD implementation performs 100 loop iterations. The SIMD vector implementation operates on 4 consecutive elements in each of 25 iterations — this produces 4 partial sums. The last step is called horizontal add — it produces the total from the 4 partial sums.

Slide 12 – 13 Optimizing Data Structures

A compiler requires appropriate **data structures** to support the vector processor.

Consider a 3D graphic object described by 200 vertices and 200 color pixels. A programmer might be tempted to describe this object in an array of structures (AoS):

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

with 200 elements that describe one 3D vertex (x,y,z) and one pixel (r,g,b). But the vector processor cannot access successive operands for parallel execution. The first 4 operands are **`x[0], y[0], z[0], r[0]`**, which cannot be meaningfully processed with the next 4 operands, **`g[0], b[0], x[1], y[1]`**.

The **hybrid data structure** on slide 13 first separates the vertices and the pixels into separate structures — these can be processed by separate threads. Each structure is an array of 50 structs — each struct holds 4 consecutive values of one coordinate or color. The vector processor can then operate on 4 **x** values at one time: **`x[0], x[1], x[2], x[3]`**, followed by 4 **y** values, and so on.

Slide 14 – 16 Graphics Processing Unit

The **Graphics Processing Unit** (GPU) is a specialized graphics processor usually located separately from the CPU (on the computer's main board or a separate graphics card). The CPU can offload graphics operations to the GPU as an auxiliary processor.

The GPU is a complex system beyond the scope of this course (and requires a background in graphics processing to understand its fundamental terms). This section is intended only to introduce the subject in a basic way.

A typical x86 CPU and a typical Nvidia GPU are about the same size, approximately 3 billion transistors. The main difference is that the CPU is optimized for the efficient processing of sequential code (including different types of ALU and significant resources for cache memory), while the GPU consists of a very large number of identical execution cores.

GPUs have undergone a three-step evolution.

In **Step 1**, the execution core was a **hardwired graphics pipeline**, designed to perform the standard operations required to render a 3D object described by vertices into a graphic image. The stages were:

1. Geometry —transformation of a set of 3D **vertices** (x, y, z) into a description by 2D **triangles**, the area enclosed by sets of three vertices.
2. Rasterization —transformation of the **triangles** into **fragments**, a description in terms of computer screen coordinates (**raster**) of a set of points with a single color.
3. Fragment — color interpolation between neighboring fragments to produce a texture made of colored fragments.
4. Composition — transformation of the textures into an image, a set of pixel coordinates and colors appropriate for display on the screen

In this model, the application provides the GPU with 3D data, and the GPU renders the data into a graphics image for display.

In **Step 2**, the hardwired graphics pipeline was replaced by a **programmable pipeline**, in which the application provides the GPU with a command to perform on a set of vertices. The command allows application control over the standard pipeline stages found in the hardwired pipeline.

In **Step 3**, the programmable pipeline was replaced by a **shader**, a completely programmable processor core with hardware optimized to support graphics operations.

On the simple repetitive operations appropriate to a GPU, its performance is about 4 times faster than a standard single core CPU of its era.

The maximum **theoretical performance** of the vector processor on a single core CPU (the SSE on a 3.7 GHz Pentium 4) can be found as:

1 pipeline $\times$ 4-wide SIMD $\times$ 3.7 GHz = 14.8 GFLOPS (Giga Floating Point Operations Per Second).

The maximum **theoretical performance** of the Nvidia GeForce 6800 Ultra can be found as:

16 pipelines $\times$ 8-wide SIMD $\times$ 520 MHz = 63 GFLOPS

Slide 16 shows the structure of an Nvidia General Purpose GPU (GPGPU). The **shader** element receives:

1. Graphics data from graphics cache that accesses general memory,
2. Instructions from the application — instructions are processed using a library of graphics routines (implemented in silicon).

The Nvidia GeForce 8800 GPU (2005) consists of 8 thread processors with 16 shaders/processor, for a total of 128 shaders.

The Nvidia Kepler GPU (2012) consists of 15 SMX streaming multiprocessors with 192 shaders per SMX, for a total of 2880 shaders.

The Nvidia Volta GPU (2017) consists of 5120 shaders.

Slide 17 Superscalar Pipeline Models for DLX

A superscalar provides explicit ILP by initiating multiple instructions at the same time (in the same clock cycle). There are two possible approaches:

1. **Replicate the entire pipeline** and launch a pair of instructions on each CC — one instruction in each pipeline. Each pipeline operates in the same way as on a single pipeline CPU. This approach is difficult to implement and was only attempted on one mass-market commercial processor — the Intel Pentium in 1993.
2. **Replicate the execution units** (EUs) and send instructions to EUs as needed. In this model, the IF and ID stages operate quickly to fetch and decode multiple instructions per clock cycle. The decoded instructions are stored in an **instruction pool**, a large register holding the instructions along with their source operands (read from the registers). A controller sends the pooled instructions to EUs as needed.

Slide 18 – 20 Superscaling in Intel Pentium

The Pentium included two pipelines of the type designed for the Intel 486 CPU. The 486 pipeline stages are somewhat different from a RISC pipeline:

1. Instruction Fetch (IF)
2. Stage-1 Decode (D1) — identify the operation, read register sources and calculate the register part of the address for memory sources
3. Stage-2 Decode (D2) — calculate the physical data address and generate the microcode required to execute the instruction
4. Execution (EX) — integer ALU operations, FPU operations, and data memory accesses
5. Register Writeback (WB) — update the destination registers

The ideal pipeline in the 486 is shown in the upper table on slide 19 and the ideal pipeline in the Pentium is shown in the lower table. In the ideal case, the Pentium has $CPI^{ideal} = 0.5$.

The ideal case for multiple pipelines is difficult to achieve, for two reasons:

1. Dependent instructions cannot be paired. In the program example on slide 20, I_0 depends on the result of I_1 and so they cannot be executed in parallel. Instead, I_1 is stalled in stage ID1 for an extra CC until the result for EAX is available.
2. If the first instruction of a pair is stalled, the second instruction must stall to prevent it writing before the first (WAW hazard). In the Pentium, some registers (such as the stack pointer ESP) are not forwarded, causing implicit hazards on stack operations (push/pop).

Slide 21 Instruction Pool Example

A superscalar with replicated EUs is simpler to implement. This strategy was chosen for Intel x86 processors after the Pentium (beginning with the Pentium Pro, Pentium II, and used to this day).

The example shows a superscalar version of the DLX pipeline and a program fragment. In this example:

1. I_1 — I_3 have no dependencies. I_1 and I_2 are ALU instructions and I_3 is a Load. All three can be run in CC1 — I_1 and I_2 in two EX units and I_3 in a MEM unit.
2. Each of I_4 — I_6 is dependent on the previous instruction and must be run sequentially in CC2 — CC4.
3. I_7 is independent of prior instructions — it could be rescheduled to run in a MEM unit in any of CC1 — CC3.

Most programs consist of sequential steps and contain data dependencies, as seen in I_4 — I_6 . On CC2 — CC4, only one instruction is executed, an inefficient use of the multiple EUs. Finding more instructions, such as I_7 that can be rescheduled to run in these clock cycles improves performance. For this reason, rescheduling is a critical feature of superscalar pipelines.

Slide 22 Instruction Rescheduling Strategies

Strategies for instruction rescheduling must avoid the **hazards** associated with data dependency. The goal of rescheduling is to remove RAW hazards, but the rescheduling must not create new WAR or WAW hazards.

If instruction I_{k+j} depends on the result of instruction I_k then I_{k+j} must wait (not be executed) until I_k completes. If this wait causes a pipeline delay then the delay is called a **stall**. If other useful work can be rescheduled into the waiting time, then the CPU is kept busy — this is called **hiding the latency**.

Some data dependencies are caused by the compiler reusing the same register for successive **unrelated** instructions in the high level code. For example, the compiler might use R1 as a load and ALU register in the instructions

x = x + 4

y = y + 5 .

This creates a dependency through R1 even though x and y are independent. This is called a **false dependency** and can be repaired by **register renaming**. The compiler can use R1 for the x instruction and R2 for the y instruction — there is no longer any dependency in the compiled code.

In general, instruction rescheduling can be described as increasing the "distance" between dependent instructions I_k and I_{k+j} — hiding the latency by rescheduling unrelated useful work between the dependent instructions. There are two basic methods of rescheduling:

1. **Compiler optimization** of the object code in the second pass,
2. **Dynamic rescheduling** (out-of-order execution) of the machine code by the CPU at run time.

Slide 23 Efficient Register Reuse

A basic compiler strategy is efficient reuse of registers (without consideration of rescheduling). Registers are allocated to loaded memory operands, and program efficiency often depends on de-allocating (retiring) registers as soon as they are not required, so they can be reused for new instructions.

In the program fragment, R1 is **allocated** in the LW instruction. LW overwrites R1, so that any value assigned by a previous program fragment is lost.

The ADDI instruction reads and writes R1 and an operand. The SW instruction reads R1 as an operand and stores its value. The store instruction is the last use of R1 in this instruction block. It will not be used again as a source in the instruction block. This signifies the **de-allocation** of R1 for the instruction block.

The fourth instruction (ADD) again overwrites R1. The old value of R1 is lost and R1 is now **re-allocated**.

Slide 24 – 26 Register Reuse versus Register Renaming

When rescheduling is performed, the efficient register **reuse** is replaced by register **renaming**. Instead of re-allocating a retired register, a new register is allocated to avoid false dependencies.

In the first pass of the compiler on the program fragment, the last instruction (ADD) re-allocates R1. In a second optimization pass, the register is **renamed**. This means that instead of re-allocating R1 the compiler allocates R10. Then the renamed instruction can be **rescheduled** to follow the first load. Since the DLX requires a stall cycle between LW and ADDI, this rescheduling removes the stall by performing useful work (ADD) while the LW is loading the operand. It is said that this rescheduling hides the latency.

The example on slide 25 shows three lines of high level code and the first pass output of the compiler. R2 is allocated and re-allocated to variables X, Y, Z. This code cannot be rescheduled. In the second pass, the compiler performs register renaming — it allocates R3 to Y and R4 to Z. The resulting code can be rescheduled because the three program blocks (corresponding to the three high level instructions) are independent.

Comparing the program execution on a scalar (single pipeline DLX), the rescheduled code runs in 13 CC instead of 16 CC — a 23% improvement in performance (slide 26).

Slide 27 – 29 Scheduling Algorithm — Instruction Scoreboard

The instruction **scoreboard** method permits detailed control over instruction execution order, taking account of data dependencies and systems policies. Each decoded instruction in the instruction pool is assigned a status field. The DLX requires four status values:

1. **NR** (Not Ready) — A source operand is not available so the instruction is not ready
2. **R** (Ready) — All source operands are available and the instruction can be executed
3. **X** (Executed) — The instruction was executed in the ALU but the result is not available
4. **F** (Finished) — The Instruction was executed and the result is available

Only instructions marked **R** can be executed. Initialization instructions in any program must have status R for any legal program. Execution order also depends on system policy.

The scoreboard table shows the program listing with instructions initially marked NR or R. The table is updated on each clock cycle, as an R instruction is executed. The columns show the CC in which the instruction finishes in the EX execution stage (ALU).

Most instructions are marked F when they complete EX. LW instructions load the operand in stage MEM (after EX) and so are marked X — in the following CC they are marked F.

Slide 28 shows the scoreboard for the renamed (but not rescheduled) code from slide 25. In this example, the policy is that instruction execute in original program order. At program start, each LW is ready. Clock cycles are counted so that the first LW enters the ALU and is marked X in CC1. No other instructions are affected. In CC2, the first LW is marked F, so that the first ADD is marked R (its source operands are now available). In CC3 ADD is executed and marked F (it does not need the extra CC that LW needs), so that SW can be marked R. This pattern continues until the last SW. On CC1, CC5, CC9, no instruction is marked F — this indicates the pipeline stall caused by the Load-ALU hazard.

Slide 29 shows the scoreboard for the same code, but this time the policy is that instructions are executed in the order they become Ready. At program start, each LW is ready. The first LW is marked X in CC1, the second LW is marked X in CC2, and the third LW is marked X in CC3. In CC2, the first LW was marked F, so the first ADD is marked R — this add is executed and marked F in CC4. Similarly, the SUB was marked R in CC3 and is executed and marked F in CC5. This pattern continues until the last SW. An instruction is executed (to X or F) on every clock cycle, so there are no stall cycles. The order of instruction execution is the order of the rescheduled code from slide 25.

Slide 30 – 31 Virtual Register Architecture

The registers visible to the programmer at the level of ISA semantics are called **Architectural Registers** (AR). The DLX was defined with 32 integer ARs, R0 – R31.

Most modern CPUs have additional **Virtual Registers** (VR) not visible to the programmer. The VRs are n real physical registers **VR0, VR1, ... , VRn-1** hidden in the instruction pool. Generally, the VR set is larger than the AR set. The Pentium 4 has 8 ARs compared to 128 VRs.

The CPU assigns a VR to each AR found in an instruction block. This way, register renaming in the compiler is not necessary. If two unrelated blocks both use R1, then one block will be assigned VR1 ← R1 and the second block will be assigned VR2 ← R1. This is a form of **automatic register renaming** in the CPU hardware.

Slide 31 shows an example with 2 instruction blocks. In the first block, the machine instruction allocates VR1 to R1 by loading an operand from memory, and then allocates VR2 to R1 by updating the register. The instructions saved in the instruction pool do not refer to R1 — they are recoded using VR1 and VR2. This instruction block accesses VR1 as destination and source operands until R1 is re-allocated. Stable (unchanging) registers (R2 in the example) do not require virtual registers — the decoded instruction in the instruction pool is written using the literal value read from the AR. When the next block re-allocates R1, the decoder allocates VR3 to R1, and this block uses VR3 as its operand.

VRs can be **retired** (de-allocated) when they are no longer the source for any operation in a block. The de-allocation of VRs requires copying the VR value to the original AR (commitment to state). Commitment must be performed in the correct program order to prevent WAW and WAR hazards. Thus, the first block must write VR2 to R1 before the next block writes VR3 to R1.

Slide 32 – 33 Bookkeeping — Register Allocation Table (RAT)

In order to keep track of the VR allocations, the instruction pool includes a **Register Allocation Table** (RAT). The RAT can be implemented in several ways.

Since there are more VRs than ARs, an intuitive approach is a RAT indexed by the VR number. The entry describes the AR for which the VR is allocated, an instruction block ID, and the VR status (allocated or available).

On Intel processors, the RAT index is the AR name, so that each index can have multiple entries. An entry describes the allocated VR, a sequence number that keeps the allocated VRs in chronological order, and a status field that marks the AR as stable or allocated (currently assigned as a destination to an unfinished instruction in the instruction pool).

When an instruction is decoded in an Intel CPU,

1. The name of each stable source AR is replaced by the value stored in the AR.
2. The name of each allocated source AR is replaced by the VR with the highest sequence number in the RAT for that AR. Thus, in the example on slide 31, the processor must commit VR1 to R1 before committing VR2 to R1 (or it may just skip committing VR1).
3. The name of each destination AR is replaced by a new VR allocation, and a new RAT entry is written. Available VRs are found by searching the RAT for unallocated VRs. The decoder allocates available VRs sequentially mod n, in order to maximize the code distance between VR reuse.
4. The recoded instruction (with literals and VRs) is placed in the instruction pool. Instructions are executed from the pool by reading and writing VRs.

A VR is retired on either of two conditions:

1. An instruction that overwrites the allocated AR is marked finished.
2. The VR does not appear as a source for any unfinished instructions in the pool. In this case, the VR (with the highest sequence number) is copied to the AR and the AR is marked stable in the RAT. The next use of the AR as a source will access the stable AR value.

Slide 34 **Virtual Register Example**

I1 allocates VR1 to R1

I2 allocates VR2 to R2

reads VR1 as the source for R1

I3 reads VR2 as the source for R2

I4 overwrites R2 $\Rightarrow$ retire VR2

allocates VR3 to R2

reads stable register values as the sources for R3 and R4

I5 allocates VR4 to R3

reads VR3 as the source for R2

I6 overwrites R2 $\Rightarrow$ retire VR3

allocates VR5 to R2

reads stable register value for R4

I1 and **I2** have completed and been marked finished. VR1 is no longer a destination register in the instruction pool and can be retired and committed. VR1 is copied to R1. R1 is marked stable.

I7 allocates VR6 to R4

reads stable register value for R1

reads VR5 for R2

Slide 35 – 36 WAR and WAW Pseudo-Dependencies

The code fragment on slide 35 contains dependencies through R1. Since I2 and I3 write to R1, all three instructions are actually independent. This is called **pseudo-dependency**. Like real data dependency, the instructions cannot be rescheduled:

There is a WAR hazard if I1 is placed after I2 or I3.

There is a WAW hazard if I2 is placed after I3.

The pseudo-dependency can be removed using virtual registers:

In I1, allocate VR1 to R2 and read stable register values from R1, R3

In I2, allocate VR2 to R1 read stable values from R3, R4

In I3, allocate VR3 to R1 read stable values from R3, R5

The recoded instructions are

I1 ADD VR1, R1, R3

I2 ADD VR2, R3, R4

I3 ADD VR3, R3, R5

The recoded instructions can be executed in any order. But the virtual registers VR1, VR2, and VR3 must be retired in program order. If I3 is in the instruction pool when VR2 is retired, the value of VR2 will not be committed to R1 (only the last write to a VR associated with R1 is committed).

In a more realistic code fragment on slide 36, the red and green sequential code sections have pseudo-dependencies through R1. The code must run sequentially over 7 clock cycles on a superscalar CPU with multiple ALU and memory EUs. After automatic register renaming with virtual registers, the two code sections can be efficiently rescheduled to run in 4 clock cycles, a 75% improvement in performance.

Slide 37 Dynamic Scheduling

Rescheduling can be performed by the compiler or by the CPU. When the compiler schedules the code, it can only refer to the limited set architectural registers. The CPU can take advantage of the larger set of virtual registers to achieve a better optimization. Rescheduling in the CPU is called **dynamic scheduling**, first introduced in the Tomasulo algorithm for the FPU in IBM mainframes (1967).

Modern dynamic scheduling is based on the superscalar CPU with a fast decoder, automatic register renaming (with VRs), and a scoreboard that marks instructions in the pool with a status field — not ready, ready, finished.

Instructions marked ready in the instruction pool are **issued** (sent to an EU for execution), returned to the pool with the instruction result in the destination VR, and marked finished. Finished instructions are retired in original program order, and their VRs are committed to state.

Slide 38 – 45 Tomasulo Algorithm

The Tomasulo algorithm was the first step in dynamic scheduling. Operation was similar to modern dynamic scheduling, except that there was no RAT and instructions were committed immediately after execution, not in program order.

In the Tomasulo algorithm the instruction pool is called a **reservation station** (RS). Each decoded instruction is buffered in an RS until it is ready for execution. The number of the RS is used as a VR to identify the destination register for that instruction. Source values (literals) from stable registers are encoded into the instruction. Source values from an unfinished instruction are allocated the RS number (as a VR) that points to the instruction that will provide the value.

Ready instructions are moved to an appropriate EU for execution. Immediately after execution, the RS number and result are broadcast on the **Common Data Bus** (CDB). All instructions waiting for that result update their source VRs with the numerical result and store instructions to data memory are completed.

Slides 40 – 43 show a sample program in execution. Slide 40 shows the decoded program, which can run in 3 clock cycles. Slides 41 – 43 show the scoreboard after CC1, CC2, and CC3.

On CC1, three instructions with stable source operands are marked finished. Two instructions that use the destinations of these three are now ready and execute in CC2. On CC3 the VR3 and VR7 are updated. As soon as the value of VR is broadcast on the CDB, the SW instruction can be completed (store operations can go directly from NR to F).

The Common Data Bus provides a simple distributed control mechanism. There is no central scoreboard controller or RAT. This simplicity causes certain problems. The CDB becomes a bottleneck, limiting the performance of other subsystems. Also, instructions are retired by execution order, not program order. This means that branch instructions must be completed before any following instructions are executed (this issue is discussed in presentation 4). Also, there is no well-defined break point for exceptions.

Slide 46 In-Order Retirement

The update of the Tomasulo algorithm is **in-order retirement**, which commits results to architectural state in the order of the original program. The retirement order is set at instruction decode time by the value of the source PC for the instruction. The PC of the last retired instruction is the **progress index** for the program. Instructions beyond the progress index remain in the instruction pool until all instructions with a lower PC are retired.

In-order retirement permits **speculative execution**. This means that it is possible to schedule the execution of instructions that may never run, for example instructions following the target of a conditional branch. If the branch evaluates as taken, then the execution of these instructions turns out to be useful work. But if the branch evaluates as not taken, then these finished speculative instructions must be canceled and deleted from the instruction pool. Speculative execution includes **branch prediction** — the CPU guesses whether a branch is taken or not taken and corrects the instruction pool if the branch evaluates differently (**misprediction**).

In-order retirement also permits **precise exceptions**. If an instruction with PC_n raises an exception that changes the program order, then the finished instructions in the pool with $PC > PC_n$ are canceled and deleted. The exception is handled for PC_n and the subsequent instructions are decoded and executed after the exception is handled.

Instructions are moved from the instruction pool to the **retirement queue** in the order of PC, if they are marked as finished and valid. Valid means, no errors, exceptions or mispredictions.

Slide 47 – 48 Dynamic Scheduling — Tomasulo + In-Order Retirement

Many advanced RISC processors, including IBM Power, ARM, and Sparc CPUs, perform dynamic scheduling. All x86 CPUs since the Pentium Pro (1995) use dynamic scheduling, also called out-of-order execution.

Because the instruction pool permits dynamic scheduling, it is called a **Reorder Buffer** (ROB). An ROB entry contains the instruction PC, the operation, status, VR pointers, and stable operand values. When all VR pointers for an instruction describe destination operands of finished instructions, the instruction is ready.

Instructions are loaded to the ROB from the decoder. When an instruction is marked ready, it is sent to a reservation station (RS) for execution in an EU. The instruction result replaces the destination VR for the instruction in the ROB, the RAT is updated and the instruction in the ROB is marked as finished. Finished instructions are moved to the retirement unit in program order, according to the instruction PC.

Instructions proceed out-of-order between the ROB and RS. Instructions proceed in-order (FIFO) from the decoded to the ROB to the retirement unit.

All data forwarding (bypass) is performed in the ROB. For the pair of instructions

$\mathbf{I_k}$ `ADD R1, R2, R3`
$\mathbf{I_{k+j}}$ `ADD R4, R5, R1`

the decoding of $\mathbf{I_{k+j}}$ depends on the condition of $\mathbf{I_k}$:

If $\mathbf{I_k}$ is retired, then the value of **R1** is loaded from the architectural register

If $\mathbf{I_k}$ is finished but not retired, then the value of **R1** is loaded from the destination register of $\mathbf{I_k}$

If $\mathbf{I_k}$ is not finished, then the value of **R1** is set to a source VR equal to the destination VR for $\mathbf{I_k}$

Slide 49 – 52 Intel x86 Dynamic Scheduling

Since the Pentium Pro, the **x86 ISA** has had certain common features. Modern x86 CPUs can be thought of as a superscalar RISC core with an adaptation layer (in silicon) for handling the IA-32 CISC instruction set.

The x86 Fetch and Decode stages convert IA-32 instructions to 1 – 6 μops per CC. These micro-ops are essentially RISC-type instructions. The CPU performs dynamic scheduling with an ROB that holds micro-ops until ready for execution. A scheduler looks for ready micro-ops and issues micro-ops to parallel execution in multiple EUs — ALU, FPU, Load, Store. Finished micro-ops update execution results to instruction pool. Retirement (in-order write back) is performed by PC order so that finished micro-ops commit to architectural state in original program order.

Slide 50 shows an example of three IA-32 assembly language instructions located in instruction memory (cache). Although IA-32 ARs are named EAX, EBX, and so on, the decoder allocates VRs

R1 — R7. The 3 machine language instructions are decoded to 9 micro-ops and placed into the ROB.

Execution order is according to the order that instructions become ready, but limited by available EUs to 1 load and 1 store per clock cycle. Slide 51 shows the scoreboard for program execution:

Initially all the LW instructions are ready, but they must execute sequentially because there is only one load unit.

On CC1, the first LW is executed and marked F. The ADD is now marked ready.

On CC2, the second LW and the ADD execute in parallel. Now the first SW and first SUB are marked ready.

On CC3, the first SW, first SUB, and last LW execute in parallel. The second SW and second SUB are marked ready.

On CC4, the second SW and second SUB execute in parallel. The third SW is marked ready.

On CC5, the third SW is finished.

Slide 52 shows the organization of the **Nahalem Micro-Architecture**, the version of the x86 pipeline found in one core of an i7 CPU.

At the top in purple is L1 instruction cache and the IF unit. Instructions are fed to the decoders shown in orange. The decoded micro-ops are sent to the 28 Entry LSD buffer. The LSD is a loop stream detector, an example of a technique called **trace cache** (discussed in presentation 5). The LSD detects loops in programs and stores the sequence of micro-ops in each loop — this eliminates the need to decode the loop separately on each iteration, improving loop performance.

Micro-ops are then moved to the RAT for VR allocation and then to the ROB and reservation stations in yellow. The EUs are shown in blue (integer, FP, and SSE vector units) and green (memory access units and data cache). The memory ordering buffer (MOB) is another optimization. Many instructions depend on data loads to proceed, but data stores do not affect performance. The MOB reorders data loads before stores so the program can continue quickly. If a load refers to a location that has not yet been updated by a waiting store, then the MOB provides the temporary value for that location from the queue of store instructions.

The retirement unit is seen in gray to the right of the ROB.

Slide 53 – 54 Deep Superpipeline

The organization of the Nahalem Micro-Architecture on slide 52 is an implementation of the idealized pipeline on slide 49. The number of pipeline stages in this complex implementation is a highly technical question depending on detailed analysis of the number of steps in the operation of each subsystem. The Pentium III was reported to be organized into a 10 stage and the Pentium 4 reported to have a 20 stage pipeline. This is an example of superpipelining (first discussed in slides 42 – 43 of presentation 1).

The general idea of **deep superpipelining** is to divide each pipeline stage in half, so that each new stage performs half the work in half the time.

For the 5 stage DLX pipeline with stages IF, ID, EX, MEM, WB, superpipelining would produce a 10 stage pipeline, with stages IF1, IF2, ID1, ID2, EX1, EX2, MEM1, MEM2, WB1, WB2. Although the pair IF1 , IF2 performs the same work performed in the original IF, the diagram shows that

instruction I_{k+1} is (partially) fetched when instruction I_k moves from IF1 to IF2. This means that

$$\tau_{\text{superpipeline}} = \frac{\tau_{\text{pipeline}}}{2} \quad \Rightarrow \quad \text{Run time}_{\text{superpipeline}} = \frac{1}{2}\text{Run time}_{\text{pipeline}}$$

The problem with deep superpipelining is that not all instructions can be effectively split in half. Some operations require a fixed time and do not scale with clock rate. For example, exceptions such as page faults, branch penalties, and cache updates depend on latency in external devices. If the clock runs faster then more stall cycles will be counted while waiting for the device, and so CPI^{stall} will be larger.

In the transition from the 10 stage pipeline of Pentium III to the 20 stage pipeline of Pentium 4, this caused non-scaling latency a problem for early models. Based on the CPU equation, a 1.5 GHz processor should be 50% faster than a similar processor at 1.0 GHz. That is, if the CPI and IC are about the same for two CPUs then

$$S = T / T' \approx \tau / \tau' = R' / R = 1.5 \text{ GHz} / 1.0 \text{ GHz} = 1.5$$

But comparisons of SPEC Cint on two models found that the 1.5 GHz Pentium 4 (P4) is only 20% faster than 1.0 GHz Pentium III (PIII). Assuming that compilation for the two processors gives a similar IC, one finds that $CPI^{P4} = 1.25\ CPI^{PIII}$, so that it takes 25% more CC to run a program on the P4 than on the PIII. Assuming that $CPI^{ideal} = 1$ on both CPUs and $CPI^{stall-PIII} \approx 0.2$, one finds

$$CPI^{stall-P4} = 1.25 \times CPI^{stall-PIII} + 0.25 \approx 0.5 \quad \Rightarrow \quad CPI^{P4} \approx 1.5$$

Intel publications at the time claimed that the "meaning" of clock speed was different for the PIII and P4, and so the 20% improvement was pure improvement. This claim was equivalent to assigning the factor of 1.5 to the meaning of clock speed, instead of discussing CPI and causes of stalls in deep superpipelining. The matter was soon forgotten because the P4 clock speed was quickly raised to about 4 GHz, a speed the PIII structure could never attain.

Slide 55 – 56 Hyper-Threading

To solve the problem of increased pipeline stalls caused by superpipelining, Intel added **Hyper-Threading**® to the Pentium 4. A Hyper-Threaded processor contains a standard single core pipeline with two separate copies of the architectural registers (architectural state). Since the CPU has two separate program counters, it can fetch two separate program threads at the same time. To the OS, the processor looks like two processors: CPU0 and CPU1.

Since the processor has one execution core (EUs and data memory) it shares the threads by interleaving — it issues instructions (fetches and decodes instructions to the ROB) from alternate threads on alternate clock cycles. Micro-ops in the ROB are labeled by thread and (since the threads are independent) provide increased opportunities for dynamic scheduling.

With Hyper-Threading, if one thread stalls the other thread can continue running until the stall is resolved. CPU 0 and CPU1 keep busy most of the time.

The improvement from Hyper-Threading can be estimated by writing

$$CPI^{stall} = CPS \times P_s$$

where CPS = cycles per stall, and P_s is the probability of a stall per instruction.

With Hyper-Threading, the CPU only suffers a stall if both threads stall at the same time. Since the threads are independent, $P(2 \text{ simultaneous stalls}) \approx (P_s)^2$ and $CPI^{stall\text{-}HT} \approx CPS \times (P_s)^2$. Assuming that each stall is 2 clock cycles, so $P_s = 0.5 / 2 = 0.25$, the speedup is found from

$$S = \frac{CPI \times IC \times \tau}{CPI^{HT} \times IC^{HT} \times \tau^{HT}} = \frac{CPI}{CPI^{HT}} = \frac{1 + CPS \times P_S}{1 + CPS \times \left[P_S\right]^2} = \frac{1 + 2 \times 0.25}{1 + 2 \times \left[0.25\right]^2} = 1.2$$

This 20% speedup is precisely the value published by Intel (slide 27 in presentation 7).

Slide 57 Hardware Utilization in x86 Example

The program on slide 51 executed in 5 clock cycles, instead of 9 clock cycles (if each instruction executed sequentially). The speedup of 9/5 = 1.8 is significant. But most of the execution units were idle most of the time. Higher performance requires more instructions scheduled to run in parallel. Increasing instruction level parallelism (ILP) requires attention to general program structure.

Slide 58 – 60 Overhead in Performance Calculations

A typical program section consists of:

1. **Head** — set up instruction that run one time
2. **Main functions** — program loops that run over many iterations
3. **Tail** — shut down instructions that run one time

The head and tail contribute a small **overhead** to the main functions that require most run time.

Similarly, the execution of a program in a superscalar consists of overlapping activities:

1. The Fetch + Decode units run in the **background** to keep the ROB filled. Because they are pipelined, after the first few CC required to fill the ROB, these units contribute a small **overhead** to run time.
2. The **program executes** on a set of u EUs (EU_1, EU_2, ... , EU_u). The larger IC, the longer the run time in the EUs. The more EUs being utilized, the shorter the run time.
3. The **retirement unit** removes instructions from the ROB. Because it is pipelined, after the first few CC, this unit contributes a small **overhead** to run time.

For example, a program might require 8 CC to fill the ROB, 10^6 CC to execute a loop (while IF+ID continuing filling the ROB and the retire unit removes completed instructions in the background), and another 8 CC to finish emptying the ROB after the main loop completes. The total number of CC required to run a program can be split into

$$N = N_{IF+ID} + N_{program} + N_{retirement} = N_{IF+ID} + N_{head} + N_{main\ loops} + N_{tail} + N_{retirement} = N_{overhead} + N_{main\ loops}$$

$$\text{where } N_{overhead} = N_{IF+ID} + N_{head} + N_{tail} + N_{retirement}$$

For a simple program with one loop, with $N_{one\text{-}loop}$ clock cycles per loop iteration and $n_{iterations}$ iterations,

$$N_{main\ loop} = n_{iterations} \times N_{one\text{-}loop}$$

In this approximation, the CPI can be estimated as

$$CPI = \frac{N_{overhead} + n_{iterations} \times N_{one\text{-}loop}}{IC_{overhead} + n_{iterations} \times IC_{one\text{-}loop}} = \frac{\frac{N_{overhead}}{n_{iterations}} + N_{one\text{-}loop}}{\frac{IC_{overhead}}{n_{iterations}} + IC_{one\text{-}loop}} \xrightarrow[n_{iterations} \to large]{} \frac{N_{one\text{-}loop}}{IC_{one\text{-}loop}} = CPI_{one\text{-}loop}$$

In the ideal case, dividing the work among u EUs divides the number of clock cycles by u, and so reduces CPI to

$$N_{one\text{-}loop} = \frac{N_{one\text{-}loop}^{u=1}}{u} \Rightarrow CPI = \frac{1}{u} \times \frac{N_{one\text{-}loop}^{u=1}}{IC_{one\text{-}loop}} = \frac{CPI_{ideal}^{u=1}}{u} = \frac{1}{u}$$

The example on slide 60 shows a program with data dependencies in R1 and R2, so that only SW and BNEZ can be run in parallel. The ADDI instruction only runs once and is overhead. In each loop iteration, 5 instructions run during 4 clock cycles.

The loop runs over 0x1000 / 4 = 0x400 = 1024 iterations. Including the IF+ID and retirement overhead, and ADDI

$$CPI = \frac{overhead + 4 \times 1024}{1 + 5 \times 1024} = \frac{\frac{overhead}{1024} + 4}{\frac{1}{1024} + 5} \to 0.80$$

Clearly, the average CPI for the program can be approximated by the CPI for one loop.

Slide 61 – 69 Loop Unrolling

One method to increase instruction level parallelism (ILP) is called **loop unrolling** — combining two or more loop iterations into one. With multiple EUs and automatic register renaming, it is often possible to improve performance by running multiple loop iterations at the same time.

The program on slide 60 (and on the left side of slide 61) must run sequentially because of the data dependencies through R1 and R2. On slide 60, the CPI was found to be 0.80 on the superscalar. With register renaming (performed automatically in the CPU or by the compiler as an optimization) the loop can be unrolled to perform 4 loop iterations at one time.

The scoreboard for the unrolled program on the 2-ALU superscalar is shown on slide 62. In the unrolled program, 14 instructions complete in 7 CC, so CPI = 0.50 . This represents an improvement of S = 0.8 / 0.5 = 1.6 over the initial version. The scoreboard for the same program on a CPU with 4 ALUs + 4 Read Units + 4 Write Units is shown on slide 63. The CPI is now 0.29 with an improvement of S = 0.8 / 0.29 = 2.8 over the original version.

Slide 64 shows a code fragment written in x86 assembly language. This program loads data from memory, adds a table entry to the operand and stores the result. Slide 65 shows the VR allocation and recoded instructions for the code fragment. The loop unrolling for this program is accomplished in the CPU after decoding to micro-ops. The table index ESI that runs from 1 to 400 in the original version is split into VR30 that runs from 1 to 200, and VR31 that runs from 201 to 400. The initial decoding is shown on the left side of slide 67, and the unrolled decoding is shown on the right side. On the right side, 4 instructions are doubled. The scoreboard on slide 68 shows that one loop iteration for the original version takes 4 clock cycles. The program runs sequentially, with low utilization of the EUs in the superscalar. The scoreboard on slide 69 shows that one loop iteration for the unrolled version also takes 4 clock cycles. The program runs with higher utilization of the EUs in the superscalar, executing 4 instructions per clock cycle. The unrolled version runs twice as fast because the number of loop iterations is divided in half.

Slide 70 – 76 Software Pipelining with Explicit Parallelism

Some modern CPUs, such as the Intel 64-bit Itanium (not an x86 processor), permit the compiler to indicate parallel code blocks explicitly. In assembly language, the end of a parallel block is indicated by the symbol **;;** which tells the CPU that instructions up to **;;** are marked ready and instructions after **;;** are marked not ready. This facility permits a method of software unrolling called **software pipelining**.

Conventional code appears at the top of slide 70. Every instruction executes sequentially on the superscalar because of data dependencies. Below appears a software pipelined version. The header prepares all sources required for the first loop iteration. In loop iteration N, the loop code reads only sources prepared in iteration N – 1 and prepares all of the destinations that will be read as sources in iteration N + 1.

The left side of slide 71 shows a sequential code fragment and the pipelined version appears to the right. Most instructions in the header run sequentially. All instructions in the loop run in parallel. After the final loop iteration, the tail completes the storage of the last results.

Slides 72 and 73 show the register values while running the header and first two loop iterations. Each source is the last value set for that operand. Slide 74 shows the complete run if the original loop had 3 iterations.

The unrolled software pipelined code appears on slide 75. The scoreboard is shown on slide 76, and as expected the entire loop of 14 instructions runs in 1 clock cycle, so CPI = 0.07, an improvement of 0.80 / 0.07 = 11 over the original version and an improvement of 0.50 / 0.07 = 7 over standard loop unrolling.

Slide 76 Improving Hardware Utilization — Additional Directions

Loop unrolling can improve hardware utilization under appropriate conditions (in particular, independence between iterations).

When a single program thread is highly sequential and with tight data dependencies (for example, bubble sort), finding independent instructions to keep the EUs busy can be difficult.

Instead of parallelizing a single loop-oriented procedure, another source of independent instructions is **speculation**. In speculative execution, **branch prediction** is used to find branch operations that are likely to execute (for example a conditional function call in a high level program). With dynamic scheduling, it is possible to execute instructions that follow the branch and save them in the ROB until the branch is evaluated completely. If the prediction is correct, then much work has already been finished. On a misprediction, the instructions in the ROB are cancelled.

Presentation 4

Speculative Execution

ביצוע ספקולטיבי מתייחס לשיטות לקביעת תחזית (ניחוש מושכל) על ההתנהגות העתידית של תהליך עיבוד כדי למנוע או להסתיר latency. השימוש בשיטות לזיהוי תבניות מאפשר למעבד עכשווי לדייק בחיזוי התנהגות פקודות סיעוף (branch) ועדכון מטמון (cache) — שני מקורות עיקריים לשיהוי בריצת תוכניות. בנוסף, היכולת לחזות התנהגות מבני בקרה מורכבים מאפשרת ביצוע פקודות באופן ספקולטיבי גם להשיג סיום מוקדם של משימות וגם להעלות את ניצולת המשאבים במעבד.

Slide 2 **Speculative Execution**

Instruction stream speculation refers to methods of program execution that:

1. Predict program behavior before the actual execution flow is precisely determined,
2. Execute the predicted program flow,
3. Verify the execution flow and program results before committing results to state,
4. Recover from misprediction.

The most common method of speculation is **branch prediction**, in which the results of a branch instruction, the branch **action** (taken/not-taken) and the **target** address, are predicted by hardware. The prediction is based on pattern recognition techniques applied to branch statistics dynamically stored in the CPU during program execution. Modern processors can achieve a 90% correct prediction rate or better, depending on the type of code.

A related method is instruction and data **cache prefetch**. Normally, cache updates are performed on a cache miss — this involves a cache penalty (delay) while the update is processed. Prefetch permits identification of instruction and data flows and enables speculative cache updates and prevent the cache penalty.

A more advanced method is **trace cache**, which caches frequent instruction flow sequences that can span multiple control blocks. When running complex loop oriented program sequences, trace cache can improve performance of repeated branch sequences.

Branch predication is a replacement for branch instructions in many situations, especially if-then-else structures. Instead using conditional branch instructions to jump over instructions, instructions can be predicated, that is, assigned a predicate flag set by various program conditions. The predicate flag enables conditional execution — the instruction is either executed or cancelled. It is possible to execute multiple branch flow alternatives and commit to architectural state only the correctly verified instruction flow.

Slide 3 **Branch Behavior**

The typical **if-then-else** structure is illustrated in the following fragment:

PC (address)	Instruction
PC_{BEQZ}	`BEQZ REG, offset`
PC_{BEQZ} + 4	`fall-through`
	…
PC_{BEQZ} + 4 + offset	`target`
	`continue`

The conditional branch BEQZ implements:
if (REG == 0)
then PC + 4 ← PC + 4 + offset ⇒ program control continues at the target instruction
else the program continues from the fall-through instruction at PC + 4

In dynamic scheduling, instructions are executed out-of-order, but results are committed to state in program order as indexed by the PC of the original program listing (slide 46 in presentation 3). In principle, out-of-order execution could create a **barrier at branch instructions**, because it is not known whether to execute the target or fall-through until the branch is correctly evaluated. A barrier would mean that instructions following a branch could never be executed out-of-order (before the branch itself), even when they are marked ready.

Speculation removes the barrier in the following way:

When PC < PC_{branch} (meaning that the branch has not yet been evaluated and committed to state), the CPU predicts the branch action (taken or not-taken) and the target address (used if taken). Instructions are decoded to the ROB, executed in the EUs and results are returned to the ROB according to the prediction. These speculative instructions are kept in the ROB and marked as **speculative** — they cannot be retired (committed to state) until the branch is **verified**.

The verification of the branch occurs when PC = PC_{branch} and the **branch result is committed**. After verification, the speculative instructions in the ROB are marked as **valid** (for a correct prediction) or cancelled. Valid instructions are then retired in program order.

Slide 4 Basic Branch Prediction

Branch prediction requires two new hardware elements in the fetch+decode stage of the pipeline:

The first element is a **Branch History Register** (BHR). The index is the source identifier (PC) of a particular branch instruction in the program. The entry consists of the branch action history and a branch prediction made by some algorithm. The branch prediction includes the target address and the branch action (taken or not-taken).

The second element is **Branch Predictor** logic that enters the PC of each branch instruction into the BHR and updates the instruction fetch stage with the PC for fetches past the PC of the branch. Generally, the PC of the branch is ahead of the PC used by the retirement unit, and behind the PC used by the instruction fetch unit. For example, at any moment it may be that

1. The retirement unit has committed instructions up to PC = 100,
2. There are finished instruction in the ROB instructions up to PC = 150 and unfinished instructions in the ROB up to PC = 250,
3. A branch instruction was found in the fetch stream at PC = 200,
4. The fetch unit is currently fetching instructions from PC = target = 750. These instructions are marked speculative.

The branch is ready for execution when all of its sources (the condition to be checked) are ready. When the branch is evaluated, the result is compared with the prediction and the branch history is updated in the BHR. If the prediction is correct, program flow continues. Otherwise, there was a **misprediction** — the speculative instructions are now marked invalid. The invalid locations in the ROB are now available for filling by new instructions fetched from the address found in evaluating the branch instruction.

Slide 5 – 7 Branch Prediction for DLX Pipeline

Slide 5 shows a simple branch prediction system that could be added to the 5 stage DLX pipeline. The branch prediction logic and BHR are located in the IF stage. A branch instruction is fetched and identified on CC1, and a prediction is made for the next instruction — target or fall-through. On CC2, IF fetches from the predicted address and ID evaluates the branch. The BHR is updated in the IF stage. If the prediction is correct, the IF stage continues fetching instructions from the predicted location. On a misprediction, the predicted instruction is flushed (marked a NOP, introducing a pipeline bubble) and the correct instruction is fetched on CC3.

Slide 6 shows a simple example for the DLX with simple branch prediction. The main loop consists of a block of ALU instructions followed by a branch. On the first loop iteration, the branch is loaded for the first time and has no history in the BHR. In this case, IF must predict the fall-through instruction (it has no other options, since the target address is still unknown). When the branch evaluates in stage ID, the BHR is updated, the fall-through is flushed and the target is loaded. The pipeline flush causes a stall of 1 CC. At the end of the second iteration, the branch is predicted as taken and the target address in the BHR is used. This prediction behavior continues until the last iteration. On the last iteration, the branch is predicted as taken, but that is now a misprediction. The BHR is updated, the target instruction is flushed and the fall-through instruction is loaded after a 1 CC stall. After N iterations the number of instructions is $N \times B + 2$ (the 2 are the initial ADDI and the fall-through that runs once) so $CPI^{stall} = 2 / (N \times B + 2) \rightarrow 0$ for large N. On programs of this simple type, branch prediction is very effective, removing the 1 CC branch stall on almost every loop iteration.

In the typical case, branch stalls cause significant performance degradation. Most program run time is devoted to control loops, with branch instructions about 20% to 25% of all instructions run. Since about 2/3 of all branches are taken, in the worst case taken branch instructions make up $1/4 \times 2/3 = 1/6$ of all instructions. If the branch penalty (the number of instructions to be flushed after a misprediction), then CPI^{stall} must be at least $N / 6$. Even with $N = 1$ on the DLX, this large value of CPI^{stall} limits performance.

Slide 8 Dynamic Branch Prediction

General **Dynamic Branch Prediction** uses pattern recognition techniques to predict future branch behavior based on past behavior. The simplest techniques involve one-level prediction, in which the recent branch history leads directly to a prediction based on the relative number of taken and not-taken results. Two-level prediction examines recent history to find patterns in the branch behavior that followed prior events matching some particular branch history. The patterns can be global (predicting from the recent execution pattern of all recent branches), local (predicting from the recent execution pattern of each individual branch), or some combination.

Branch prediction is limited by the complex hardware required to store and analyze the long-term behavior of many branch instructions. The BHR can become very large making both the time to learn the behavior and the time to make a prediction long.

Slide 9 One-Level Prediction

In **one-level prediction**, the Branch History Register (BHR) is a table whose index is a hash function of the branch source PC (the address of the branch instruction in memory). The entry consists of an N-bit profile — the history of the N most recent branch outcomes. The results are shifted in from the right (shifting out the oldest entry from the left).

To predict the next branch, the **N-bit BHR** entry is counted by an **n-bit saturating up/down counter** and the branch target (the target PC of last taken branch).

A saturating up/down counter is an n-bit register initialized to 0. On each taken branch the counter is incremented by 1, but saturates at 11...111 (the value $2^n - 1$). On each not-taken branch the counter is decremented by 1, but saturates at 00...000.

The branch prediction is taken from the value of the high order bit. The branch will only be predicted as taken when the counter is at least 10...000 (the value 2^{n-1}), following about 2^{n-1} taken branches.

If a branch profile is stored in a 6-bit BHR as 110110, then the oldest outcome is 1 and the newest is 0. The counter transitions in a 2-bit counter are:

00—>01—>10—>01—>10—>11—>10

Slide 10 1-Bit Saturating Up/Down Counter

A 1-bit counter is either 0 or 1. Its prediction is always equal to the previous branch action. It can predict a single static loop with an accuracy of about 1 / (loop iterations). On an alternating taken / not-taken branch it will mispredict the branch action 100% of the time.

Slide 11 – 12 2-Bit Saturating Up/Down Counter

The 2-bit counter counts from 0 to 3. It can predict a pair of nested loops with some accuracy. The prediction states are called

00 — strongly not-taken (a strong prediction of not-taken)

01 — weakly not-taken (a weak prediction of not-taken)

10 — weakly taken (a weak prediction of taken)

11 — strongly taken (a strong predication of taken)

Alternative 2-bit counters are shown on slide 12.

The first counter predicts strongly taken after 2 consecutive taken branches and predicts strongly not-taken after 2 consecutive not-taken branches.

The second counter also predicts strongly taken after 2 consecutive taken branches, but counts down more gradually. This counter favors a prediction of taken, consistent with realistic branch statistics.

Most research suggests that neither alternative counter significantly improves prediction performance.

Slide 13 – 14 Short Nested Loop

Slide 13 shows the outputs of 1, 2, 3, and 4 bit counters for 12 iterations of a short nested loop, based on global branch behavior in One-Level Prediction.

The inner loop (in blue) runs Len times in index R2 per outer loop. The outer loop (in green) runs Len times in index R1. The general pattern of taken and not-taken branches is

$$\underbrace{11\ldots101}_{1}\underbrace{11\ldots101}_{2}\underbrace{11\ldots101}_{3}\ldots\underbrace{11\ldots101}_{\text{Len-1}}\underbrace{11\ldots100}_{\text{Len}}$$

where 1 represents a taken branch and 0 represents a not-taken branch.

For Len = 3,

$$1234\,5678\,9ABC$$
$$\underbrace{1101}_{1}\,\underbrace{1101}_{2}\,\underbrace{1100}_{3}$$

the BHR entries as the program runs are shown in the table.

The predictions made for these BHR entries depend on the size of the counter.

A 2-bit counter operating on the 4-bit BHR entries will produce predictions as follows (the Next is the outcome if the program runs again):

	Init	Count 1	Count 2	Count 3	Count 4	Prediction	Next
0000	00	00	00	00	00	0	—
0001	00	01	00	00	00	0	—
0011	00	01	10	01	00	0	—
0110	00	00	01	10	01	0	—
1101	00	01	00	01	10	1	1
1011	00	01	10	01	10	1	1
0111	00	01	10	11	10	1	0
1110	00	00	01	10	11	1	1
1101	00	01	00	01	10	1	1
1011	00	01	10	01	10	1	1
0111	00	01	10	11	10	1	0
1110	00	00	01	10	11	1	1
1100	00	00	00	01	10	1	—

Slide 14 compares the branch predictions for the 1 and 2 bit counters with the actual branch actions. The two left columns show the value of registers R1 and R2 when the program executes the instruction at label L2.

The 1-bit counter initializes to 0 and mispredicts the first branch (mispredictions are shown in a red square). The first branch raises the counter to 1, which correctly predicts the next L – 1 taken branches. The last inner loop branch is mispredicted and the not-taken lowers the counter to 0 which mispredicts the taken branch in the outer loop. The total number of mispredictions for the 1-bit counter is $1 + (L - 1) \times 2 + 1 = 2L$.

The 2-bit counter initializes to 0 and mispredicts the first two branches (this represents the longer learning curve for the wider counter). It correctly predicts the next L – 1 taken branches, but mispredicts the last iteration of the inner loop. The two bit counter weakly (but correctly)

predicts taken for the branch in the outer loop. The total number of mispredictions for the 2-bit counter is $2 + (L - 1) \times 1 + 2 = L + 3$, which is half the value of the 2-bit counter.

Most 1-step branch predictors use a 2-bit counter.

Slide 15 – 17 Yeh-Patt Two-Level Adaptive Prediction

The **Yeh-Patt** method for two-level branch prediction is implemented in Intel x86 CPUs. As in one-level prediction, the index to the Branch History Register (BHR) is a hash of the source PC of a branch instruction. The BHR stores the **profile** (the actual sequence of recent branch actions), along with the target address of the last taken branch. The stored profile is used as the index to a Pattern History Table (PHT), whose entry is called a **pattern** — the history of next branch actions that followed a specific profile of branch actions. The PHT entry characterizes the pattern of branch behavior when it has repeated a given branch history. The prediction logic feeds the pattern sequence into an n-bit saturating counter — the prediction is the high order bit. This method is a form of pattern recognition.

The profile stored in the BHR is a k-bit bitmap shifted in from the right. If a certain branch has been observed (executed and verified) b times, then the bitmap consists of k bits (1 for taken and 0 for not-taken) starting on the right with result b, then (moving left) result b – 1, b – 2, and so on down to bit b – k – 1 on the left. The profile records the history of k branch results — the 6-bit bitmap 011101 records a branch that was not-taken, then taken 3 times, then not-taken, and then taken.

For each profile in the BHR, the PHT records the next observed action. For example, if the profile 011101 points to the 5-bit PHT entry 00101, this says that on the last 5 times that the branch had behaved as **011101**, its subsequent actions were **0**, and then **0**, and then **1**, and then **0**, and then **1**. This records 5 branch actions from this actual history (distributed among many other results):

0111010, **0111010**, **0111011**, **0111010**, **0111011**

The prediction for the next result is made by entering the pattern into an n-bit counter. The output of a 2-bit counter (initialized to 00) from the pattern **00101** is

00	→	00	→	00	→	01	→	00	→	01
init	**0**		**0**		**1**		**0**		**1**	

The prediction based on this pattern is not-taken (because in 3 of 5 cases when the profile was 011101 the next branch was not taken).

Slide 18 Yeh-Patt — Simple Example

The short nested loop with N = 3 produces the global sequence of branches **110111011100**. A 2-bit global BHR is initialized to zero and new branch actions are shifted in from the right, as shown in the table columns labeled **Global Profile m**. The columns labeled **Observed Profiles** show the 2-bit combinations that have appeared in the BHR up to each branch event. The columns labeled **Actual Next Action** lists the observed result that follows each of the observed profiles.

For example, the BHR is initialized to 00. One the first execution of a branch, the action is taken and so a 1 is shifted into the BHR from the right — the new BHR entry is 01. At this point, 2 separate profiles have been observed — 00 and 01. The branch action following the appearance of profile 00 is 1, and the branch action following the appearance of profile 01 is also 1. This 1 is

shifted into the BHR from the right, producing entry 11 (also a new profile appearing for the first time). The action of the next branch after the appearance of 11 in BHR is 0.

The prediction is a 1-bit counter (last action). On initialization, no next action for the profile 00 has been recorded in the PHT and so there is no prediction (predict not-taken). The next action following 00 is 1, and on the next branch following the pattern 00 (if there is one), the prediction will be 1. The next action after the appearance of the profile 01 is 1 — on the next branch following the pattern 01 the prediction will be 1.

No actions have been observed for the profiles 00, 01, 11, 10 in the first 3 branches. No predictions are possible for these branches — they are predicted not-taken. Only one prediction is correct (at random), because the third branch result is actually not-taken.

For the next branches in the program, observed next actions are available for all 4 possible profiles. On the 4th branch, the profile in the BHR is 01. The entry in the BHT for index 01 is 1, and so this branch is correctly predicted as taken. The next 2 branches cannot be correctly predicted with this configuration because the profile 11 is followed by 0 half the time and followed by 1 half the time. The last branch is mispredicted because the program ends.

Two basic observations are (1) short loops are difficult to predict, and (2) 1-bit counters do not predict accurately.

Slide 19 Yeh-Patt — Building BHR and PHT

Slide 19 shows an example of how the BHR and PHT are constructed dynamically.

Some branch instruction executes an alternating sequence of 12 taken / not-taken actions, 101010101010. The 5-bit BHR is initialized to 00000. It is convenient to present the sequence of branches in a table, in order to identify the **profile** in the BHR Entry and the actual branch (**next**) that will be executed following the observed profile.

						Branch Number													
						1	2	3	4	5	6	7	8	9	10	11	12		
	Init					1	0	1	0	1	0	1	0	1	0	1	0	BHR Entry	**Next**
0	0	0	0	0	0						**Learning Phase** Some of the 5 bits in the BHR are the initialized state							00000	1
1		0	0	0	0	1												00001	0
2			0	0	0	1	0											00010	1
3				0	0	1	0	1										00101	0
4					0	1	0	1	0									01010	1
5						1	0	1	0	1								10101	0
6							0	1	0	1	0							01010	1
7								1	0	1	0	1						10101	0
8									0	1	0	1	0					01010	1
9										1	0	1	0	1				10101	0
10											0	1	0	1	0			01010	1
11												1	0	1	0	1		10101	0
12													0	1	0	1	0	01010	—

Each row shows the 5-bit entry in the BHR found from shifting in (from the right) the most recent branch **action** (in red). After branch 1, the action 1 is shifted in from the right, producing the profile 00001. The actual action of the **next** branch (branch 2) is 0. There are 12 branches in the profile, so there is no action after the 12th branch.

The PHT is initialized to 000 for all profiles. The PHT entry for profile 01010 evolves as:

After Observed Branch	Pattern			
1	0	0	0	01010 not yet observed
2	0	0	0	
3	0	0	0	
4	0	0	0	
5	0	0	1	01010 followed by 1
6	0	0	1	
7	0	1	1	01010 followed by 1
8	0	1	1	
9	1	1	1	01010 followed by 1
10	1	1	1	
11	1	1	1	01010 followed by 1
12	1	1	1	

The observed profiles and observed next actions after all 12 branches are:

Profile				
00000	0	0	1	These profile are observed only 1 time
00001	0	0	0	
00010	0	0	1	
00101	0	0	0	
01010	1	1	1	This profile is observed 5 times after branches 4, 6, 8, 10, 12
10101	0	0	0	This profile is observed 4 times after branches 5, 7, 9, 11

Since the pattern for 10101 remains unchanged (000), the entries in the PHT are stable after branch 8.

Suppose that the program continues so that the BHR and PHT are used to predict the actions of a second run. Using a 2-bit counter to predict the next branch, the prediction for the profile 10101 will always be correct — the PHT entry is 000 and the next branch not-taken is correctly predicted. For the profile 01010, the PHT entry used to predict branch 9 is 011, so the branch is correctly predicted as taken. Similarly, the PHT entry used to predict branch 11 is 111, and the branch is correctly predicted as taken.

Slide 20 – 22 Example — Alternating Branches

Slide 20 shows the performance of one step prediction with 1-bit and 2-bit counters (per branch) on a sequence of alternating branches. The BNEZ in the inner loop is taken on alternate iterations, generating a sequence of alternating taken / not-taken branches. The BEQZ in the outer loop is only reached when BNEZ falls through and so it is always taken.

The 1-bit counter always predicts the previous result, so that the 1-bit counter for the BNEZ inner loop is always mispredicted. The 1-bit counter for the BEQZ outer loop is initialized to 0 and updates to 1 after the first branch, providing a correct prediction on branch 4 and onward.

The 2-bit counter value for the BNEZ inner loop alternated between 00 and 01. It always predicts not-taken and is correct half the time (equivalent to the outcome of no branch prediction at all). The 2-bit counter value for the BEQZ outer loop is initialized to 00 and updates to 01, 10, 11. It provides a correct prediction on branch 6.

Slide 21 shows the evolution of a 4-bit BHR and 2-bit PHT in two-step prediction for the same sequence of BNEZ results. The grayed area shows the learning stage for the branch predictor.

The required branch profiles 0101 and 1010 are observed on the 3rd and 4th branches — the 2-bit PHT registers hold valid observations after 7 branches. Branch 8 and onward will be correctly predicted for the BNEZ.

Slide 22 shows the program and BHR / PHT evolution. Initialized zeros are shown in gray. Observed next branches (the BNEZ result in that row) are shown in blue — these values are used to predict the next occurrence of the profile. The right column (Y-P) shows the prediction for the branch in that row.

Slide 23 Example — Another Example

Slide 23 shows a more irregular example. The BHR shows 17 profiles (after the learning phase) and the PHT shows patterns for 7 observed profiles (after the learning phase). Except for the profile 10101, all profiles will produce correct predictions.

The problem with 10101 is that half the time the next is 0 and half the time the next is 1. So half the time the PHT entry will be 101 and half the time 010.

For 010 the counter produces 00—>00—>01—>00 which gives prediction 0

For 101 the counter produces 00—>01—>00—>01 which gives prediction 0

The counter always predicts 0 but half the time the next is 1. This pattern is called unpredictable.

Slide 24 – 29 DEC Alpha Branch Predictor

Slides 24 to 29 show the branch predictor in the Alpha RISC CPU designed by the Digital Equipment Corporation (DEC) as a RISC evolution from the VAX. It was manufactured until 2007. In its time, the Alpha was the fastest CPU available. In the Alpha, a local predictor records the last 10 actions of each branch instruction (indexed by its PC). This branch profile is the index of a local predictor (a 3-bit saturating counter). The highest order bit of the counter forms the prediction. The global path history records the last 12 actions of all branches. This global profile is used as the index for two separate units: a global predictor (a 2-bit saturating counter), and a choice predictor whose entry is a 2-bit counter of correct predictions made by the global predictor. When the counter in the choice predictor is 00 or 01, the result of the local branch predictor is used for the prediction. When the counter in the choice predictor is 10 or 11, the result of the global branch predictor is used for the prediction.

Slide 28 compares the Yeh-Patt two-step prediction with the Alpha prediction. Aside from the Alpha's use of local and global prediction (Y-P is always local or global), the main difference is the way branches are counted. In the Yeh-Patt method, the most recent s-bit pattern is sent to the counter — any information from before s instances of a profile is lost (this is called an **event horizon**). In the Alpha, the counter is continuously updated with the most recent result, so that the counter output is based on the results of all previous instances of a given profile.

The first table on slide 29 shows (in red) the next actions following the profile 11 in the branch sequence 11 11110 11110 11110 11110... The initial 1 bits sets a bias in the data.

The second table shows this sequence of next results (first row) and the two methods of prediction. The second row shows the output of a 2-bit counter operating on this sequence of next results. Because of the bias in the data, the direct counting (Alpha method) predicts taken

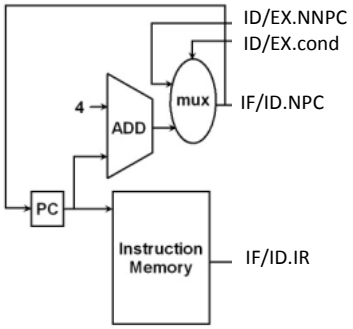
for all branches after the 2nd. Since most branches are actually taken, only the not-taken branches are mispredicted. The third row shows the patterns that will be saved in the PHT (Y-P method) as new results are shifted into the 2-bit PHT entry from the right. The 4th row shows the output of a 2-bit counter fed the 2-bit PHT entry. The predictions from these outputs are incorrect most of the time. This example only shows that the results of the Alpha and Yeh-Patt methods differ — no further conclusions can be drawn from this unrealistic example.

Slide 30 Address Prediction

The predicted target address if found from the **Branch Target Buffer** (BTB). The BTB is organized as cache with LRU eviction (the oldest, least recently used data is deleted to make room for new branch targets to be stored). The BTB index is the PC of the branch instruction. The entry is the last verified target address, a status (valid) field, and a special field used to indicate low confidence in the predicted address.

Slide 31 DLX Pipeline with Y-P Branch Prediction

Slide 31 shows the addition of Yeh-Patt two-step branch prediction to the IF hardware in the DLX. The standard DLX IF stage appears below.



With the addition of the branch prediction, the **cond** flag is used to control storage of branch history and target address (from NNPC — the calculated target address). The current PC is fed to the prediction logic. When the address is recognized as a previously observed branch instruction, the predicted target is used as NPC, which provides the next instruction fetch address to the PC register. If the current PC is not recognized, the default PC + 4 is used as the next PC.

Slide 32 – 33 Low Confidence Address Predictions

Certain branch instructions are unusually **difficult to predict**, especially indirect branches and returns from function calls, traps, and interrupts.

The Intel x86 ISA permits branches of the type **`JMP [ESI]`** which loads reads a data memory address from a register, and loads the program counter with an instruction address stored at in data memory. Because the register and / or data memory can change at run time, the target address for the branch can be highly dynamic, with no observable pattern. RISC ISAs provide a similar function in the jump register instruction. The Pentium M introduced a cache mechanism to follow indirect branches of this type.

A return from a subroutine can be difficult to predict, because a given function may be entered from various points in a program. The return point depends on the point in the program that called the subroutine.

Slide 33 shows a proposed (but not yet implemented) method for handling return branches.

In the x86 architecture, a subroutine call stores the return address in the hardware stack. A call instruction can trigger copying of the return address to the BTB for use on the return branch.

In a RISC ISA, function calls are implemented with the JALR / JR instruction pair. This permits the system to identify a function call and copy the return address to the BTB.

Slide 34 Exotic Branch Mispredictions

Because of the loop structure of most programs, simple branch prediction is accurate in about 90% of all branches. Of all branches, about 80% of branches (especially loop iterations) have an accuracy of nearly 100%, while 20% of branches have a 50% accuracy (as good as a random guess).

The least accuracy is found on the one-time test of a single operand, typically found in branches for if-then-else structures. In a loop of 1000 iterations that performs `if (x == 0) {...}` in each iteration, the loop branch can be predicted with high accuracy, but value of variable x is unknown until tested.

A proposal (not yet implemented) method for handling this situation is to maintain a separate cache of operand values that are tested. Any update to a tested operand will update the prediction table for the associated branch instruction.

Slide 35 – 36 DLX Memory Hierarchy

Slides 35 and 36 show the relationships among memory units in the DLX. The Instruction Memory in stage IF and the Data Memory in stage MEM of the pipeline are level 1 (L1) cache. The L1 cache memories are connected to the cache controller. When a memory address (PC or data address formed in the ALU) is not located in L1 (L1 cache miss), the controller simultaneously accesses level 2 (L2) cache in the DLX package and Main Memory (through the external I/O controller). If the memory location is found in L2 cache (L2 cache hit), the Main Memory access is cancelled. If the location is not found in L2 cache (L2 cache miss), then the location is copied to L2 and to L1.

Slide 37 – 38 Cache Organization

Cache memory holds a copy of a section of main memory. If main memory contains 4 GB of memory locations, with a 32-bit address space of 00000000 to FFFFFFFF (hexadecimal), then L1 cache typical contains a copy of 64 KB of these locations while L2 cache contains a copy of 5 MB of these locations. In order to find the required locations in the cache, they must be copied in an orderly fashion according to a well-defined **cache organization**.

To support cache, the memory space is divided (logically, not physically) into **blocks** (also called **lines**). Each block contains B bytes, where B is a power of 2. In most cases, the page size in virtual memory (under OS control) is an integer multiple of B.

Similarly, the cache memory is divided (logically, not physically) into S **sets** and each set is divided (logically, not physically) into W **slots**. Each memory block copied to cache must be placed into a specific set, but it can be placed into any unused slot in the set. There are thus W different ways to place a block in a set, and so W is called **associativity**.

A slot holds 1 data line, so that

> **Size of the cache** = B bytes per line × W slots per set × S sets per cache.

In detail:

> The memory space is given as $N = 2^n$ bytes, where the address is an n-bit number.
>
> Each line has $B = 2^b$ bytes. The line number is given by the integer part of N / B, the upper n – b bits of the address. The lower b bits of an address is called the **byte offset** (the count from 0 to B – 1 of the byte location in the line).
>
> The cache is logically divided into $S = 2^s$ sets. The **set index** is the set number to which the line must be copied in cache. The set index = (line number) mod S = lower s bits of the line number.
>
> The line can be copied to any of W slots in its appropriate set.
>
> The **upper n – (b + s) bits of the address** is called the **tag**. When a line is stored in cache, the tag is written to a table for the set. The CPU looking for a memory address can find the line number and set number by the known method. To search for the line in cache, the CPU must search all the tag numbers listed in the table of the appropriate set.

Slide 39 Cache Definitions and Policies

The CPU accesses cache, either by sending the PC to instruction cache or by sending an address calculated in the ALU to data cache. When the cache line containing the address is found in cache, this is called a **cache hit**. If the cache line is not found in cache, it is a **cache miss**.

Every **read miss** (a cache miss on a read access) causes the line to be copied to cache. A **write miss** can be handled in one of two ways (policies):

> **Write allocate** policy means that the line is loaded to cache to be written. Most write misses are handled this way.
>
> **No write allocate** policy means that the address is written directly in main memory, without copying the line to cache.

If a cache set is full (all the slots hold valid cache lines copied from main memory), but a new line must be added to the set, one of the lines in the set must be **swapped out**. This means that a line must be copied back to main memory and overwritten with a new cache line. The line chosen for swapping out is the **least recently used** (LRU) — the line whose last access was earliest (oldest). Two policies are relevant to swapping out:

> **Write back** policy allows the CPU to write to cache without updating main memory. Main memory is only updates when the line is swapped out. This policy simplifies memory organization and is used on most cache writes.
>
> **Write through** policy requires the CPU to update main memory on every write to cache.

Slide 40 – 42 Cache Performance Issues

The performance of cache memory depends on the number of clock cycles required for the CPU to access a memory location (data or instruction). On an L1 cache hit, the CPU memory access takes place in just 1 clock cycle, as described in RISC implementation. But on an L1 cache miss, the CPU stalls while L1 cache copies the missing line from L2 cache or main memory.

The **miss rate** is the relative number of cache misses per cache accesses. The miss rate depends on the cache size and organization, the size of the program and data structures. The **miss penalty** is the cache update latency, the number of stall cycles while the cache copies the missing line from L2 or main memory.

A CPU with a two-level cache is shown on slide 41. The miss rate and miss penalty can be different at L1 and L2. Since L1 is split into instruction (I) and data (D), the respective miss rates can be different. The parameters for the hit and miss rates are written in the following notation:

M_{L1}^{I} = Instruction Miss Rate at L1

$1 - M_{L1}^{I}$ = Instruction Hit Rate at L1

M_{L1}^{D} = Data Miss Rate at L1

$1 - M_{L1}^{D}$ = Data Hit Rate at L1

M_{L2} = Miss Rate at L2

1-M_{L2} = Hit Rate at L2

The miss penalties at L1 and L2 are written:

P_{L1} = Miss Penalty at L1 P_{L2} = Miss Penalty at L2

Analysis of the two-level cache can be simplified by the following assumptions (as shown in the probability tree diagram):

1. L1 instruction cache is accessed for every instruction fetch
2. L1 data cache is accessed for every data load or data store
3. On an L1 hit there is no penalty
4. On an L1 miss, the CPU accesses L2 cache.
    a. On an L2 hit, the total cache penalty is P_{L1}.
    b. On an L2 miss, the L2 must access main memory with an additional cache penalty given by P_{L2}.

The CPI^{stall} is stall cycles per cache miss (miss penalty) × cache miss per access (miss rate). There is one instruction fetch access at L1 (I) for every fetched instruction, which misses with the probability $M_{L1}{}^{I}$. There are ($IC^{data\ access}$ / IC) data accesses per instruction, which miss with probability $M_{L1}{}^{D}$. The weighted average miss rate is

$$\text{Cache miss per access} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{data\ access}}{IC} \right]$$

The miss penalty is P_{L1} on a simple L1 miss, with the addition of P_{L2} when L2 also misses (with probability M_{L2}. Together the average miss penalty is

$$\text{stall cycles per cache miss} = \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

Together,

$$CPI_{cache}^{stall} = \frac{\text{cache miss stall cycles}}{\text{instruction}} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{\text{data access}}}{IC} \right] \times \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

This expression demonstrates the advantage of the two-level cache system. If there is only one level of cache, the cache penalty is L1 to main memory, about 50 clock cycles. In two level cache, an L1 miss causes a short 5 CC miss penalty to L2 and an additional 50 CC penalty to main memory only on the small number of L2 misses. If the miss rate at L2 is 1%, then the miss penalty in two-level cache is 5.5 CC compared to 50 CC for the one-level system.

Slide 43 – 46 Issues Affecting Miss Rate

The causes of cache miss can be divided into three groups.

A **compulsory miss** occurs because updates to cache only occur on the first access to a cache line (known as a **demand access** update). This means that the first access to any cache line must be a miss. Cache size and associativity have no effect on compulsory misses.

A **capacity miss** occurs because a line that was previously in cache was swapped out to make room for another cache line. The larger the cache, the more room for many lines, which lowers the capacity miss rate.

A **conflict miss** is similar to a capacity miss — a line that was previously in cache was swapped out to make room for another cache line in a specific set, even when there is spare capacity in some other set. The conflict miss rate is lower when the cache has greater associativity, because there is more room for lines in each set. If the total cache size is fixed, then a cache with fewer sets must have more slots per set. But this makes locating a cache line for complex, because more cache tags must be searched.

The graph on slide 44 shows measured miss rate as a function of cache size and associativity. For all associativity levels, cache miss rate falls as cache size increases, as expected. At any given size, the cache miss rate drops with increased associativity (this is the effect of conflict miss). But for common cache sizes (16 KB and above), there is little advantage to associativity above 4. As seen on slide 45, larger associativity means fewer sets, a smaller set index, and so a longer tag number. So, more associativity means search through more slots per set to find a longer tag, making associativity above 4 inefficient.

The tags can be searched in three ways:

> In **serial access** the tag controller searches the tag for each way, one by one. This trades longer latency for lower power.
>
> In **parallel access**, the tags controller searches all tags at the same time. This trades shorter latency for higher power.
>
> In **way prediction**, the way is predicted in the manner of branch prediction, based on patterns of previous accesses.

Slide 47 Cache Prefetching

The **demand access** method in which cache is updated in response to a cache miss involves latency associated with the cache penalty. The method of **prefetching** "hides cache latency" by updating cache before a miss, and bringing the memory block to cache before it is needed.

Data cache prefetching is useful for sequential accesses to large data structures. It can be implemented by a software prefetch mechanism, in which assembly language **prefetch instructions** perform the cache update. These instructions can be inserted by the programmer or automatically by the compiler. The MIPS architecture was seen to provide a prefetch instruction. Prefetch can also be implemented in hardware with a **stride buffer** that automatically predicts sequential accesses.

Instruction cache prefetching must use a hardware-based prefetch mechanism, which predicts instruction cache updates based on program structure.

Slide 48 – 49 Data Cache Latency Example

Consider a loop over some operation involving a large data structure, such as a 1024 by 1024 element array of 16-byte SSE operands. The data structure is 16 MB and each element is accessed one time. A Pentium 4 processor with a 64-byte cache line can hold 4 operands, so that every 4 operands there is an L1 data cache miss — a total of 256 Kmisses.

The software prefetch on Intel CPUs fetches 2 cache lines (128 bytes), which provide 8 SSE operands. If the line is already in cache, the prefetch instruction is cancelled to a NOP. The code fragment on slide 49 identifies and loads the next 2 data lines for each data line in the current loop iteration. The prefetch hides the latency by initiating the cache update while performing useful work on the current data.

Slide 50 – 54 General Stream Buffer

In the context of cache, a **stream** is a set of successive accesses to sequential cache lines. A stream buffer is a FIFO buffer that generates and buffers update requests for successive cache lines.

The stream buffer can be implemented in two ways:

> A request buffer receives the requested line from the higher memory level and copies it directly to cache.
>
> A line buffer keeps the requested line in the buffer until the demand miss.

In both cases, the stream buffer performs request filtering — it skips lines that are already in cache, in order to minimize prefetches.

Slide 51 provides more details. On clock cycle k, the stream buffer is allocated to missed line L. The miss penalty causes latency while line L is updated. On subsequent clock cycles, k + 1, k + 2, and so on, the stream buffer requests line L + 1, L + 2, and so on. There are no additional miss penalties following the miss of line L.

Slide 52 shows a numerical example. The content of cache is shown in the first line. Access to lines 2 and 3 are hits, but the access to 4 is a miss. This generates update requests for lines 4, 5, and so on. No update requests are generated for line 6 and 8 because these lines are already present in cache.

A **stride buffer** operates in the same manner as a stream buffer, except that subsequent cache updates are separated by the stride s.

Slide 54 shows a stream buffer added to the DLX MEM stage. A data cache miss in L1 triggers the stream buffer to request sequential cache updates through the cache controller to L2 and main memory.

Slide 55 – 56 Out-of-Order Fetch

The **Miss Information Status History Register (MSHR)** is a variation of the stream buffer for instruction cache. An instruction queue is inserted between the IF stage and the ID stage. On a cache hit, the fetched instruction is placed in the queue. On a cache miss, a pointer is to the MSHR is placed into the queue instead of the instruction. The MSHR acts as a stream buffer initiating cache updates. When the cache line becomes available, the instruction queue copies the instruction instead of the pointer. An example is shown on slide 56.

Slide 56 – 57 Fetch Directed Instruction Prefetching

When branch prediction is used, it can also be used to predict instruction cache prefetches. In this method, instead of inserting an instruction buffer between the IF and ID stages, an address buffer is inserted between PC and the IF logic. The instruction address sent by the branch predictor to the PC is placed into the buffer. Search logic immediately checks the tags in the L1 cache to see if the required line is in cache. If the line is missing, then L1 cache can initiate the cache update immediately, while earlier instructions are being fetched according to addresses in the queue. Slide 58 shows the DLX with the prefetch included in the IF stage.

Presentation 5

Advanced Techniques for Avoiding Misprediction

ההצלחה של ביצוע ספקולטיבי הביאה למצב בו כל misprediction יקר מאוד וגורר תהליך התאוששות מורכבת אחרי ביצוע מספר רב של פקודות לא נכונות. שתי טכניקות מתקדמות למניעת misprediction הן trace cache ו- predication. השיטה הראשונה שומרת סדרות ארוכות של פקודות שהתבצעות בעבר, כולל פקודות סיעוף. השיטה השנייה מחליפה פקודות סיעוף עם ביצוע מותנה, שיטה עדיפה עבור סיעופים שקשה לחזות.

Trace Caching

Slide 2 **Branch Mispredictions Limit ILP**

In order to achieve high performance, superscalar CPUs require a high degree of instruction level parallelism (ILP). Speculative execution, including branch prediction and cache prefetch, is a key technique in providing a large number of independent instructions for parallel execution. With fast decoding, long speculative instruction sequences fill the cache and are moved to the ROB for out-of-order execution.

This means that a **misprediction** causes a very **long recovery process**, involving the cancellation of a large block of executed instructions, the eviction and replacement of many instruction cache lines, and the **decoding and rescheduling** of many instructions in the ROB. Instead of smoothly advancing the **predict-prefetch-decode pipeline**, the CPU enters a recovery mode during which the EUs starve (stall) with no available work. This is called a predict-prefetch-decode bottleneck.

Slide 3 – 4 **Trace Cache**

A **trace** is a long instruction sequence in dynamic execution order, the order of execution determined by the ROB (not the program listing order). A trace may include multiple branch actions (taken/not-taken). Trace cache stores instruction traces constructed dynamically on first run of the instructions in each trace.

With trace cache, branch prediction can be implemented as trace prediction for rapid recovery from a misprediction. If a predicted trace evaluates as a misprediction and the correct prediction is a different trace in cache, then recovery simply involves copying the cached trace to the ROB. This process skips repetition of the slower process of instruction fetch, branch prediction, decoding, and dynamic rescheduling.

The trace cache is **constructed** on the first run of a set of instructions. These instructions undergo the usual pipeline stages: loading of cache lines, instruction fetch, branch prediction, decoding instructions into the instruction pool, and scheduling the instructions for dynamic execution in EUs. A **trace cache entry** is started on the target or fall-through of a branch instruction. The trace embeds the next few branch actions, with the instructions in validated taken and not-taken execution paths. The entry ID is the PC of the first instruction and a flag representing the sequence of taken and not-taken actions embedded in the trace.

Slide 5 – 8 **Trace Cache**

Slide 5 shows a program fragment consisting of 8 program blocks (PB0 to PB7) and 4 branch instructions: two while loops (based on conditions C2 and C4) and one if-else control structure (one based on conditions C1 and one unconditional).

The red arrow shows the behavior of the conditional branch at the end of the **while(C4) loop**, and the purple arrow shows the behavior of the **while(C2) loop**. The blue arrow shows the behavior of the **if(C1)** structure. The green arrow shows the program skipping IB5 (unconditionally) when condition C1 is false.

Slide 6 shows that the program has 32 instructions — either branch instructions or members of instructions blocks. The runtime sequence with particular values of the conditions C1 – C4 is shown on slides 7 and 8. First IB0 and IB1 complete. Because **if(C1 = 0)** is false, the program continues with IB2 and IB3. IB3 runs 4 times in a loop, followed by IB4 and IB6, then back to IB1.

The trace cache is assembled beginning with the fall-through of the first branch, which is IB2.

Trace T1 contains IB2, IB3, J(C2).

Trace T2 contains IB3, IB3, J(C2).

Trace T3 contains IB3, IB4, J(C3=1).

Trace T4 contains IB6, J(C4), IB1, J(C1).

The correct loop sequence is T1 T2 T3 T4.

The branch at the end of a trace is predicted, pointing to a new trace. The trace is loaded and validated while running.

At the end of T1, the taken prediction of J(C2) points to T2. (valid)

At the end of T2, the taken prediction of J(C2) points to T2. (misprediction)

T2 is reloaded, but this is mispredicted — IB3 would run 5 times instead of 4.

The ROB cancels the instructions of T2, and the valid trace T3 replaces T2.

At the end of T3, the unconditional branch points to T4. (valid)

At the end of T4, the not taken prediction of J(C1) points to T2 (valid).

At the end of T2, the updated not taken prediction of J(C2) points to T3. (valid)

Slide 9 – 11 Trace Cache Header

A trace cache entry consists of an entry header and the instructions in the trace. The header is shown on slide 9. The fields are:

Start PC to identify the first instruction of the trace.

Flag indicating the sequence of branch actions in the trace.

Mask to show which flag positions hold valid information.

End type to indicate the type of the last instruction in the trace, conditional or unconditional branch, or not a branch (fall-through is next PC by default).

Fall-through / target provides the PC for next instruction fetch.

Trace headers for the example are shown on slide 10.

In order to make cache updates efficient, the length of a trace cache entry is set to the length of a cache line. The number of branches that can be stored in a trace is limited by the speed of the branch predictor. A trace is always terminated (filled with NOPs) at a subroutine return, because the return address is difficult to predict.

Slide 12 – 15 Instruction Fetch Unit

Slide 12 shows the relationship of **trace cache** to the **instruction fetch logic** and **branch prediction**. The next-PC logic predicts individual branches and the **multiple branch predictor** (MBP) predicts the multiple branches within traces. A new PC address is sent to instruction cache and the MBP in parallel. If the PC misses in the trace cache (no trace is found starting at that address), then the instruction is fetched from instruction cache, and the MBP updates the PC register. If the PC hits in trace cache, then the access to instruction cache is cancelled and a trace is fetched to the instruction pool. The MBP must pick the trace by predicting the multiple branches stored in multiple trace entries starting at the same PC.

The single branch predictor (**next-PC logic**) is a one-level or two-level branch prediction system of the type discussed in presentation 4. The **multiple branch predictor** predicts m branches that are embedded in a trace. The prediction is an m-bit bitmap — each bit is set as the last prediction made for that branch in the single branch predictor. When the single branch predictor makes a PC prediction that hits in trace cache, the MBP reads the bitmap for the branches embedded in that trace to predict the taken / not taken sequence, which chooses the particular trace in cache starting with that PC.

The BHR and PHB for the trace example is given on slide 14. The MBP is indexed by the PC, which is always a target or fall-through instruction.

The trace cache for the Intel Nehalem architecture is shown on slide 15.

Branch Predication

Slide 17 Another Look at Branches

The application of branch instructions can be divided into two basic contexts with distinct characteristics, **loops** and **if-conditions**. These contexts are compared in the table on slide 17.

In a simple loop, a conditional branch appears at the end of a long code sequence and jumps backward in order to reuse the code on a stream of data. The loop can also be built with a conditional forward branch (that breaks from the loop) at the top of the code and an unconditional branch at the end. The branch instructions are a small part of the loop code and are highly predictable. Branch prediction performs well on loops.

In an if-condition, the branch generally jumps forward to navigate among alternative code blocks based on a number of dynamic conditions to be tested. The branch instructions are often a high proportion of the code block, and are difficult to predict. Branch prediction functions poorly with if-condition structures, especially on superscalars that may be executing multiple branches per clock cycle.

Predication is an alternative to prediction.

Slide 18 Predication

An ISA that implements **predication** defines a set of 1-bit predication registers p0,...,pN that can be set by ALU instructions. A predicated instruction is run conditionally, so that the code

```
<p1> Instruction
```

causes Instruction to be executed if **`p1 = 1`**, or Instruction to be cancelled (NOP) if **`p1 = 0`**. This permits the conversion of if-condition branches to conditional execution, eliminating the branch stall at the cost of canceling extra work. For example, the code

```
     BNEZ R1, L1
     SUBI R1, R1, #1
L1:  ADD R2, R2, R1
```

skips the SUBI instruction unless R1 is zero. The equivalent predicated code is

```
      p1 = CMP R1, 0
<p1>  SUBI R1, R1, #1
      ADD R2, R2, R1
```

Which cancels the SUBI instruction unless R1 is zero, which sets **`p1 = 1`**. There is no clear advantage to the predicated form on a simple scalar CPU — the advantage is seen on a superscalar with dynamic scheduling.

Slide 19 – 21 Predication in IA-64

IA-64 is Intel's 64-bit ISA for the Itanium series processors. It defines 16 predicate registers, where p0 always equals 1. There are 48 virtual predicate registers for automatic renaming.

The general syntax of an IA-64 instruction is

`[(qp)] mnemonic[.comp1][.comp2] dests = srcs`

where **`qp`** is an optional predicate, **`.comp`** is a comparison, and **`dests`** can be predicate registers.

Slide 20 shows a simple switch control structure in C, with case 'a' distinct from case 'b'. The compilation for IA-32 (x86 ISA) is:

```
        MOV  EAX, [type]          ; copy type to register
        CMP EAX, 'a'              ; compare register to 'a'
        JNE L1                    ; if not equal then skip to next compare
        ADD EAX, 10               ; if equal, add 10 to register
        JMP L2                    ; skip to end
    L1: CMP EAX, 'b'              ; compare register to 'b'
        JNE L2                    ; if not equal then skip to end
        ADD EAX, 20               ; if equal, add 20 to register
    L2: MOV [type], EAX           ; store new value to type
```

The code has 9 instructions with 3 branch instructions that cannot be predicted because they run only one time. The branch prediction is not-taken for all three. There is one misprediction (stall) for either 'a' or 'b' and 2 mispredictions for neither.

The compilation for IA-64 is:

```
    mov  r1 = type                ; copy *type to register
    ld4  r2 = [r1]                ; copy type to register
    cmp.eq  p1 = 'a',r2           ; p1 = result of compare register to 'a'
    cmp.eq  p3 = 'b',r2           ; p3 = result of compare register to 'b'
    (p1)  add r2 = 10,r2          ; if p1 = 1, add 10 to register
    (p3)  add r2 = 20,r2          ; if p3 = 1, add 30 to register
    st4  [r1] = r2                ; store new value to type
```

There are 7 instructions, with no branches and no mispredictions. After scheduling for the Itanium, two instructions are executed in CC3, but one or both will be cancelled. If type is not 'a' or 'b' then no useful work is performed in CC3 — the equivalent of a stall. The IA-64 version removes 1 stall for each outcome.

Slide 23 Predication in ARM

The ARM ISA defines predicated versions of certain instructions that are executed conditionally by testing the status flags. The instruction SUBS performs SUB and sets the flags. The instruction ADDNE performs ADD only if the Zero flag is not set (not equal). The instruction SUBEQ performs SUB only if the Zero flag is set (equal). No branches are required.

Slide 23 Predication in PIC

PIC is a family of microcontrollers widely used in embedded systems. The PIC ISA defines instructions that test conditions and conditionally cancel the next instruction. The assembly code on slide 24 shows an implementation of the switch:

```
switch (PIR1)
{
  case 'TMR1IF':
    T1_INT;
    break;
  case 'ADIF':
    AD_INT;
    break;
  case 'LCDIF':
    LCD_INT;
    break;
  case 'RBIF':
    PORTB_INT;
    break;
  default:
    INT_ERROR_LP1;
    break;
}
```

The assembly code cancels the GOTO (branch) instructions when the condition is not matched:

```
BTFSC PIR1, TMR1IF      ; Cancel next if (PIR1<TMR1IF> == 0)
GOTO T1_INT             ; branch to Timer1 ISR
BTFSC PIR1, ADIF        ; Cancel next if (PIR1<ADIF> == 0)
GOTO AD_INT             ; branch to A/D ISR
BTFSC PIR1, LCDIF       ; Cancel next if (PIR1<LCDIF> == 0)
GOTO LCD_INT            ; branch to LCD ISR
BTFSC PIR1, RBIF        ; Cancel next if (PIR1<RBIF> == 0)
GOTO PORTB_INT          ; branch to PortB ISR
GOTO INT_ERROR_LP1      ; default ISR
```

Slide 24 – 33 Generalized Predication Model

The predication model can be generalized by setting multiple predicate registers per compare instruction. In this syntax,

`p1 = CMP.type1 src1,src2 <p>`	comparison type1 sets register p1
`p1,p2 = CMP.type1.type2 src1,src2 <p>`	comparison type1 sets register p1
	comparison type2 sets register p2

Complex conditions can be encoded into the predicate registers by predicating the comparisons with <p> and by modifying predicates after being set.

The syntax shown on slide 25, where a comparison is notated

```
C = CMP.type1.type2
```

In addition to the usual arithmetic comparisons, the type may include the comparison C (type T) or its logical complement C' (type F). The type also specifies the assignment:

U (unconditional)

The predicate register is always set.

Its value is set to **`(p && C)`** for type UT or to **`(p && C')`** for UF.

C (conditional)

The register is only set if **`p = 1`** — otherwise it is left unchanged.

The register is set to C for CT or C' for CF.

O (OR)

If **`(p && C) = 1`** for OT the register is set to 1.

If **`(p && C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

A (AND)

If **`(p && !C) = 1`** for OT the register is set to 0.

If **`(p && !C') = 1`** for OF the register is set to 0.

Otherwise, it is left unchanged.

∨

If **`(p || C) = 1`** for OT the register is set to 1.

If **`(p || C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

∧

If **`(p && C) = 1`** for OT the register is set to 1.

If **`(p && C') = 1`** for OF the register is set to 1.

Otherwise, it is left unchanged.

Additional instructions convenient for manipulating predicate registers are shown on slide 26.

Slide 27 shows an example in which a sequence of 5 instructions is used to set predicate registers p1 and p2. The first instruction sets p1 to the value of comparison C1. The second instruction overwrites p1 to 1 on condition C2. The third instruction overwrites p1 to 0 on condition C3. The fourth instruction sets p2 to 1. The fifth instruction overwrites p2 to 0 if either condition C4 is 0 or p1 is 0.

Slide 28 shows a short while loop in C and its compilation for IA-32. The gcc compiler uses two branch instructions — the first runs once (mispredicted) and the second runs on every iteration (mispredicted on the first and last iteration). The predicated version unconditionally sets p1 to the loop condition and cancels the loop instructions when it is no longer satisfied. It uses one branch that runs on each iteration and misses once (it is cancelled on the last iteration). In dynamic scheduling with speculative execution, the pointless execution of the cancelled instructions can be less costly than a misprediction.

Slides 29 and 30 show two similar while loops with one additional condition. The compilation uses an additional branch to implement the compound condition. The predicated version implements the compound condition by unconditionally setting p1 to the first condition and then conditionally overwriting p1 with the second condition.

Slide 31 shows a similar while loops with 3 conditions — the compilation uses 4 branch instructions to implement the compound condition. The predicated version implements the compound condition by unconditionally setting p1 to the first condition and then conditionally overwriting p1 with the second and third conditions.

Slide 32 shows a while loop with an if-condition in each iteration. The IA-32 compilation requires 5 branch instructions:

```
        mov [x], 0x0        ; x ← 0
        jmp L3              ; start at comparison
L1:     mov eax, [x]        ; eax ← x
        mov [y], eax        ; y ← eax = x
        add [x], 0x1        ; x++
        mov eax, [x]        ; eax ← x
        and eax, 0x1        ; zero all but low order bit of eax
        test eax, eax       ; compare eax with zero
        jne L2              ; if not zero skip to L2 (skip the if block)
        mov eax, [y]        ; eax ← y
        add eax, [x]        ; eax ← x + eax
        mov [z], eax        ; z ← eax
        jmp L3              ; skip to L3 (skip the else block)
L2:     mov eax, [x]        ; eax ← x
        sub eax, [y]        ; eax ← eax - y
L3:     cmp [x], 0x9        ; compare x < 9
        ja end              ; if x > 9 jump to end
        cmp [y], 0x4        ; compare y > 4
        jna L1              ; loop if y < 5
end
```

Slide 33 shows the predicated implementation. The loop is again controlled by setting a predication register p1 with a condition and conditionally overwriting it after a second

comparison. The if-condition is handled with two new predicate registers p2 and p3, which are set by the true or false result of comparing register EAX with zero. Either the instructions predicated with p3 or the instructions predicated with p3 will be cancelled.

Slide 34 – 35 Hyperblock Formation

The **conversion** of an if-else control block from branching to predication is shown on slide 34. The typical if-else block begins with a **Head** that ends with a conditional branch to **if-block B**. Below the Head is **else-block A**, which ends with a jump to skip over the **if-block B** to the **Tail**.

The first step in conversion is to convert the conditional branch to a predicate set as p and p', and to predicate the jump to B with p and the block A with p'.

The second step predicates B with p, and copies the Tail, one predicated with p and one with p'.

The last step is to reorder the blocks.

> If p = 1, the jump to B, B, and the Tail for B will run. A and its copy of Tail will be cancelled.
>
> If p = 0, the jump is cancelled, along with B and its copy of Tail. A and its copy of the Tail will run.

The program now has only one absolute branch, which is only mispredicted on its first run. In a superscalar with trace cache, both A+Tail' and B+Tail are in trace cache after 1 run, so that speculative execution can run with high efficiency and minimum misprediction recovery.

Hyperblock conversion can be performed as a compiler optimization for an ISA that defines predication. Since the x86 does not permit predication, it is not found in the PC market.

Slide 36 – 37 Multipath Execution

In multipath execution, the block predicated by p and the block predicated by p' can run in parallel. Both blocks can even be scheduled speculatively before p is evaluated. When the value of p is found, one block is cancelled in the ROB and the other block can be retired and committed to state. A simple example is shown on slide 36.

The table at the top of slide 37 shows an execution example on a superscalar DLX. If R3 is zero I_1 runs as fall-through with no stall, followed by J I_3 that has a 1 CC stall. If R3 is not zero, then BNEZ executes with a 1 CC stall jumping to I_2.

The table at the bottom of slide 37 shows the same CPU with predication. All 4 instructions can be run in parallel. When the result of the first instruction is found, either I_1 or I_2 is cancelled in the ROB.

Presentation 6

Queuing Theory: Memory Buffer Limits on Superscalar Processing

תורת התורים עוסקת בהתנהגות מערכות שבהן חוצץ (buffer) יוצר תור של בקשות לטיפול. מערכות כאלו נפוצות מאוד במחשבים, מהתור לגישות לדיסק הקשיח ועד למערכות שרת/לקוח מבוזרות. ניתן להתייחס למפענח (decoder) במעבד כלקוח ששולח בקשות לשירות ליחידות הביצוע (EUs) שפועות כשרת, באמצעות ה- ROB שמתפקד כחוצץ. על פי המודל, תורת התורים נותנת חסמים על היעילות של שיטת superscalar.

Slide 2 – 5 Client/Server Model of I/O

Queuing theory assumes that a fast CPU acts as a client for the services of a slower device over an I/O channel (input/output). The buffer stores client requests and forwards them to the server at the slower server response rate. Some basic definitions are:

Latency	The time between the client request and the server response.
Throughput	The actual number of services provided per unit time by the server.
Bandwidth	The maximum data transfer rate on the I/O channel (including the buffer).
Capacity	The maximum throughput of the server receiving requests through buffer (depends on the bandwidth and the service rate).
Utilization	The request rate as a proportion of capacity.

The client makes requests to the server intermittently. Most of the time the client makes no requests, but while requests are made the volume of requests per second can reach the peak capacity of the client. The client's peak request rate is much higher than its average request rate over a long time. This is called **bursty** behavior (the client emits requests in short, high speed bursts). Generally, the peak client request rate is also higher than the server's maximum service rate.

The FIFO buffer stores client requests in order to forward them to the server at the maximum rate that the server can accept them. The buffered requests form a **queue**.

The **minimum latency** is determined by the server's maximum service rate. Since the service rate is expressed as services per second, the latency is the inverse, expressed as seconds per service. The minimum of seconds per service is the time required to service one request.

If the client continues to fill the buffer with requests at a rate faster than the service rate of the server, the buffer will eventually overflow. The **net buffer fill rate** = fill rate – empty rate. For example, a CPU can send a hard disk 10^9 read requests per second. The disk can only respond to 10^7 read requests per second (it takes 10^{-7} seconds to service one request). If the buffer can hold 1000 read requests, and the CPU makes requests continuously, the buffer will overflow in about $1000 / (10^9 – 10^7)$ seconds.

As utilization grows, the client makes more requests per second. But the server cannot work faster, so more requests are held in the buffer (the queue length grows). The total latency for any given request is the time required to process that request plus the time required to process the requests in the queue when the request is made. As the queue grows longer, so does the latency for any given request. A basic result of queuing theory is that **latency and utilization grow together**, so that low latency requires low utilization.

Slide 6 Queuing Theory

Queuing theory makes the following **assumptions**:

1. **Client Requests** arrive independently and obey Poisson statistics. Each request has a random length (bytes to transfer) and requires a random service time.
2. The **buffer** stores requests and forwards them in order of arrival (FIFO) at the maximum available service rate. Over a long time, the buffer reaches an average "**steady state**" level of stored requests.
3. The server provides services to each request independently with a **service time** obeying Poisson statistics.

The **principal results** are:

1. Average latency = 1 / (average service rate – average arrival rate).
2. Average buffer level = average arrival rate × average buffer latency (Little's Law).

Slide 7 – 16 Queuing Theory Details

Slides 7 to 16 show the details of the **derivation** of the basic queuing theory results.

Slide 7 reviews the Poisson statistics for randomly distributed independent events. The average arrive rate is given by α (the Greek letter alpha).

Slide 8 describes the client requests as Poisson events with an average request rate given by λ (the Greek letter lambda) and describes the servicing of a request as Poisson events with an average response rate of μC (the Greek letter mu times C).

Slide 9 shows the possible transitions for a buffer whose level (number of buffered requests) is given by n at time $t + \Delta t$:

1. The level at t was n – 1 and 1 request arrived during the time interval Δt
2. The level at t was n + 1 and 1 request was serviced during the time interval Δt
3. The level at t was n, 0 requests arrived and 0 requests were serviced during the time interval Δt
4. The level at t was n, 1 requests arrived and 1 requests were serviced during the time interval Δt

Using the rules for independent events, the total probability of buffer level n at time $t + \Delta t$ is given on slide 10 as the sum of these four possibilities. Each possibility is the probability of the previous buffer level at time t and the probabilities of new requests and requests serviced.

Slide 11 makes a linear approximation of the probabilities for very small interval Δt.

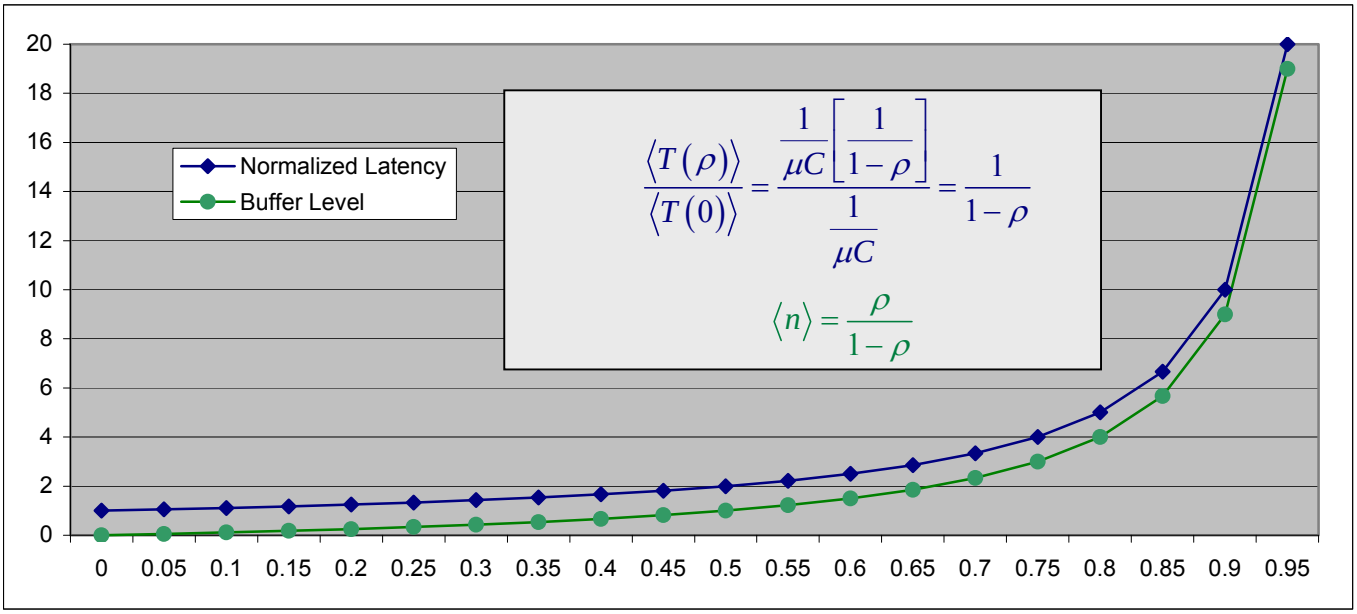
Slide 12 uses the steady state assumption to show that the probability of a 0 to 1 transition must exactly balance the probability of a 1 to 0 transition. Using the approximate expressions for the probabilities, this leads to $P_1 = \rho P_0$, where ρ is the system utilization.

Slide 13 uses the steady state assumption for the general case, following a few algebraic manipulations to set the time-derivative of the probability to 0 (steady state ⇒ no change). This leads to a recursion relation for P_{n+1} in terms of P_n and P_{n-1}.

Slide 14 inserts the relationship between P_1 and P_0 to show that $P_n = \rho^n P_0$ for all n. Since the sum of all probabilities must sum to 1. The probability of buffer level n is found to be $\rho^n(1-\rho)$.

On slide 15, the average buffer level is shown to demonstrate the results claimed on slide 6.

Slide 16 shows the relative values of buffer level and latency as a function of utilization.



Slide 17 Memory Queuing

Slide 17 shows three **examples of a buffer queue** within a single computer system.

The memory ordering buffer (**MOB**) in the x86 microarchitecture buffers data writes to L1 data cache, so that writing can be interrupted to perform higher priority reads.

The sustained throughput (maximum long term write speed) to a **SATA 3.0 hard drive** is 6 Gbps, which is slightly less than 600 Mbytes per second. The sustained long term write speed of the PCI Express 4.0 I/O bus is slightly less than 16 Gbytes per second. The typical write buffer on a SATA hard drive is 64 MB, so that a typical PC I/O system can overflow the disk buffer in 4.16 ms.

External hard drives, especially with a USB interface, must be safely removed before disconnection. Safe removal guarantees that the most recent writes to open files are written to the external device from the USB buffer, and the files are marked closed before disconnection. Unsafe removal can leave unwritten data in the buffer, leading to data loss.

Slide 18 Client/Server Model of CPU

A superscalar CPU can be modeled as a **client/server** system. The fetch and decode stage is modeled as a client that sends λ requests (instructions) per clock cycle to a server (EUs) through a buffer (the ROB). The **buffer level** is the number of instructions in the ROB waiting to be executed. The **instruction window** is the total number of instructions in some stage of

execution in all the EUs in any clock cycle. The **maximum service (execution) rate** μC in the EUs is one instruction window per clock cycle.

The goal of the superscalar CPU is to maximize the CPU utilization $\rho = \lambda / \mu C$ for a fixed maximum execution rate μC without overflowing the ROB. This means making the average decoder rate λ as close as possible to the execution rate μC (but always smaller).

Slide 19 – 20 General Superscalar Model

A **general superscalar** is made up of several execution units (EUs). Each EU may be pipelined with a number of stages. Slide 19 shows an example of a CPU with 2 ALUs (each ALU has 1 stage), 1 load/store EU (2 stages), and 1 FPU (3 stages).

In this example, there can be $2 \times 1 + 1 \times 2 + 1 \times 3 = 7$ instructions in some stage of some EU at the same time. For this CPU, the **instruction window** is 7.

The general case is shown on slide 20, where u_i is the number of EUs for instruction type i, and s_i is the number of pipeline stages in an EU for type i. The instruction window is given by

$$IC_{EU} = \sum_i u_i \times s_i = \text{total pipeline stages in CPU}$$

Slide 21 – 23 Queuing Theory and Decoder Rate

The service rate μC for the EUs is the total number of **instructions per cycle** (IPC) it can execute. For instructions of type i, with each EU operating in parallel,

$$IPC_i = u_i \times IPC_i^{scalar} = u_i / CPI_i^{scalar} = u_i / (1 + CPI_i^{stall})$$

where CPI_i^{scalar} is the number of cycles per instruction on a scalar (single pipeline) processor.

Summing over instruction types, the service rate for the EUs is

$$\mu C = IPC = \sum_i IPC_i = \sum_i \frac{u_i}{1 + CPI_i^{stall}}$$

The queuing theory result is that the **average ROB level** (number of instructions in the ROB waiting to execute) is

$$\langle n \rangle = \frac{\lambda}{\mu C - \lambda}$$

where $\mu C = IPC$. To keep the EUs busy, we need to have the ROB level approximately equal to the instruction window IC_{EU}. Setting

$$IC_{EU} = \frac{\lambda}{IPC - \lambda} \qquad \rightarrow \qquad \lambda = \frac{IC_{EU} \times IPC}{1 + IC_{EU}} = \frac{s \times u \times IPC}{1 + s \times u}$$

Neglecting CPI^{stall}, IPC = u, and so the ideal decoder rate is approximately

$$\lambda^{ideal} = \frac{s \times u^2}{1 + s \times u}$$

Slide 22 shows that in a CPU with 2 integer ALUs (1 stage), 2 FPUs (3 stages), 2 load/store units (2 stages), the instruction window is 12 instructions and the decoder should provide and average of 5.54 decoded instructions per clock cycle (the approximate number in an x86 CPU).

Slide 23 considers the **scaling behavior** of this model. Scaling up from 6 to 15 EUs with and average of 8 superpipelined stages, the decoder would need to provide around 15 instructions per clock cycle, and maintain an instruction window of 120 independent instructions.

Of 120 instructions, around 25 to 30 instructions would be branches, leading to a large probability of mispredictions, and therefore a decoder bottleneck. For this reason, larger superscalar CPUs are not practical.

Presentation 7

Beyond Instruction Level Parallelism

בעשור האחרון שיטת superscalar הגיעה לגבולות התחום שלה והיכולת לשפר ביצועים. באותן השנים השתפרו טכנולוגיות שמאפשרות ייצור מספר רב של ליבות בשבב סיליקון אחד, וניצול הליבות על ידי חלוקת תוכנה לפתילים (threads). מצד אחד, הטכנולוגיות האלו מעודדות מעבר מדגש על מקבילות ברמת הפקודות (ILP) לדגש על מקבילות ברמת הפתיל (TLP). ומצד שני, תכנות מקבילית היא לא תחום חדש ועדיין סובלת מבעיות תיאורטיות שלא גילו להן פתרון בחמישים השנים שעוסקים בה. הפרק הזה עוסק בשיטות כלליות בתכנות מקבילית. טכנולוגיות פרטניות לארכיטקטורות מקבילות יוצגו בפרקים 8 ו- 9.

Slide 2 – 3 Summary of Superscalar Processing

Slide 2 shows a general pipeline for a superscalar CPU core with dynamic scheduling, and summarizes the techniques introduced for recent generations of RISC processors to improve performance:

In the IF stage, branch prediction and trace cache minimize branch penalties.

In the IF stage, prefetch minimizes cache miss penalties.

In the ID stage, virtual registers and automatic register renaming prevent false data dependencies and permit greater ILP.

In the ROB, out-of-order execution and in-order retirement allow instructions to be executed in the most efficient order while maintaining atomic semantics at the level of machine code.

Multiple execution units permit multiple instructions to be executed in every clock cycle.

A stream buffer minimizes cache miss penalties in the data cache access, and a memory reorder buffer permits efficient scheduling of loads and stores.

Slide 3 reviews the difficulty in scaling these superscalar methods.

Slide 4 – 7 Sequential and Parallel Operations

The state of the art in programming and hardware implementation meet in an awkward manner at the level of machine language. Most high-level programs combine **parallel and sequential constructs**, using model-dependent sections: processes, threads, classes, procedures, and control blocks. But these sections are compiled for a particular ISA made up of low-level CPU operations: data transfers, arithmetic/logic operations, and branch operations. When the high-level job is executed, the CPU executes machine instructions, small sequential operations with only local information about 2 or 3 operands. The CPU cannot recognize the abstract model-dependent structures used by the programmer — the information about inherent parallelism is lost in translation to the CPU. The sophisticated hardware techniques described on slide 2 can be seen as attempts to rediscover some of the parallelism intended by the programmer, using heuristics such as prediction, renaming, cache, and inherent parallelism.

Concurrent tasks are defined to run independently, whether in parallel (executing simultaneously on multiple copies of hardware) or interleaved (a single hardware unit alternates between tasks). For example, a program may respond to mouse events, respond to keyboard input, and accept a network message concurrently.

Concurrency can be divided into two types:

In **functional concurrency**, each task performs a different set of operations. For example, the rotation of a vector in the plane maps the vector A to A' = R(θ) × A where R is a rotation matrix. As a mathematical abstraction, this rotation takes place concurrently, but the sequential implementation in code performs two slightly different operations

$$A_x' = A_x \cos\theta + A_y \sin\theta$$

$$A_y' = -A_x \sin\theta + A_y \cos\theta$$

In **data concurrency**, each task performs the same operation on different data. For example, the sum of two vectors is performed concurrently as the mathematical abstraction C = A + B, but the implementation in sequential code performs the operations

```
for (i = 0, i < n, i++) C[i] = A[i] + B[i]
```

While high-level languages permit the definition of concurrent structures such as vectors and matrices, compilation for a sequential machine language loses most of the information about concurrency. The hardware applies heuristics in order to recover some of this lost information.

In **parallel programming**, the programmer identifies inherently parallel operations in the high-level job, designs a **parallel algorithm** and translates the job into source code that specifies parallel operations to the compiler. Parallel threads for are defined for functional and data decomposition. The OS receives deterministic instructions reflecting the inherent parallelism, including code and threading instructions. The hardware disperses instructions defining as vectorized operations, pre-grouped independent operations, and thread level parallelism, to multiple processors or execution units.

Slide 8 – 9 The "Old" Parallel Processing

Early R&D in the field of parallel processing experienced growth and decline:

1958 Research was begun at IBM on parallelism in arithmetic operations

1960 – 1980 Mainframe computers were designed as symmetric multiprocessors (SMP) with 4 to 24 CPUs. The OS dispatches processes from a shared ready process queue to idle processors.

1980 – 1995 A research boom led to a deeper understanding of parallel processing, but limited practical success:

Automated parallelization by the compiler achieved limited success. Compilers cannot easily identify inherent parallelism.

Parallel constructs in high-level languages require a long learning curve. Parallel programmers are typically specialists.

Inherent complexities were identified:

Processing and communication overhead includes inter-process message passing (to spawn and assemble tasks with many CPUs) and synchronization to prevent race conditions (data hazards).

Data structures for the shared memory model must be defined with good blocking to cache organization, to avoid a cache bottleneck.

1999 It was fashionable to consider parallel processing a dead end that would never provide good performance in a reasonable code development cycle.

The graph on slide 9 shows the relative number of papers on multiprocessors (in red) and their interconnection networks (in green) submitted to the International Symposium on Computer Architecture. As seen in the graph R&D reached a peak around 1980 to 1995, and dropped off sharply after 1996.

Slide 10 It's Back — the "New" Parallel Processing

The new push toward multicore processing was prompted by the limits in earlier strategies for improving CPU performance. As seen in slide 3, superscalar growth has reached practical limits.

Although clock speed tripled from 1 GHz to 4 GHz between 2000 and 2005, faster speeds face physical limits. The time required for an electrical signal to cross a 3 cm of empty space at the speed of light is one clock cycle of a 10 GHz clock, making time synchronization difficult. Another problem is removal of heat from the CPU core. Digital circuits conduct electrical current during 0 to 1 and 1 to 0 transitions, so faster clock rates mean higher power consumption. As chips get larger, it becomes more difficult to remove the heat from the center of the CPU, causing damage to the circuits.

The "old" parallel processing technology, based on assigning tasks to processors in physically separate chips is still not sufficient. The new developments that make multicore CPUs viable include the low price of placing several cores in the same physical package, compiler support for program decomposition into lightweight threads along with debugging tools, and OS level thread scheduling.

Slide 11 – 13 Processes and Threads

A **process** (task) is one instance of an independently executable program. It was the basic unit of OS kernel scheduling on a traditional non-threading kernel. A process is associated with an entry in a process control block (PCB) that defines resources: ID, state, PC, register values, stack and memory space, I/O descriptors, among others. A task (context) switch is a high volume transfer operation. A process may be organized into one or more threads owned by the process.

A **thread** is one instance of an independently executable instruction sequence that is not organized into smaller multitasked units. Threads own limited private resources — a PC, a stack, and its registers values. Other resources are shared with other threads owned by the parent process. Threads may be scheduled by the OS kernel or by user code. A thread switch is a low volume transfer operation.

One standard example of a multithreaded program is a web server. The server listens on an open socket and creates a new socket for each accepted client connection. Since client connections are independent the server can create a new thread to handle the new client.

The multithreaded web server is an example of **functional decomposition**. In a 3D game, functional decomposition could be the creation of independent threads for handling the ground, the sky, and a character. Higher performance can often be achieved with **data decomposition**, such as the division of the sky into n sections and the creation of an independent thread to update each section.

Slide 14 – 17 Hardware Implementation of Multithreading

Multithreading does not impose any special hardware requirements. Multithreaded code can run on a single core or multicore processor. The run-time efficiency depends on the interaction between hardware and software. The five basic approaches to multithreading are:

Coarse-grained multithreading	A single CPU switches to a new thread on a long stall
Fine-grained multithreading	A single CPU switches to a new thread on each clock cycle
Simultaneous multithreading (SMT)	A superscalar pools instructions from multiple threads in the ROB, creating a large instruction window
Hyper-Threading	Intel technology combining fine-grained multithreading and SMT
Multiprocessing	A scheduler dispatches threads to multiple independent cores

The upper illustration on slide 15 shows the utilization of EUs on a **single threaded superscalar** CPU over time. In CC1 (at left), two EUs execute instructions in parallel while two EUs are idle. In CC2, three EUs work in parallel while one EU is idle. In CC3, only one EU performs useful work. IN CC7 and CC8, the program is stalled and no EU is actively executing an instruction.

The center illustration shows the utilization of EUs on a **coarse-grained multithreading** CPU over time. On CC1 to CC4, thread 1 (blue) runs, utilizing 2, 3, 1, and 4 EUs. On CC5 to CC7, thread 2 (green) runs, utilizing 2, 4, and 3 EUs. On CC 8 to CC10, thread 3 (yellow) executes, and on CC11 and CC12, thread 4 (pink) executes.

The bottom illustration shows the utilization of EUs on a **fine-grained multithreading** CPU over time. On CC1 thread 1 (blue) runs, on CC2, thread 2 (green) runs, On CC 2, thread 3 (yellow) runs, and so on.

Notice that course-graining and fine-graining can usually insure that the pipeline is kept busy on a stall, but cannot insure high EU utilization.

Slide 16 shows the utilization of EUs on a **simultaneous multithreading** CPU over time. The ROB holds many instructions from many independent threads. Because the threads are independent, there are no data dependencies between instructions of different threads, creating a large instruction window. When a thread is cancelled following a misprediction, only the instructions associated with that thread are deleted from the ROB — about 75% of the instructions in the ROB remain valid, eliminating much of the decode bottleneck (slide 2 in presentation 5).

Slide 17 shows Intel Hyper-Threading technology (see slide 55 in presentation 3).

Slide 18 – 20 Thread Coexistence

Multithreaded code provides a source of independent instructions, permitting high processor utilization on a single SMT superscalar core. Although independent applications running in parallel should produce no data dependencies, their independence can actually create memory resource conflicts. For multiple threads running, each application requires different data blocks in to be in cache, and each creates a different branch prediction cache and trace cache. Also, parallel threads of a single application run in a coordinated fashion, communicating, synchronizing, and exchanging data. A stall in one thread can stall related threads, through cache misses, page faults, branch mispredictions, and so on.

The helper thread model is intended to coordinate memory conflicts. Helper threads perform no useful work or change any program result — their results not committed to memory. They perform loads and branches that appear in the work of other threads. The CPU encounters cache misses in the helper thread and updates cache, before the working thread requires the data. As an example, the code at the top of slide 20 suffers a cache miss on the load and a branch misprediction on the branch at the end of the loop. But pairing this code with a helper thread, that merely performs a load from the same data address and branches on the same instruction, preloads the data block to cache and prepares the branch predictor to recognize the taken branch.

Slide 21 Flynn Taxonomy for CPU Architectures

See slide 4 in presentation 3 for a discussion of the Flynn Taxonomy.

Slide 22 – 25 Multiprocessor Architecture

The top of slide shows the structure of a typical **SISD / SIMD workstation**. The central feature is the bus adapter chipset (usually PCI). The adapter is connected at left to a dual core CPU, connected at right to main memory, and connected below to the I/O bus that connects peripheral hardware (storage, user interface, network connections, and so on).

A general **MIMD multiprocessor** contains multiple CPUs, each with multiple cores. The cores are connected to main memory (localized or distributed across the system) through some interconnection network (also called a fabric). The network may be as simple as a data bus (such as PCI) or a specialized switching fabric. The network also connects the CPUs and memory to the I/O system. Multiprocessors are usually characterized by the relationship of CPUs to memory, a relationship that also determines the software parallelization model.

A **shared memory system** is shown at the top of slide 23 (discussed at length in presentation 8). In shared memory, the global memory space A may be physically partitioned into M blocks (0 to M – 1) but the full memory appears unified through the switching fabric. Each of the N processors (0 to N – 1) can access the full address space (address 0 to A – 1) via the internal network. Threads running on the N CPUs communicate among themselves by writing and reading shared addresses in memory. For many activities, shared memory requires synchronization of memory accesses to prevent data hazards.

A **message passing system** is shown at the bottom of slide 23 (discussed at length in presentation 9). A node consists of a CPU with its private address space A (0 to A –1). The multiprocessor is made up of N nodes (0 to N – 1). CPUs cannot generally access the memory of any other node. Processors communicate by formulating, passing, and reading inter-CPU messages over the internal network. The messages combine data and memory synchronization information.

Slide 24 shows the Flynn-Johnson Taxonomy, a revision that distinguishes shared memory from message passing, and global memory (located in a centralized location) from distributed memory. In this framework, most shared memory machines are either GMSM or DMMP.

The characteristics of the shared memory model and the message passing model are compared in slide 25. In shared memory systems, the synchronization of memory accesses, implemented by maintaining the coherence of cache in the multiple CPUs is a source of communication overhead. In message passing, cache coherency is not required, but the passing of messages contributes a similar overhead. A major difference between models is scalability. Shared

memory systems are limited in scale by the exponential complexity of shared access by multiple CPUs to the memory system. The sharing also requires the CPUs to be physically close, and is applicable to fine grain parallelism — light, parallel threads with short code length and small data volume. Message passing is limited only by network capacity (the CPUs may be separated by thousands of miles) and is applicable to course grain parallelism — heavy parallel threads with long code length and large data volume. Shared memory programming is facilitated by pthreads and the OpenMP model that defines parallel constructs for C/C++. Message passing programming is facilitated by the Message Passing Interface (MPI) API.

Slide 26 – 30 Amdahl's Law for Multiprocessors

Amdahl's equation for parallel processing was obtained on slide 25 of presentation 1. It is derived again on slide 26 by defining IC_P (the number of instructions that can be run in parallel) as $F_P \times IC$, where F_P is the fraction of the program that can be run in parallel. The remaining part of the program, $1 – F_P$, runs sequentially.

Slide 27 shows results obtained by Intel for the speedup of multiprocessor systems operating on programs written to handle **on line transaction processing** (OLTP). Without Hyper-Threading (which contributes a 20% improvement as discussed in slide 56 of presentation 3), the speedup is found to be 1.7 for N = 2 CPUs and 2.6 for N = 4 CPUs. Inverting Amdahl's equation to find F_P one finds that about 80% of the instructions can be run in parallel. Slide 26 shows that with this proportion, the maximum speedup as $N \rightarrow \infty$ is S = 5, corresponding to an ideal CPI = 0.20.

Some additional details on OLTP are given on slide 28. The transactions processed in OLTP are client requests to a server accessing a database, for example, banking requests, order processing, inventory management, accesses to the student information system, and so on. The requests are inherently independent and so the workload in the server is inherently multithreaded, with one thread per request. The server sees a large batch of small parallel threads. The code is generally short and inherently sequential — SQL transactions made up of accesses to multiple tables. The multi-table accesses produce a severe memory latency of several CPU stalls per thread. Running the SPEC Cint benchmark suite on a certain 8-pipeline dynamic scheduling superscalar led to a measured CPI = 0.31 (equivalent to $CPI^{stall} = 1.48$, counting idle EUs as stall cycles). Running an OLTP load on the same hardware leads to a measured CPI = 1.27 (equivalent to $CPI^{stall} = 9.16$). General purpose, server-oriented CPUs (such as Intel's Xeon family) do not run OLTP efficiently. For this reason, Oracle (whose database software specializes in OLTP) purchased Sun Microsystems (once the leader in high performance workstations) to obtain the SPARC CPU, better suited to simultaneous multithreading (SMT) than general purpose dynamic scheduling.

A measure of multiprocessor efficiency is found from $E = S / S_{ideal}$ where S is actual speedup and S_{ideal} is the speedup for $F_P = 1$ (full parallelization). It is shown on slide 35, that unless F_P is exactly 1, the efficiency goes to zero for a large number of processors. This simply means that throwing a lot of processors at a problem with a significant sequential component is not efficient.

Slide 31 – 32 Grosch's Law versus Amdahl's Law

Long before Amdahl's work, Herbert Grosch at IBM studied computer performance and found that the performance-to-price ratio improves linearly with price (more performance per dollar at higher prices). This is formalized as

$$\text{Cost}(N) = \alpha \times N \quad \Rightarrow \quad N = \text{Cost}(N)/\alpha$$

$$\text{performance}(N) = k_G \times (\alpha \times N)^2 = k_G \alpha^2 N^2 \quad \Rightarrow \quad \frac{\text{performance}(N)}{\text{cost}(N)} = k_G \alpha^2 N$$

$$S = \frac{\text{performance}(N)}{\text{performance}(1)} = N^2$$

But Amdahl's law implies that

$$\text{performance}(N) = \frac{k_{Amdahl}}{(1-F_P) + \frac{F_P}{N}} = \frac{k_{Amdahl}}{(1-F_P) + \frac{F_P}{\text{Cost}(N)/\alpha}} = \frac{\text{Cost}(N)}{(1-F_P)\text{Cost}(N) + \frac{\alpha}{k_{Amdahl}} F_P}$$

$$\frac{\text{performance}}{\text{Cost}(N)} = \frac{1}{(1-F_P)\text{Cost}(N) + \frac{\alpha}{k_{Amdahl}} F_P} \quad \text{for some constant } k_{Amdahl}$$

So, by Amdahl's law the performance-to-price ratio should decline with rising cost (diminishing returns from throwing a lot of processors at a problem that has a significant sequential component).

The difference between Amdahl's and Grosch's results can be understood by looking at two different types of computability problem.

On the one hand, a typical problem running in a shared memory environment will initialize a few variables, execute a loop over a shared data set of MB scale, and display the result. The initialization and display will always execute sequentially, even if every loop iteration is handled in parallel on a separate core. No additional improvement can be expected by making the CPU number larger than the number of loop iterations.

On the other hand, a typical problem running in a message passing environment will initialize a few variables, execute a long loop over a data set of GB to PB scale (1 PB = 1Mega GB), and display the result. Such problems include large simulations, weather forecasting, and analysis of astronomical data. Because these data sets can usually be decomposed into effectively independent sets (with periodic exchange of data at the boundaries between sets), message passing problems typically scale with the size of the problem. The Gustafson-Barsis Law applies to large problems in which the number of processors grows with the size of the problem. The run time is expressed as a sequential part and a part that grows with problem size.

$$\text{run time in serial execution} = s + p \times n \text{ , } n = \text{size of problem}$$

$$\text{speedup compared to serial execution} = \frac{s + p \times n}{s + p} \xrightarrow[n \text{ large}]{} n$$

As a proportion of instructions to run, the sequential part becomes very small as the problem (and number of CPUs) grows. The Gustafson-Barsis Law is equivalent to the Grosch result taking s = 1 instead of s = 2.

Slide 33 Interconnection Network Types

Slide 33 shows a taxonomy of **interconnection networks** for multiprocessor systems. The basic distinction is between static (permanent) connections between nodes and dynamic connections established on demand. Static connections can provide either full node-to-node connectivity or limited connectivity that requires multi-hop message transport.

Dynamic interconnection networks are divided into shared medium transport, typically a data bus, over which nodes must arbitrate (compete) for access to the network, and switched transport, in which the switching elements are specially configured for each message.

Slide 34 – 35 Communication Overhead and Amdahl's Law

Communication, whether for shared memory or message passing, creates parallelization overhead that affects performance. In analogy with instruction processing time, the time required for communication can be written as

$$T^{comm} = CPI^{comm} \times IC_p \times \tau = CPI^{comm} \times F_p \times IC \times \tau$$

CPI^{comm} = processor clock cycles devoted to communication per instruction executed in parallel

F_p = fraction of program running in parallel

Defining

$$F_{overhead} = \text{overhead factor} = CPI^{comm} / CPI,$$

which characterizes the ratio of communications activity to computing activity, the speedup compared to 1 CPU becomes

$$S = \frac{CPI \times IC}{CPI \times IC \times (1 - F_p) + \frac{CPI}{N} \times F_p \times IC + CPI^{comm} \times F_p \times IC} = \frac{CPI}{CPI \times (1 - F_p) + \frac{CPI}{N} \times F_p + CPI^{comm} \times F_p}$$

$$= \frac{1}{(1 - F_p) + F_p \left(\frac{1}{N} + F_{overhead} \right)},$$

a modified form of Amdahl's equation. For a large number of processors,

$$S = \frac{1}{1 - F_p (1 - F_{overhead})}.$$

When the overhead factor approaches 1, meaning that communication activity is comparable to computing activity, the speedup drops to 1, even for a large number of CPUs.

Presentation 8

Shared Memory Architecture

מודל זכרון משותף הוא צורת העבודה בתחנת עבודה בעלת מעבד רב-ליבות, סוג המחשב הנפוץ ביותר היום. המודל הזה מתאים למקבילות ברמת הפתיל (TLP) כהמשך ישיר של שיטות ה- ILP במעבד superscalar. מספר ליבות (בדרך כלל 2 עד 8) מחלקות ביניהם פתילים שמטפלים בחלק ממבנה נתונים אחד, על פי חלוקה פונקציונלית או חולקת נתונים. כיוון שכל הפתילים פועלים על אותה קבוצת נתונים שיושבת בזכרון משותף, חייבים למנוע סיכונים (data hazards) עם סנכרון מתאים.

Slide 2 – 4 Shared Memory Multiprocessor

In **shared memory**, the global memory space A may be physically partitioned into M blocks (0 to M – 1) but the full memory appears unified through the switching fabric. Each of the N processors (0 to N – 1) can access the full address space (address 0 to A – 1) via the internal network. Threads running on the N CPUs communicate among themselves by writing and reading shared addresses in memory. For many activities, shared memory requires synchronization of memory accesses to prevent data hazards.

Memory access to shared memory systems can be subdivided into three types. In uniform memory access (UMA), the whole memory is accessible by all processors, with uniform access time from any CPU to any memory location. In non-uniform memory access (NUMA), each CPU is physically attached to one part of shared memory. Access time to a given memory address depends on the distance to memory from the processor. In cache-only memory architecture (COMA), each processor has part of shared memory as private cache, without main memory. COMA requires that data be migrated from processor cache to processor cache for sharing.

Because of the rapidly decreasing cost of multicore systems, **symmetric multiprocessor** (SMT) is the most popular example of the UMA model. An SMT consists of a number of processors of an equivalent type with a symmetric relationship to main memory, as shown on slide 4. Each core is basically equivalent, and each has the same relationship and access time to main memory. Generally, each core has a private L1 cache, while the cores in a single package share a common L2 cache. In a certain deviation from pure SMT, the sharing of data among cores within one package is performed in L2 and is simpler than sharing between packages.

Slide 5 – 7 Open MP for Shared Memory Systems

Open MP is an API (application program interface) for multiprocessing designed to simplify thread-based parallelization for shared memory programming in C/C++ and Fortran. OMP provides directives for explicit parallelization, extending familiar constructs to a Fork-Join Model. Programs initiate as a single thread called the Master thread (also called the consumer thread) and execute sequentially until a parallel construct is encountered. The Master thread creates (forks or spawns) a team of producer threads that execute in parallel on multiple CPU cores. When the producer threads complete, a Join operation synchronizes and terminates the producer team, leaving the Master thread to continue. Forks can be nested within parallel sections.

The general code structure is shown on slide 6. The statement

#pragma omp parallel private(..., ..) shared(...) { ... }

defines a parallel team of threads with private (per thread) variables and shared variables. Slide 7 shows a partial list of options, including reduction (a joining of private producer results to a single Master result.

Slide 8 "Hello Worlds" Program

The "Hello Worlds" program for OMP is shown on slide 8. The **pragma omp parallel** defines the private variable tid used by each thread to get its thread ID using the OMP system call omp_get_thread_num(). Each thread displays "Hello World from thread = ID" for its thread ID number. The master thread (with tid = 0) obtains the total number of threads using the OMP system call nthreads = omp_get_num_threads() and displays the number. The Fork then terminates in a Join and the program returns to sequential execution.

Slide 9 – 11 Parallel For Directive

Within the **pragma omp parallel** block, the **pragma omp for** performs data decomposition of a standard C for loop. Slide 10 shows a program that initializes arrays a[N] and b[N] with N = 1000, and then performs

```
for (i=0; i < N; i++)    c[i] = a[i] + b[i];
```

in chunks of 100 over (1000 values / 100 values per thread) = 10 threads.

The **pragma omp parallel** directive defines a[], b[], c[], and chunk as shared, and defines a private value of i for each thread. The **pragma omp for** defines producer threads that are dynamic — when a thread completes, it is closed without waiting to synchronize. A new thread can run on the same CPU to perform another chunk of the work. Slide 11 illustrates data decomposition in graphic form.

Slide 12 Sections Directive

Within the **pragma omp parallel** block, the **pragma omp sections** performs functional decomposition of instructions within a standard C for loop. Slide 12 shows a program that initializes arrays a[N] and b[N] with N = 1000, and then performs

```
for (i=0; i < N; i++)    c[i] = a[i] + b[i];
for (i=0; i < N; i++)    d[i] = a[i] * b[i];
```

in two parallel threads. The **pragma omp parallel** defines a[], b[], c[N], d[N] as shared and defines a private value of i for each thread. The **pragma omp sections** opens a team of producer threads, and the **pragma omp section** defines the activity of each thread.

Slide 13 – 15 Race Conditions

A race condition is a data hazard caused by parallel access to shared memory. For example, the program fragment

```
#pragma omp parallel shared(x) num_threads(2) {  x = x + 1; }
```

should create two threads to independent increment x by 1 and provide the result x ← x + 2. But as shown on slide 13, if the threads execute as interleaved, the sequence of data reads and writes to the shared variable x produces the final result x ← x + 1. To prevent race conditions, synchronization methods can protect threads from conflict with other threads.

Four synchronizations defined in OMP are:

#pragma omp master	Only the master thread (thread 0) performs the following block.
#pragma omp critical	Only one thread can execute the following block at a time. Other threads wait for this thread to leave the critical section before entering.
#pragma omp barrier	Each thread reaching the barrier waits until all threads reach the barrier.
#pragma omp atomic	A memory update in the next statement must be completed atomically. This is a mini-critical section for memory writes.

Slide 15 shows how the race condition in the previous example can be corrected by defining the instruction x = x + 1 as critical. This way, the threads are made to run sequentially. The first thread to begin will complete before other threads read x.

Slide 16 Reduction

The reduction directive joins a list of private variables into a final result for the master thread. For example, the directive **reduction(+:sum)** adds the private copies of the variable sum from each thread into the master thread's copy. The example on slide 16 shows a team of producer threads, each one evaluating part of the scalar product of vectors a[] and b[]. The reduction combines the partial results into the full scalar product.

Slide 17 – 19 Bus-Based Symmetric Multiprocessor (SMP)

Most symmetric multiprocessor (SMP) systems are based on a PC workstation architecture in which processors are interconnected with main memory through a bus. Typically, one core is selected at system initialization time as the bootstrap processor (BSP) that performs system startup and runs OS functions. The other cores are designated application processors (AP), which remain in the halt state until the OS allocates application threads.

Two possible methods for sharing of cache between cores in a processor package are shown on slide 18. In the Pentium D package (Intel's first dual core x86 CPU), each core has private L1 and L2 cache. Threads running on the cores must update main memory to share data between them. In more recent x86 CPUs, each core has a private L1 cache and a common L2 cache, allowing memory sharing between threads without updating main memory. Slide 19 shows the organization of cache in the Pentium M (the last single-core upgrade of the Pentium 4) and in the Core Duo (dual core CPU). In the Pentium M, a cache manager (control unit) mediates between the external bus (to main memory), L2 cache, prefetch, and the core. The Core Duo introduces a fairness unit to insure that neither core suffers data starvation. The fairness unit connects both cores to the cache manager (control unit), which is now split between bus access functions and L2 access functions.

Slide 20 – 23 Communication Overhead in SMP

Performing the sum c[] = a[] + b[] sequentially requires 4 load-add-store operations. On a 64-bit CPU, each array is 4 elements × 8 bytes per element = 32 bytes, so that an optimizing compiler places the three arrays in a single 128-byte cache block, for a maximum of one cache miss.

Slide 20 shows an OMP routine to add the arrays in parallel. Running in parallel, each of 4 cores must perform a cache update to load the cache block (sequentially, over the serial bus), perform 1 load-add-store operation, and then update the main memory. Since all the cores are writing to the same cache block, this operation must be performed atomically. Cache updates take much more time than arithmetic operations, so that the parallel version will suffer a much longer run time than the sequential version. This is an extreme example of a common issue in SMP.

Another issue is the time to implement the reduce operation. The left side of slide 21 shows the sequential evaluation of the scalar product

$$\sum_{\mathtt{i=0}}^{\mathtt{3}} \mathbf{a}[\mathbf{i}] * \mathbf{b}[\mathbf{i}] .$$

Ignoring cache overhead, the sequential version executes 4 iterations × 4 instructions per iteration + 1 store = 17 instructions. The parallel code spawns 4 threads to perform 1 iteration per thread. Then, the reduce operation performs 2 parallel partial sums (another iteration) and then a final iteration over the two partial sums. The 3 iterations perform 12 instructions, and the speedup is 17/12 = 1.42. Inverting Amdahl's equation

$$1.42 = \frac{1}{1 - F_P + \frac{F_P}{4}} \qquad \Rightarrow \qquad F_P = \frac{1 - \frac{1}{1.42}}{1 - \frac{1}{4}} = 0.394$$

the overhead is equivalent to 39% of the program running in parallel, or 61% of the program running sequentially.

An estimate of the capacity of an SMP system can be found by requiring that interconnection network capacity be greater than the volume of exchanged data. The capacity can be written as

$$\text{Interconnection capacity} = R_n \times W$$

R_n = transfer rate (transfer cycles per second)

W = transfer width (bytes per transfer)

and the volume of data can be written as

$$\text{Exchange demand volume} = N \times D_x \times M$$

N = CPUs

D_x = average data access rate (bytes per second) per access

M = inter-CPU access rate = cache miss rate

The total volume of data transferred will be less than the interconnection capacity if the total number of CPUs satisfies

$$N \le \frac{R_n W}{D_x M}$$

For example, on a 1 GHz CPU with PCI express at 16 GB per second,

R_n = 1 GHz

W = 16 bytes per transfer

M = 0.1 (communication dominated miss rate)

D_x = 4 bytes per load × 0.25 loads per instruction × 10^9 instructions per second

The limit is $N \leq \frac{10^9 \times 16}{10^9 \times 0.1} = 160$ CPUs

Slide 24 – 26 Performance

Slide 24 shows a comparison of the Core Duo dual core and Pentium M single core processors on the (non-threaded) SPEC benchmarks Cint and Cfp. Since the code is not multi-threaded, this graph shows how one core of a Core Duo CPU compares with the single core of the Pentium M. On integer programs, the two CPUs are about equal in performance, while the Core Duo shows some speedup on floating point programs because of improvements to the FPU and cache.

Slide 25 shows a comparison of the CPUs on multi-threaded benchmarks. On some tests, the Core Duo is twice as fast as the Pentium M, showing the effectiveness of parallel execution.

Slide 26 compares the CPUs using data decomposition and functional decomposition. Generally, data decomposition provides a higher speedup, because it divides loop length. Functional decomposition divides some of the work among cores, but the loop length is unchanged and most of the work remains sequential. The results on functional decomposition depend strongly on the operating mode of the CPU. Performance optimization is the fastest CPU mode, used when the computer is connected to external power. Power optimization is a power-saving mode, used when the computer is running on battery power. In this mode, many CPU functions are shut down when not needed to save power. Under these conditions, functional decomposition is less effective than data decomposition, which finishes quickly permitting the CPU to be shut down.

Slide 27 – 28 Data Hazards in Shared Memory SMP

Preventing data hazards in shared memory systems requires analysis of three levels of multiprocessor behavior. At the **highest level**, programmers must use synchronization directives to avoid race conditions when performing inherently sequential operations in concurrent programs. As seen in OMP, synchronization permits atomic execution of individual read/write operations or whole program blocks defined as critical sections. At the **system level**, SMP systems must enforce cache consistency protocols to insure that multiple processors writing shared copies of data maintain valid copies of data values. At the **hardware/software** boundary, algorithms must take the complexity of real hardware operations into account. Although instruction-level memory semantics appear atomic, algorithms must be designed to function correctly if the pipeline level implementation does not enforce atomic reads and writes to memory.

The general approach to hazards is to choose an operational definition of **program consistency**, typically correct program execution under interleaving. Enforcement of program consistency enables unambiguous program validation and detection of data hazards.

Strict consistency holds when all memory operations are performed in the order intended by programmer, which is possible to implement only on a single processor system. **Sequential consistency** (Lamport 1979) refers to any clear, consistent, repeatable definition of execution order. A consistent concurrent program will produce an identical result in any execution for which each processor executes one instruction in the specified sequential order, according to

the order specified by program. Any well-defined interleaving, such as round robin, satisfies the sequential ordering.

The programmer must assume that unsynchronized threads execute interleaved, that cache enforces write-order consistency (as specified by the interleaving and program order), and that hardware enforces the consistent ordering of reads and writes.

Slide 29 Implementing Critical Section with Semaphore

Slide 15 showed how a race condition can be corrected in OMP by defining an instruction block as critical, so that the blocks are forced to run sequentially. In order to enforce criticality, some method of signaling must be employed.

One obvious possibility is the use of **semaphores** provided by the OS. A semaphore is an unsigned number s with two operations, P(s) and V(s). The operation V(s) increments the value of s by 1. If s > 0, then P(s) decrements s by 1, and otherwise P(s) waits synchronously (blocks subsequent instructions from executing) until some other process increments s.

A **mutex** (mutual exclusion) is a binary semaphore initialized to 1 and saturated at 1 (that is, if s=1, then V(s) = 1). As shown on slide 20, a mutex can be used to define an instruction block as critical by placing P(s) before the block and V(s) after the block. As soon as any CPU enters the block, all other CPUs find s = 0 and are prevented from continuing into the block. When that CPU completes the block it sets s = 1 so that another CPU checking s will be able to run.

It is not practical to implement the mutex using OS semaphores, because it would require disabling all interrupts across the system between P and V.

Slide 30 – 32 Shared Variable Lock

Another method of enforcing criticality is the use of shared variables as a form of signaling. A simple appears on slide 30, using flags initialized to 0. Two CPUs receive the threaded code

```
      Flag1 = 1;
loop: if (Flag2 == 1) loop
      <critical section>
      Flag1 = 0;
```

```
      Flag2 = 1;
loop: if (Flag1 == 1) loop
      <critical section>
      Flag2 = 0;
```

in which each thread sets its own flag and checks the other thread's flag. Superficially it appears that if one thread begins first, the other will be forced into a **spin loop** (a branch instruction that jumps to itself), until the first thread completes and clears its flag.

The program can be checked for interleaved execution by assigning times t1, t2, t3, t4. With round robin interleaving (one possible ordering), the instructions run as

```
      Flag1 = 1;
      Flag2 = 1;
loop: if (Flag2 == 1) loop
loop: if (Flag1 == 1) loop
```

Since both flags have been set in the first two instructions, the second two instructions will create a condition of **deadlock**. The idea of sequential consistency is that the program must function properly if it executes in an interleaved order.

The program may be corrected through repeated rewriting of the flag variables:

```
loop: Flag1 = 1;
      if (Flag2 == 1)
         {Flag1 = 0 ; loop}
      <critical section>
      Flag1 = 0;
```

```
loop: Flag2 = 1;
      if (Flag1 == 1)
         {Flag2 = 0 ; loop}
      <critical section>
      Flag2 = 0;
```

When this version executes interleaved, the instructions runs without deadlock as

```
      loop: Flag1 = 1;
      loop: Flag2 = 1;
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            if (Flag1 == 1) {Flag2 = 0 ; loop}
      loop: Flag1 = 1;
            <critical section>
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            Flag2 = 0;
      loop: Flag1 = 1;
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            <critical section>
            Flag1 = 0;
```

This version will run properly on most hardware, but is contains a subtle bug — it relies on the atomic semantics of the write to the flag variables.

Suppose that the assignment `Flag1 = 0` does not complete atomically in the hardware, so that the data read by the subsequent test `(Flag1 == 1)` has not been updated at read time. This could happen if both flags are bits in the same integer variable — when one thread writes its flag it will accidentally overwrite the flag of the other thread. This would lead to a **livelock** condition in which both CPUs keep updating their flags but never enter the critical section.

The Dekker algorithm repairs the bug associated with non-atomic hardware writes. A third variable **turn** is initialized to 1.

```
      Flag1 = 1;
      turn = 1;
loop: if (Flag2 == 1
            && turn == 1) loop
      <critical section>
      Flag1 = 0;
```

```
      Flag2 = 1;
      turn = 2;
loop: if (Flag1 == 1)
            && turn == 2) loop
      <critical section>
      Flag2 = 0;
```

When this version executes interleaved, the instructions runs without deadlock as

```
            Flag1 = 1;
            Flag2 = 1;
            turn = 1;
            turn = 2;
      loop: if (Flag2 == 1 && turn == 1) loop   // false
      loop: if (Flag1 == 1 && turn == 2) loop   // true
            <critical section>
      loop: if (Flag1 == 1 && turn == 2) loop   // true
            Flag1 = 0;
      loop: if (Flag1 == 1 && turn == 2) loop   // false
            <critical section>
            Flag2 = 0;
```

In this version, even if one of the flags is accidentally overwritten, exactly one of the conditional branches will execute as false. Modern cache coherency protocols prevent most such nonatomic memory errors.

Slide 33 – 35 Machine Language Support

Some ISAs define instructions that support program synchronization at the system level. Atomic machine language primitives provide a **hardware-level semaphore M** to enable implementation of critical constructs at the compiler level.

The most basic primitive is **test and set**. The semaphore **M** refers to an address in memory. The assembly instruction **Test_and_Set M, R** copies the value stored at address **M** to register R, and then sets **M** to 1 if its value had been 0. A variation is **fetch and add**, which writes the value stored in **M** to a register and then adds the contents of another register to **M**. Both are used with the instruction **swap** that exchanges the values in **M** and some register. The typical use of these instructions is

```
L1:  Test_and_Set M, R1 ; spinlock
     BNEZ R1, L1
     <critical section>
     Swap M, R1
```

If thread 1 and thread 2 run this code interleaved (with private copies of register R1 and M initialized to 0):

L1: Test_and_Set M, R1	R1 ← 0 M ← 1		
		L1: Test_and_Set M, R1	M = 1 R1 ← 1
BNEZ R1, L1	Not taken		
		BNEZ R1, L1	Taken
<critical section>	Execute		
		L1: Test_and_Set M, R1	M = 1 R1 ← 1
Swap M, R1	R1 ← 1 M ← 0		
		BNEZ R1, L1	Taken
		L1: Test_and_Set M, R1	R1 ← 0 M ← 1
–		BNEZ R1, L1	Not taken
		<critical section>	Execute
		Swap M, R1	R1 ← 1 M ← 0

Initializing R2 to 1, the identical behavior is obtained by replacing test and set with **Fetch_and_Add M, R1, R2**.

Slide 34 shows **compare and swap**, another possible machine instruction that can be more efficient than **test and set**. The instruction **CAS M, R1, R2** exchanges M and R2 if M equals R1 when tested. It also sets a flag when the exchange is made or clears the flag when the exchange is not made. When this instruction is used, all the CPUs may enter the critical section without restriction, but only one will be able to commit its result to state. The other CPUs will have to begin again.

A typical example is

```
L1:  LW R1, x          ; load x
     ADDI R2, R1, #1   ; prepare new value for x
     CAS x, R1, R2     ; if no change in stored x, update x
     BEQZ Cflag, L1    ; else start again
```

Any CPU can enter this section. But if the value of **x** was changed since being read by this CPU, then the CPU must cancel its work and begin again.

If thread 1 and thread 2 run this code interleaved (with private copies of registers):

L1: LW R1, x	R1 ← x		
		L1: LW R1, x	R1 ← x
ADDI R2, R1, #1	R2 ← x + 1		
		ADDI R2, R1, #1	R2 ← x + 1
CAS x, R1, R2	R1 = x x ← R2 R2 ← R1 Cflag ← 1		
		CAS x, R1, R2	R1 != x Cflag ← 0
BEQZ Cflag, L1	Not taken		
		BEQZ Cflag, L1	Taken
		L1: LW R1, x	R1 ← x
		ADDI R2, R1, #1	R2 ← x + 1
—		CAS x, R1, R2	R1 = x x ← R2 R2 ← R1 Cflag ← 1
		BEQZ Cflag, L1	Not taken

Slide 35 shows another possible pair of machine instructions that improve on CAS: **load reserve** and **store conditional**. This pair prevents a thread from updating a variable if it was written since being read in the thread, even if the new value was identical to the old value. The **load reserve** reads a memory location and writes to a flag. The **store conditional** instruction only writes to memory if the flag associated with the read is still set. A typical application of these instructions is

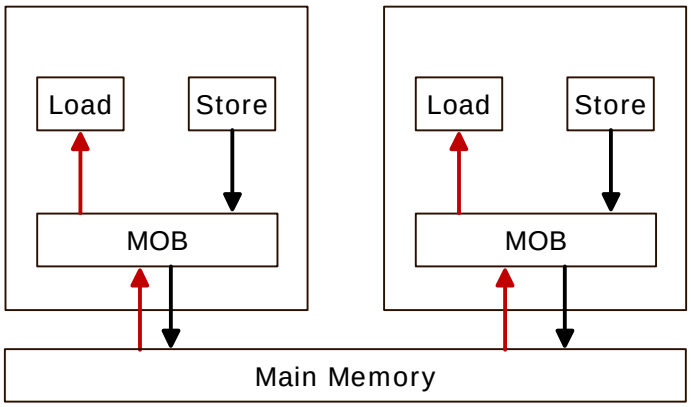
```
L1: load-reserve R1, x
    ADDI R1, R1, #1
    store-conditional x, R1
    BEQZ status, L1
```

If thread 1 and thread 2 run this code interleaved (with private copies of registers):

L1: ld-res R1, x	flag ← 1 adr ← &(x) R ← x		
		L1: ld-res R1, x	flag ← 1 adr ← &(x) R ← x
ADDI R1, R1, #1	R1 ← x + 1		
		ADDI R1, R1, #1	R1 ← x + 1
sto-cond x, R1	flag = 1 adr = &(x) flag ← 0 adr ← 0 x ← R1 status ← 1		
		sto-cond x, R1	flag != 1 adr != &(x) status ← 0
BEQZ status, L1	Not taken		
		BEQZ status, L1	Taken
–		L1: ld-res R1, x	flag ← 1 adr ← &(x) R1 ← x

Slide 36 – 40 Hardware Level Consistency without Cache

Hardware level optimizations in superscalar CPUs with dynamic execution can introduce hazards in seemingly harmless code, by changing the order of instructions. Slide 37 describes a hazard that can occur if memory writes are made through a memory ordering buffer (slide 17 in presentation 6) on the CPU. The write buffer on each CPU permits read instructions to bypass previous writes to main memory. A read hit to an address in the write buffer will update directly from the FIFO. But a read by one CPU from an address in the write buffer in a different CPU will not function properly.



For example, assembly code for two threads of the Dekker algorithm is

Thread 1		Thread 2	
	ADDI R1, R0, #1		ADDI R1, R0, #1
	SW [Flag1], R1		SW [Flag2], R1
	SW [turn], R1		SW [turn], R1
loop:	LW R2, [Flag2]	loop:	LW R2, [Flag1]
	LW R3, [turn]		LW R3, [turn]
	AND R4, R2, R3		AND R4, R2, R3
	BNEZ R4, loop		BNEZ R4, loop
	LW R5, [x]		LW R5, [x]
	ADDI R5, R5, #1		ADDI R5, R5, #1
	SW [x], R5		SW [x], R5
	SW [Flag1], R0		SW [Flag2], R0

The MOB in thread 1 could execute the LW instructions before the SW instructions that remain in the write buffer during the reads. Until the write buffer is emptied (the SW to memory performed), thread 1 will provide the value of turn from its write buffer (not knowing that thread 2 may have changed its value). Similarly, when thread 2 performs the LW R2, [Flag1] instruction, it will not know that a new value of Flag1 is sitting in the MOB of thread 1 waiting to update main memory. Both threads will execute incorrectly.

Slide 38 describes a similar problem that will occur if the physical write to a memory location takes several transmission hops over several clock cycles.

Slide 39 shows an example of another potential problem. In thread 1, the out-of-order execution may select **LW R5, [x]** to run early (it has no visible data dependencies to the single CPU). But this scheduling would read the value of x before thread 2 updated it, and the program would not execute properly. In slide 40, a **memory barrier** is inserted before the LW instruction. No instruction can be executed before the **MEMBAR** in a program listing.

Slide 41 – 44 Cache Consistency and Coherence

Slide 41 reviews the basic terminology for cache policies in a single CPU system. On a multiprocessor system, several processors access shared data in main memory, and each processor copies data blocks to its private cache. When a CPU writes to its private copy in cache, the cache and main memory become inconsistent. For example,

CPU-0 reads variable X to cache from main memory

CPU-1 reads variable X to cache from main memory

CPU-0 writes to variable X in cache

The copies of variable X in CPU-1 cache and main memory are not valid.

Strict consistency is the situation in which all copies of any data location are identical — this is almost impossible to implement. **Sequential consistency** is the situation in which processors perform operations in the specified sequential order, and the operations on each processor are performed in program order. **Cache coherence** implements sequential consistency for cache and main memory, which requires processor hardware support for cache management.

Slide 43 shows the sequence of instructions executed when a program runs in order under interleaving, but without enforcement cache coherence. In particular, the update x in the cache of CPU-0 without either updating other copies of x or alerting CPU-1 of the update, causes an execution error for the program. Slide 44 shows the same error in the case that the cache update by CPU-0 also updates memory, but does not alert CPU-1 that its cached copy is invalid.

Slide 45 – 52 Cache Coherence Protocols

The goal of cache coherence is to enforce sequential consistency for all caches and main memory, in order to permit system-wide atomic and critical constructs. The principle technique is called **Snoopy cache**. A status field is added to the tag table associated with every slot in every set in cache. When the cache manager searches for a memory block in cache (slides 45 – 46 in presentation 4), it checks both the tag, which indicates the presence of the block, and the status. The status bits are updated, depending on coherence policy, with the access history to the data.

The cache operates by **write-back policy** — changes to data in cache only update main memory when the block is swapped out of cache. This policy saves considerable work on the main data I/O bus. However, during every write to cache, the **address** being written in cache is **broadcast** on the I/O bus. Before writing to the I/O bus, a CPU must acquire access rights in competition with the other CPUs. Therefore, broadcasting the address to be written takes slightly more time than just writing to cache in an uncoordinated fashion, but takes much less time than performing a complete a data write.

The cache management system is called **snoopy** because it **monitors** all addresses written on system bus by all processors, compares written addresses with blocks in its own cache, and updates the state of cached block on an address hit. In this way, if CPU-0 and CPU-1 both copy variable **x** to cache, a write to **x** by CPU-0 only updates its private copy. But the write to **x** (according to its memory address) requires CPU-0 to acquire access rights to the I/O bus and broadcast the address of **x**. When the snoopy cache manager in CPU-1 sees the write to the address of **x**, it tags the copy of **x** in its own cache invalid.

Within snoopy cache there are two main types:

In **write update**, processor W places a write address on the I/O bus and snooping processors S_i tag their copies of the block as invalid on an address hit (if the block is in their private cache). Processor W then performs the write both to cache and to the bus. Processors S_i update their local copies of block from the I/O bus and tag the updated blocks valid. This insures a rapid update of all caches, but creates a very large bus overhead.

In **write invalidate with inquire**, processor W places a write address on the I/O bus and snooping processors S_i tag their copies of the block as invalid on an address hit. Processor performs the write to its local cache only, creating one unique valid copy of the data block. When another processor S_i needs the data clock, it performs an **inquire cycle**. S_i places the required address on the I/O bus, which forces processor W to perform a write-back and update the main memory. This method has much lower I/O bus overhead. Most shared memory systems implement write allocate with write invalidate.

Slide 47 defines the states that may be assigned to a data block in its status word.

Slide 48 shows the behavior of CPU W if a cache block is tagged as **modified**. The block is called **dirty** meaning that its value has been changed since loading from main memory (the copy in main memory is invalid). Other CPUs tag this block as invalid if it was loaded to their private caches (until it is overwritten by another block). W can continue to update this copy without updating main memory. W must responds to an inquire cycle — its cache manager snoops the memory address from some S_i that requests access to the clock. When W updates main memory for a write inquire, it also marks its own copy of the block invalid.

Slide 49 shows the behavior of CPU W if a cache block is tagged as **shared**. The block is called **clean** meaning that the value is the same as the value in main memory and other private caches. W can update this copy — it must write the update address to the I/O bus to alert processors S_i that mark their copies of the block invalid. After it updates its cache, W tags the block **modified**.

Slide 50 shows the behavior of CPU W if a cache block is tagged as **invalid**. The block must be reloaded before next read (as if it were not in cache). Any read or write access is a cache miss. Read misses are always loaded. On a write miss, the write may be performed after a load to cache (write allocate) or the write can be made directly to memory (no write allocate). Some memory locations (such as addresses used for peripheral I/O instead of data storage) are labeled "non-cacheable" and are never copied to cache.

Slide 51 shows the complete state transition diagram for the **MSI** (modified, shared, invalid) protocol. Slide 52 shows an example of a series of reads and writes of a single variable by two CPUs, and the status of the associated cache blocks. On the first read (line 1), the block is loaded to cache in processor P1 and marked shared (S). On the first write (line 2), the block is marked modifies in P1 (transition 2 from S to M). When P2 inquires to read (line 3), the block is updated to main memory and loaded to P2. P1 and P2 mark the block shared. When P2 writes (line 4), it marks the block modifies, and P1 marks the block invalid (I). And so on.

Slide 53 – 55 Processor Behavior — Exclusive

Exclusive is an additional state defined to make the protocol more efficient. When a block in the cache of processor W is marked exclusive, W cache holds the only valid copy of the data block among the caches. But the block is clean, identical to the copy in main memory. The copies in other processors S_i are marked invalid. If W updates this copy, it marks the block modified. As with modified, W responds to an inquire cycle. The exclusive status provides improved performance for private data — on eviction of an exclusive (E) block, the slot may be overwritten without any write-back to memory. Slide 55 shows an example of the extended MESI protocol.

Slide 56 – 58 Memory Thrashing with MSI

The use of a shared variable as a mutex together with the MSI protocol and snoopy cache can lead to a condition called memory thrashing (note: **th**rashing, and not **t**rashing). The value of the shared variable **mutex** is swapped with a local register in order to enforce criticality. Only the CPU that finds mutex = 0 can proceed to the critical section. But because mutex is a shared variable, each swap by a CPU causes: a read miss and a cache update, a write to one local cache (write allocate), and invalidation all other cached copies (write invalidate). These updates cause very inefficient movement of data across the I/O bus. Multiple mutex cache misses and loads create a large I/O overhead, much larger that the actual read and writes required to execute the critical section.

The main advantage of the load-reserve and store-conditional primitive (slide 35) is its compatibility with MSI. The shared variable mutex leaves most CPUs in a spin loop that involves multiple unrelated memory updates (for mutex) and memory thrashing (code on left of slide 57). The use of load-reserve and store-conditional replaces the spin loop with a sequence in which a CPU may have to repeat the critical section many times before it has the opportunity to commit its result to memory. But since this method (code on right of slide 57) performs many reads an only one write per CPU, the number of cache write updates is much smaller. When a CPU completes the critical section and writes to its cache, the other CPUs must invalidate their cache copies of shared variables, clear the reservation flag, and perform an inquire that causes the CPU to update main memory. Thus, all of the memory activity of the other CPUs takes place at the same time for the same reason — one CPU has succeeded in completing the critical section.

The memory thrashing caused by mutex is an example of a dependency called **false sharing**. This occurs when multiple threads access distinct address ranges within the same cache block, so that every write to the block causes block invalidation in other processors. An example is shown on slide 58, where a small array is used to hold unrelated variables accessed by separate threads. The small array is likely to sit in the same cache block, so that every element update invalidates the copies of every other CPU.

Slide 59 Cache Coherence and Hardware Optimizations

Slides 36 – 40 discussed several hardware optimizations in superscalar CPUs that can cause problems in shared memory systems. The MSI protocol that enforces cache coherence orders memory writes by enforcing sequential consistency. Since the CPU cannot write to its own cache without first acquiring access rights to the bus, snoopy cache insures that writes occur in a clear sequential order.

Read bypass in the MOB — require that the CPU acquire I/O access rights to before reading a queued value from its write buffer. This informs the other CPUs of the write to the cache block even if the block has not yet been written.

Write overlapping — since writes invalidate other cache copies, all physical writes must occur in the correct order.

Read/write reordering — writes invalidate other cache copies, so that the rescheduled read is invalidated when the other CPU writes to the shared variable.

Slide 60 – 62 Sun T1 Multiprocessor

A different example of a multicore processor is the Sun T1. This CPU was introduced by Sun Microsystems in 2005, based on its UltraSPARC architecture and is optimized for thread level parallelism for OLTP (slides 28 – 29 in presentation 7). The T1 contained 8 processor cores, but was a single-issue scalar CPU (one ALU per pipeline) and performed in-order execution (no dynamic scheduling). The pipeline was intended for the integer instructions characteristic of OLTP — the 8 cores shared one FPU.

The pipeline consisted of 6 stages — the 5 stages of a DLX type CPU plus a stage for thread selection. Each core handled 4 threads, performing fine-grained multithreading, switching thread on each clock cycle and skipping over stalled threads.

As seen on slide 61, each of the 8 cores had private L1 cache (split data and instruction caches) and shared L2 cache among themselves.

The T1 processors underwent considerable evolution, even when Sun was acquired by Oracle to enhance its OLTP server business. The T4 (2011) doubled the number of threads per core, superpipelined the pipeline, introduced dynamic scheduling, doubled the clock speed, added a per core FPU and cryptographic processor, and added an L3 cache. Oracle's T4 server consisting of 4 T4 CPUs (256 threads) + 1 TB RAM was found to be 25% faster than a comparable HP 4 Xeon server on the OLTP benchmark TPC-H.

Presentation 9

Message Passing Architecture

מודל העברת הודעות הוא צורת העבודה במחשבים גדולים ומחשבי על. המודל הזה מתאים למקבילות ברמת הפתיל (TLP) וגם ברמת התהליך, ובעיקר בעיות גדולות שמבצעים חישובים רבים בין החלפת תוצאות בין פתילים. עם אפשרות לחלק בעיות למיליוני פתילים במקביל, העבודה בדרך כלל מחלקת קבוצה גדולה מאוד של נתונים לתת-קבוצות ומקדישה מעבד שיעבוד על הקבוצה. מניעת סיכונים (data hazards) נעשית ברמת ההודעות שמזהות את המקור והגיל של כל תוצאה.

Slide 2 – 4 **Message Passing Multiprocessor**

In **message passing**, the system is made up of N nodes (0 to N – 1), each node consisting of a CPU with its private address space A (0 to A –1). CPUs cannot generally access the memory of any other node. Threads running on the N nodes communicate among themselves by formulating, passing, and reading inter-CPU messages over the internal network. The messages combine data and memory synchronization information. Although multicore processors in a node share memory locations, the primary decomposition of work in parallel algorithms is performed by message passing.

Interprocessor communication involves transactions for sending and receiving messages defined in the **Message Passing Interface** (MPI), an API for message passing parallel programming. The API defines transaction primitives between the processor and the interprocessor network — the implementation of these primitives depends on the OS and the hardware. The interprocessor network is understood as a general computer network, whose performance is determined by the communication protocols and the efficiency of its message types, routing, network access, message segmentation/reassembly (SAR). For this reason, with adequate network capacity, message passing is scalable to large systems with no geographical restriction, and grows naturally into cluster computing. In many current implementations, MPI is mapped onto the TCP/IP protocol stack in the OS.

The basic message types perform: **send data**, **send request** (for data), **send response** (to a data or status request), and **send system command**. The messages are structured, including as **header** (source ID, destination ID, timestamp) and **content** (data / request / response). On the one hand, the requirements of message management are the main source of processing overhead. On the other hand, **sequential consistency** is enforced in message passing without shared memory or snooping overhead. The code expects specific content from a specific source, so that the message header provides a consistent write order.

Slide 5 **Message Passing Example — Scalar Product**

Slide 5 shows data decomposition in a message passing system:

1. Each node performs one multiplication,
2. P0 and P2 send their result to P1 and P3 that are expecting to receive them and perform partial sums,
3. P1 sends its new result to P3 that is expecting it and performs the final sum

P3 receives two messages in its code. There is no data hazard (such as out-of-order message arrivals) because the data is labeled by the source ID in the message header.

Slide 6 Blocking versus Non-Blocking Models

Every program must pair a send instruction in one thread with a receive instruction in another thread. In **synchronous** (blocking) message passing, threads synchronize on the send/receive pair. Send and receive instructions block in code — the processor cannot proceed to other instructions until the message transmission is completed. To create the synchronization barrier, the receiver waits for an RTS (request to send) signal and the sender waits for a CTS (clear to send) signal.

In **asynchronous** (non-blocking) message passing, the sender executes the send instruction and continues processing new code. The message is buffered at the sender until being sent through network, and it is buffered at the receiver until it is needed. The receiver accepts the message by executing a receive instruction. This method can be compared to the operation of TCP in internet connections.

Slide 7 – 11 MPI Messages

Slide 7 shows some MPI environment messages. The directives are sent by the application code to the OS to initialize, configure, and close the MPI environment.

Slide 8 shows some MPI point-to-point messages, that is, directives for sending messages from one node to one other node. These include the blocking and non-blocking forms of basic send and receive directives.

Slide 9 shows some MPI collective communication messages, that is, directives from one node to a collection of nodes. These include barrier synchronization, broadcast to a group of nodes, and the collective messages **scatter**, **gather**, and **reduce**. The scatter and gather operations are shown on slide 10. In scatter, one node distributes a list of data sets to a list of destination nodes (usually each node receives one data set). In gather, one node receives and assembles a list of data sets from a list of source nodes (usually each node sends one data set).

Slide 11 shows the reduce directive. It is similar to gather, except that the OS performs a reduction operation on the received data sets, such as adding the list of results assembled from the list of sources.

Slide 12 – 14 MPI "Hello World"

Slide 12 shows a simple program that sends the "Hello, there" string from the master thread (process 0) to thread 1. Thread 1 receives and displays the message. Additional details of this use of the send and receive directives are given on slide 13. Slide 14 shows the use of scatter and reduce to implement the scalar product in MPI.

Slide 15 – 17 Implementation Issues for Processor

The requirements for a processor directly supporting message passing include:

1. A physical send and receive port for message transmission
2. Send and receive message buffers
3. Hardware blocking on message access for synchronous messages

4. Interconnect support for message passing

If there is no direct support for message passing in the processor, then the OS and interconnect network must buffer messages for the processor and handle synchronization.

Slide 16 shows the plan for the Alpha 21364 CPU designed to support message passing. The processor package was designed around the Alpha 21264 RISC core with L1 cache. The Cbox in the center of the package is a communications controller connecting the main memory buffers (Z0 and Z1) designed for Rambus (an early version of fast DDR RAM), the System Data Buffers (for the message passing system), and an internal router that connects the package to similar CPU packages and to peripheral I/O. Two configurations for multiprocessor systems built from the Alpha 21364 are shown on slide 17. The connections between any two Alpha packages are made to the N, E, W, S connections to the internal router. In its time, the Alpha 21264 RISC core was the fastest processor on the SPEC list. The 21364 was never manufactured for commercial reasons.

Slide 18 – 19 Cluster Computing

Cluster computing refers to a large message passing system with distributed memory. It can exploit MPI scalability up to millions of nodes. The typical node is a standard workstation. The node-to-node scale may be up to a few meters (interconnected through a physical bus or non-blocking crossbar switch), a few hundred meters (interconnected through a LAN), or up to thousands of kilometers (a WAN). When the interconnection network is a LAN or WAN, the MPI_COMM_WORLD list includes MAC addresses or IP network addresses. Cluster computing enjoys the advantages of large message passing systems, as reviewed on slide 19.

Slide 20 MPI and Socket API

The socket API maps each endpoint (IP address and port) to a socket number. The socket number identifies a connection to a remote endpoint. At the client, the socket identifies a connection to server endpoint, and at the server, the socket identifies a connection from a client endpoint.

In systems using both the MPI and socket APIs, joining an MPI cluster can be implemented as a client/server system. The primary node acts as a client and opens a connection to each remote server. The secondary nodes run an MPI server listening on a well-known service port number. MPI maps the socket numbers to an MPI rank construct, providing the ID number for each node in MPI_COMM_WORLD. When a node performs an MPI send to a remote node it writes to the socket for the remote node, and when a node performs an MPI receive from a remote node it reads from the socket for the remote node.

Slide 21 – 23 Network Latency in Cluster Computing

It was seen (slide 34 in presentation 7) that Amdahl's law including communication overhead is given by

$$S = \frac{1}{(1 - F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)}$$

where

$$F_P = \frac{\text{Parallel instructions}}{\text{All instructions}}$$

$$F_{overhead} = \frac{CPI^{comm}}{CPI} = \frac{\text{Communication activity CC per instruction}}{\text{Computation activity CC per instruction}}$$

In order for cluster computing to work well, F_P and N must be large and $F_{overhead}$ must be small.

In a typical switch-based cluster computer built from standard PC workstations, the total message latency is about 150 μsec (70 μsec for processing in the communications protocol stack in each node and 10 μsec in the switching hardware). On a PC with a 1 GHz clock rate, the clock cycle is 1 nsec = 10^{-3} μsec. The total message latency is then

$$\text{Message latency} = \frac{150 \times 10^{-6} \text{ seconds per message}}{10^{-9} \text{ seconds per clock cycle}} = 1.5 \times 10^{-5} \text{ clock cycles per message}$$

about the time required to handle a page fault in the OS.

The overhead depends on the number of messages passed as the program runs. That is,

CPI^{comm} = clock cycles per message × messages per instruction

where clock cycles per message is the message latency of 1.5×10^5 clock cycles per message. With these parameters, a reasonable level of overhead (F_P < 10%) requires that each node run about 1.5 million instructions between messages.

For this reason, improving the latency of the interconnection network is a critical aspect of cluster computing. The Alpha 21364 (slides 15 – 17) was designed with specialized communication hardware, as have mainframe computers since the 1960s. Specialized hardware for offloading TCP and DMA for message passing is shown on slide 22.

Slide 23 shows a typical cluster configuration using conventional PCs. Each node is a shared memory multicore SMP system running tightly coupled code over OpenMP. The nodes are connected using TCP/IP over Gigabit Ethernet. In the resulting cluster of SMPs (CluMP) nodes exchange data for loosely coupled message passing code over MPI.

Slide 24 – 26 Thunder Cluster

Slide 24 shows the Thunder Cluster, a CluMP machine build for the US Department of Energy in 2004. The machine consists of 1024 Linux nodes, where each node is a quad-core Itanium 2 "Madison Tiger4" SMP. The Itanium is Intel's true 64-bit superscalar CPU, based on a RISC ISA with compiler-based explicit parallelism (slide 67 in presentation 3), virtual registers (slide 30 in presentation 3), and instruction predication (slide 19 in presentation 5). The Thunder reached a peak theoretical performance of 23 TFLOPS (1 TFLOP = 10^{12} floating point operations per second). Additional technical details for the node are found on slide 25. The interconnection network is shown on slide 26. The switching system provides a very high transmission user-to-user bandwidth of 900 MB/s in 5 layers.

Slide 27 – 34 Blue Gene/L

Blue Gene/L was designed as a massively parallel supercomputer for the US Department of Energy. With a peak performance of 596 TFLOPS it was the fastest supercomputer in the world in 2007 (only second fastest in 2008). The system contained a total of 64 Knodes (65,536 nodes) and a total of 32 TB of main memory.

The design of a computer this large creates unusual constraints and requirements, especially regarding equipment costs. Another significant factor is the cost of electrical power to operate the machine. Generally, higher performance requires more electrical power (faster hardware requires more computations per second, which means more binary transitions per second at the

circuit level, which means more electrical current per second). It is impractical to provide more than 20 kilowatts of electrical power to a rack of computer hardware. Together, this means that

Performance per rack = performance per Watt × Watts per rack

depends strongly on **performance per Watt**. In order to achieve high performance per Watt, the designers chose to use medium power processors. While high performance superscalar CPUs with dynamic scheduling deliver high performance (high instruction level parallelism) at high power consumption, Blue Gene/L was designed to provide high thread level parallelism at lower power consumption. The CPU is a highly integrated system on a chip (SOC) based on the IBM Power architecture. Technical details are presented on slides 31 – 33.

An upgrade of the Blue Gene/L design — Blue Gene/Q — was delivered to the US Department of Energy in 2012. It contains 98,304 (96 K) 16-core nodes with a total of 1.6 Petabytes ($1.6 \times 1024 \times 1024$ GB) of main memory.

Slide 35 – 38 Example of Cluster Application

A bad application for a cluster computer is multiplication of matrix **A** and vector **b** (shown on slide 35). This problem works well on a shared memory SMP but is dominated by communication overhead on a cluster.

The data decomposition is based on rows — the manager thread loops through the rows of matrix **A**, sending row_i to of the matrix to node i for execution. Each worker thread receives its row and multiplies it by a local copy of vector **b**. Each worker thread returns its result and the manager thread assembles the results as the vector **Ab**.

For an N component vector, the **computations** are N multiplications and N – 1 additions, for a total of 2N – 1 FLOPs (floating point operations).

The communications overhead is the transfer of the matrix, N^2 floats, and the return of the vector components, N floats, for a total of $N^2 + N$ floats. The communications overhead (estimated as floats sent and received — 2 operations — per flop performed) is about $2(N^2 + N)/2N$ or about N for large N. Under these conditions, the speedup goes to zero. This kind of data decomposition is not practical.

A better example is the numerical integration shown on slide 37. In this case, each node initializes its own data and must only communicate its result. Dividing the integration into N steps, the sequential version performs 2N + 1 FLOPs. In MPI, N can be made as large as the system will allow. Each node performs 6 FLOPs and sends 1 float as its result (causing 1 receive in the manager thread). In this case, with F_P about 1 and N large, the speedup is about 3.

Slide 38 shows applications in which clusters deliver good performance. Thread replicating servers (where a new thread is spawned for each client request) is a good application because there is very little communication between threads. Numerical integration and other forms of number crunching also work well. The applications run on US Department of Energy supercomputers are typically numerical solutions for 2nd order partial differential equations, such as weather forecasting and simulation of nuclear interactions. In such solutions, a large set of data is divided into regions. The computations within each region are independent, except when data on the boundaries between regions is exchanged. After several iterations, the set of independent regional solutions merge at their boundaries to form a solution over the entire set.

Slide 39 – 40 Learning Curve in Parallel Programming

Interesting research has been performed on the learning process for parallel programmers. For small-scale projects, typically "embarrassingly parallel" applications on shared memory SMP systems, skills can be acquired relatively quickly by using standard design rules.

Large-scale projects become more difficult, because debugging consistency is difficult to manage across a team of programmers. Coordination has been found to be a mixture of science and folklore.

Large-scale projects generally requires expertise in programming and the heuristics of the mathematical problem. Programmers become specialists in a specific field of high performance computing (HPC).

Slide 40 shows the results of graduate level (MSc) courses in which students were asked to implement a game. Generally, the parallel version with N = 8 CPUs compares well with the parallel version for N = 1 (showing the power of parallel programming). The N = 8 versions compare less well with the sequential versions of the program, showing the difficulty is identifying the best strategies for data and functional decomposition.

Presentation 10

Virtual Machines

וירטואליזציה מתייחסת לשימוש בטכנולוגיות מסוימות לבנות מערכת שהתנהגותה החיצונית מחקה מערכת אחרת. מאז 1964 מחשבי mainframe של חברת IBM מציגים סביבת עבודה וירטואלית למשתמשים — המשתמש רואה סביבת עבודה ליחיד (כמו ב- PC) שנוצרת בפעולות מתואמות במערכת גדולה רבת משתמשים. היום וירטואליזציה מבוססת תוכנה גם מהווה תחליף פשוט לפתרונות מבוססים בחומרה והיא מאפשרת השירותים של מערכת רצויה בלי התקנת המערכת האמיתית בהשקעת זמן ומשאבים רבים.

Slide 2 – 4 Virtual Machine (VM)

In a layered model of computation, software and hardware divided into logical layers, an abstraction which conforms to actual functioning to one degree or another. Layer n receives services from server layer n – 1 and provides services to client layer n + 1. The layers interact through a well-defined programming interface.

A **virtual layer** is the emulation in software of some hardware or software layer n. The name derives from the fact that it provides the virtues (characteristics) of layer n in a way that is transparent to layer n + 1. It provides service to layer n + 1 as expected from a real layer n. A virtual layer n can run at some layer m $\neq$ n in the real system.

Two common examples of virtual systems are shown on slide 3. When a **web browser** exchanges data with a web server, the applications perform virtual communication with one another, as if the browser and server were directly connected. The real system that implements this virtual communication consists of the OS in each system, the computing hardware, and the network infrastructure.

Similarly, in **cloud computing**, a customer enters a service level agreement (SLA) with a provider of computing services. The SLA specifies the virtual computing system that the customer's users will see — hardware, software configuration, and performance. The real system assembled by the provider to implement the virtual configuration is invisible to the customer and the user, as long as it meets the requirements of the SLA.

A basic computing system consists of hardware, an OS that manages the hardware and provides user access to hardware resources, and application programs running above the OS. Slide 4 shows three types of virtualization.

In a **Process Virtual Machine**, the VM provides application interpretation above OS — it allows applications designed to interact with the VM to obtain standard services from the OS. These applications can then run above any OS for which an appropriate VM has been written.

In a **Hosted Virtual Machine**, a virtual machine monitor (VMM) runs between the primary OS and a guest OS. The VMM provides the guest OS with software emulation of a real hardware system, so that applications written for the guest OS may run as usual.

In a **System Virtual Machine**, a VMM runs directly above the physical hardware to emulate a system-level hardware environment. The VMM permits one or more OSs to run above the virtual hardware.

Slide 5 Process VM Example — Java

A well-known example of a Process Virtual Machine is Java. The Java VM runs above a standard OS, providing an implementation dependent interface to system hardware. I/O operations are performed by calls to the OS. High level Java programs are compiled to bytecode which usually runs (interpreted) in the Java VM to machine code for the real CPU. In IBM mainframes and other certain other processors Java bytecode can be run as the native machine language, permitting execution without interpretation in the VM.

Slide 6 Hosted VM Example — Guest OS Over OS

In 32-bit versions of Windows, the program cmd.exe provides a DOS command line interface in a window. In order to create this virtual DOS environment, Windows allocates 1 MB of virtual memory space and copies the DOS kernel into low memory. Applications running in this 16-bit DOS window issue system calls that are handled by the guest DOS kernel, but the DOS accesses to hardware are trapped and served by the Windows host OS, which returns system responses to DOS. Concurrent DOS windows can be opened by multiple allocations of a 1 MB virtual memory space. The DEBUG application running on a native (real) DOS system has full access to the all system resources, including hardware registers. In a virtual DOS machine, DEBUG sees the 1 MB memory space allocated by Windows, which emulates register values for DOS.

In a similar way, host virtualization programs such as Parallels (Apple), VirtualBox (Oracle), VMware (EMC), DOSBox (GNU) permit an OS to run in a virtual environment above a host OS.

Slide 7 – 8 Virtual Machine in IBM z/990 Mainframe

The hardware layer of a mainframe computer consists of multiple CPUs, the I/O system, and an internal communication network. A VMM called a hypervisor runs above the hardware, providing an operator console for partitioning/configuring CPUs and I/O, and hardware emulation as an abstraction to the OS layer. The OS for a mainframe expects to interact with the hypervisor (not real hardware). The hypervisor configures the system into logical partitions (LPAR), each of which runs a separate instance of an OS. With multiple LPARs, the system can run z/OS, MVS, VM, Unix, Linux, Windows, and other OS instances in parallel. The user sees a single-user interface provided by one OS.

The mainframe VM serves the administrator as a system management tool that isolates user environments on single hardware platform. It permits multiple OSs or multiple copies of a single OS to run independently, while maintaining higher security. The VM also centralizes and simplifies resource management for hardware redundancy, high availability, recovery management, hardware pooling (dynamic assembly of clusters for efficient mapping of applications to hardware and load balancing).

Slide 9 – 10 Virtualization for Server Systems

As PC-type workstations evolved into application servers around 2000, it was typical to run one application per physical server that was specified to handle the worst-case load. As server farms grew, this led to a large number of underutilized servers. Competition from modern mainframes (with powerful RISC CPUs and a Posix-compatible OS) grew, presenting the advantages of dynamic load balancing, centralized power, cooling, monitoring, and backup, and very high SAR

(scalability, availability, reliability). In general, mainframes can offer lower cost per served client than server farm.

Manufacturers of traditional x86 server systems responded by offering virtualization systems to partition hardware resources for running independent applications. In particular, Intel introduces its Virtualization Technology (VT) to support its IA-32 and IA-64 ISAs, and its I/O chipset. Slide 10 shows the virtual partition of an HP Unix-oriented server system. On the left side, the unpartitioned system includes 4 CPUs and a system bus that connects main memory and a pair of independent I/O systems. This system boots UX (HP's Unix version). On the right, the system boots a virtual partition monitor (VMM) that performs a virtual partition of the hardware, and then boots independent UX OSs on each partition.

Slide 11 – 15 VM Interventions

In a mainframe or other system VM, the hypervisor (VMM) is the lowest layer above physical hardware (the host). It creates virtual machine (VM) environments for OSs and allocates physical host resources to virtual resources. Processor intensive applications run with low VMM overhead — most instructions run directly on hardware and make infrequent OS calls. But I/O intensive applications run with high overhead, making frequent OS calls for emulated I/O services. Some applications are I/O-limited — the program throughput inherently limited by the usual causes of I/O latency and so emulation adds relatively little overhead.

In a hosted VM system, the VMM intervenes in a wider variety of situations. To insure that the OS recognizes the hardware and can install required drivers, the hardware abstraction must replicate standard and familiar hardware for the guest environment and present well-defined emulated interface to the OS. The VMM must then intervene to convert driver outputs intended for the emulated hardware to the requirements of the real hardware. The VMM must also isolate guests from one another and protect itself from guest OS and application software. Guest software must not be allowed to change the allocation of physical (real) resources. Since the VMM runs in kernel mode, the applications and the guest OSs run in user mode — this can create complications for the guest OS. Although virtualization primitives are built into the mainframe ISA, the VMM on a hosted VM must trap any OS or application access to the hardware abstraction layer (HAL).

Virtualization-unaware guest OS is written to run above the physical hardware, and expects full and direct control of real hardware. Correct functioning requires extensive intervention and the VMM must remap (bluff) pointers intended for real hardware.

A virtualization-aware guest OS is written to run above a VMM, and expects to interact with the virtual host. It does not expect full or direct control of physical hardware. The OS code interfaces with the VMM code with no need to remap pointers.

Slide 14 provides a general description of the interventions required by the VMM for standard hardware operations. As shown on slide 15, some systems require full emulation of every guest activity — a common example is running a game designed for one hardware system and machine language on a different system with a different machine language. But most VMMs are intended to run an OS above a different OS written for the same hardware — in this case, most machine instructions can be sent directly to the CPU without VMM emulation.

Slide 16 – 17 Virtualization Difficulties for x86

The IA-32 ISA was designed to provide hardware support to the OS, including, memory segmentation, virtual memory and paging, task management, and interrupt management. IA-32 also defines privilege-based protection for segmentation, paging, interrupts, granting kernel mode rights (ring 0) to the OS and user mode rights (ring 3) to applications.

Initial attempts at virtualization on x86 CPUs attempted a workaround — treating the OS like user application running in user mode above the VMM running in kernel mode. This can create a kludge on these systems. An IA-32 OS expects to have the highest privilege on the system and can easily discover that they run at lower privilege. They also expect to see the full virtual memory space and have direct access to CPU-controlled system tables. Since the VMM must reserve address and table space for itself, the OS suffers memory and table compression.

In its Virtualization Technology (VT) Intel created a new privilege level below kernel mode, called the VMX root. Kernel mode and user mode are called non- VMX root modes. The VMM runs at the VMX root privilege level, so that the OS can run in standard ring 0 kernel mode, and applications run in standard ring 3 user mode. System calls requiring access to I/O or real hardware are trapped by the VMM, which switches to VMX root mode to handle these requests.

Slide 18 – 20 VirtualBox

VirtualBox is an open source hosted VMM by Oracle (acquired when it purchased Sun Microsystems). It runs on Intel and AMD x86 hardware above Windows, Linux, Mac OS X (Intel), and Solaris, providing a VM for a guest OS. It uses hardware virtualization support if available (but is not required). The host OS schedules the VM by granting a timeslice — the guest OS schedules sub-processes in the VM.

VirtualBox is designed as a client/server system. The front-end (client) is the hypervisor that runs above the host OS. The back-end (server) runs as a ring 0 driver in the host OS. In the language of the VirtualBox documentation it copes with the "gory details of x86 architecture," to allocate physical memory for the VM (guest OS), save and restore guest CPU context during a host interrupt, manage IA-32 registers and descriptor tables. It makes no interventions in guest OS process management.

Some code runs native (not emulated) on the CPU: host OS code at ring 0, host applications at ring 3, and "safe" guest application code at ring 3 (non-system activities and code that makes system calls to the guest OS). The code that runs emulated at ring 3 includes guest application code that would disable interrupts or perform prohibited accesses. Naturally, instructions produced by the VirtualBox back-end driver (to replace native I/O operation) run at ring 3. In order to minimize use of VT mode transitions, VirtualBox runs most guest OS ring 0 code at privilege ring 1 (less privileged kernel mode).

Slide 21 – 28 Xen

Xen is an open source system VMM that runs on Intel and AMD x86 hardware directly above the hardware, providing VMs with parallel guest OSs (Linux, Solaris, Windows). Hardware virtualization support is required for a Windows guest OS. For a Linux/Unix guest OS, Xen provides paravirtualization — a modified OS kernel that operates directly with Xen, and a virtual hardware layer very similar to the native hardware.

The Xen hypervisor operates directly above the hardware and boots the system on start-up. The VMM creates one Domain 0 and one or more copies of Domain U. Domain 0 runs XenLinux and Domain Management and Control (DMC). Domain U provides a VM running the guest OS. The hypervisor operates directly above hardware in ring 0 to perform CPU scheduling for the virtual machines, memory partitioning, and provide hardware abstraction to virtual machines. It has no awareness of issues handled by Domain 0: networking, external storage devices, video, or common I/O.

Domain 0 runs XenLinux, a modified Linux kernel running in a unique VM over the hypervisor. It has direct privileged access rights to physical I/O resources and provides I/O virtualization to the Domain U guest VMs. It also provides generic I/O drivers. The Network Backend Driver manages local networking hardware and processes all VM networking requests from Domain U guests. The Block Backend Driver manages local storage disk and processes all read/write data requests from Domain U guests.

A Domain U running Unix or Linux performs paravirtualization, running a modified kernel above a hardware abstraction very similar in structure to the real hardware. This OS expects no direct access to host hardware and knows that it shares host hardware with other VMs. The guest drivers provide I/O access through backend drivers in Domain 0 (the PV Network Driver and PV Block Driver).

A Domain U running a non-virtualization-aware OS is called a Domain U HVM (host VM) and runs fully virtualized machines as VMX non-root operation with Intel VT. The OS expects no hardware virtualization and no sharing with other VMs. It also expects full hardware access for boot. Xen virtual firmware runs as a VMX root operation to simulate the BIOS expected by OS. Domain 0 runs a Qemu-dm daemon for each HVM guest OS to support networking and disk access requests.

Slides 27 – 28 summarize research on the XEN performance. Slide 27 shows that using Intel VT, a Xen system generally performs very well compared to a native OS running on the same hardware. However slide 28 shows another test that revealed an interesting hidden limitation. This experiment shows that the test software experiences a communications bottleneck that limits performance. The native system (without virtualization) experienced an improvement in performance when the number of network cards in the hardware was increased from 1 to 2. The Domain 0 OS in the Xen system experienced a similar performance improvement, but the performance of the Domain U OS did not change (the native system significantly outperformed the virtual system with adequate hardware for the applications). The native system experienced another performance boost when the number of network cards in the hardware was increased to 3, but there was no similar increase for the virtual system. Generally, virtualization can provide many advantages, but does not achieve the performance expected from native systems.

Slide 29 – 33 Cloud Computing

It has become common to use the word **cloud** as a synonym for **network**. But cloud computing refers to a specific mode of operation, essentially a business model rather than a computing model. It is a form of outsourcing in which a customer contracts to receive computing services from a service provider. A service level agreement (SLA) guarantees a level of service to the customer to be enjoyed by the customer's users (usually, the customer's customers). The provider handles the complex details of OAM (operations, administration, maintenance). Cloud computing brings a number of business advantages in appropriate cases, especially the economies of scale possible for large provider. Also, contracting for computing services cuts labor and capital costs from the customer's balance sheet, which can make investors happy. On

the other hand, the contract requires monthly payment (like rent) that can become a major burden on a company struggling for income and cash flow.

Cloud computing is based on standard technologies, not a fundamentally different computing technology. The service is provided using conventional resources — hardware, software, and network — from a menu of services the provider offers customers. The unique technological issues in cloud computing center on service reliability (the provider is committed to the SLA), optimization of the provider-side resource configuration, and optimization of the customer-side resource configuration. In this case, optimization refers both to performance and cost.

Cloud computing is characterized by a service hierarchy:

In **infrastructure as a service** (IaaS), the customer sees a virtual hardware environment, some combination of real hardware and a system virtual machine. The customer installs an OS and application software, and runs jobs.

In **platform as a service (PaaS)**, the customer sees a virtual OS environment, installs application software, and runs jobs.

In **software as a service (SaaS)**, the customer sees a virtual application software environment, and runs jobs. These applications run on a private copy of the OS or are "sandboxed" on shared OS. A sandbox is a private execution environment per application instance that cannot access the resources of other running programs. Google apps is one familiar example of SaaS.

A hybrid category is **storage as a service (STaaS)**, where the user sees a virtual storage device mounted on a local computer. Examples, include Dropbox, Copy, Box, iCloud and others.

The popularity of cloud computing is an example of changing fashions in the computing industry. Slide 31 shows a timeline over the past 60 years, showing the trend away from centralized computing systems to distributed systems, flowed by the return to centralization.

Cost is a major factor in cloud computing. For the provider, economies of scale lower the cost of performing any given compute job. Cost is often the primary factor in competitiveness. For the customer, cloud computing exchanges capital and OAM costs into service costs. If the customer's income does not meet the cost of the cloud service, this exchanges start-up costs for operating debt.

Reliability is a major issue for the provider. The provider's planning must include redundant infrastructure for continuity and disaster recovery, centralized management of OAM, security, performance, for administrative control and cost management. Choices include virtualization, which can provide multiple OSs on one physical server or multitenancy, which providea multiple sandboxed application instances on one OS.

Agility is another advantage of cloud computing. The user or provider can reconfigure the service and infrastructure as needed. This permits smooth growth, effective load balancing, and time-zone serving (where one system is redirected toward a part of the world where demand is greatest, shifting as day turns to night).

In a public cloud, the service provider acts as a public utility, selling (renting) computing service. The initial providers of public cloud service were leveraging their large existing computing infrastructure — Amazon, Microsoft, Google, IBM.

In a private cloud, the cloud infrastructure is managed (internally or outsourced) for a private organization. The goal is to isolate service developers from certain implementation issues. Programmers and lower-level IT personnel see a standard virtual development platform that remains stable over many years, while the real hardware undergoes periodic reorganization. Private cloud computing is economically justified for large organizations offering technology-based services and introducing new services frequently.

Presentation 11

Energy Efficient Architecture

מערכות "ירוקות" או מערכת חוסכות חשמל מתוכננות לצמצם את צריכת האנרגיה שלהן בשימוש השגרתי. בתחילת המאה המניע העיקרי לחסכון בחשמל במחשבים היה שמירה על הסוללה במחשבים ניידים. בשנים שעברו מאז הנושא תופס יותר חשיבות ככל שמבינים יותר את המשמעות של צריכת חשמל לא מבוקרת במערכות מחשוב גדולות. פיתוחים חדשים בפיסיקה והכימיה של סוללות האריכו את הזמן היעיל של מחשב נייד לפחות פי חמש ב- 15 השנים האחרונות. הגישה בתכנון חומרת מחשוב נשארת כמו תמיד — כשלא משתמשים ברכיב מסוים, כבו אותו.

Slide 2 **Energy Efficiency**

The initial motivation for energy efficient devices began with the development of mobile computing. A large market potential was identified in the 1990s with the standardization of WiFi: wireless networking would make laptop use more attractive and energy efficient laptops would make WiFi systems more attractive. Both systems were limited by battery life-time.

A subsequent insight is that electricity consumption is a large factor in computing cost. Even without the cost of cooling, the electric bill for a large server farm can be estimated as

(500 W / PC server) × 1000 servers = 500 kW

500 kW × 720 hours / month = 360,000 kW / month

292,000 kW-Hr / month × 0.54 shekels / kW-Hr = 157,680 ח"ש / month

Power consumption also limits the performance of computing systems. Since

Performance / rack = (performance / kW) × (kW / rack)

any improvement to performance / kW improves the performance of large systems. As mainframes have become more popular, the relative energy efficiency they provide through centralization has become a highly competitive factor.

Slide 3 – 4 **Power Consumption**

Power is consumed in digital circuits during $0 \rightarrow 1$ and $1 \rightarrow 0$ state transitions. Since power is defined as energy consumed per second, it is shown in slide 3 that power consumption for a given program equals the energy per task divided by the program run time. Therefore it is possible to reduce the power consumption by lowering the clock frequency (making the program run more slowly), or by reducing the energy transition (by raising the energy efficiency of each transistor or using fewer transistors). These relationships are summarized on slide 4 as

Energy per task = power × run time

Performance = instructions per clock cycle × clock frequency

Slide 5 – 6 **System Issues**

Research into power consumption in laptop systems found that the CPU only consumes about 10% of the power required by the system. The peripherals (LCD, hard disk, memory system, networking components, I/O system) consume about 90% of the power. Therefore, making the CPU run more slowly to save power leaves the peripherals running for a longer time, and

producing very little energy savings. This realization led to a major change in strategy for laptops — increase the CPU speed (even with a moderate increase in power consumption) in order to finish the computing task as quickly as possible, and then put the CPU and the peripherals into sleep mode.

The goal became to classify changes to processor architecture according to their effect on consumed power and run time. The most desirable changes would decrease power consumption and run time. A small increase in power consumption should be considered if it decreases run time, and a small increase in run time should be considered if it decreases consumed power. Increases in both categories must be avoided.

Slide 7 – 12 Choosing Architectural Changes

Slides 7 – 12 show the method for classifying architectural changes. The CPU is assumed to have an initial state defined by its default power consumption P_0 and its default performance S_0. Each parameter will shift under an architectural change by an amount ΔP and ΔS. The criteria for accepting and rejecting changes are expressed in terms of ΔP and ΔS.

Slide 8 shows certain relationships among power, energy, performance, system voltage, and clock frequency familiar to electrical engineers. They show that lowering the system voltage can reduce CPU power consumption, but this also decreases performance.

Slides 9 – 11 show how these relations can be combined to define a criterion for architectural changes. A relative system voltage produces relative changes in performance, power consumption and total energy consumption according to **voltage scaling**:

$$\frac{\Delta P}{P_0} = 3\frac{\Delta V}{V_0} \qquad \frac{\Delta E}{E_0} = 2\frac{\Delta V}{V_0} \qquad \frac{\Delta S}{S_0} = \frac{\Delta V}{V_0}$$

For example, a 1% drop in voltage will produce a 3% drop in power consumption, a 2% drop in total energy consumption, and a 1% drop in performance. The criterion for architectural changes is that it must be better than voltage scaling. Thus, any change that lowers power by 3% but also lowers performance by more than 1% is not worthwhile — it would be preferable to just lower the voltage. But a change that lowers power by 3% and lowers performance by less than 1% is worthwhile — it cannot be achieved by lowering the voltage.

These criteria are graphed on slide 12.

Slide 13 – 20 Pentium M

The Pentium M was designed at the Intel design center in Haifa as a mobile version of the Pentium 4. Design requirements set by Intel were:

1. Satisfy IA-32 Instruction Set Architecture (ISA) for integer, float, MMX and SSE/SSE2 vector extensions.
2. Implement the microarchitecture of the Pentium 4
3. Save power — maximize battery life in laptop while providing "close-to-desktop" performance (comparable to a Pentium 4 of the same generation)

The design strategy was to introduce architectural changes that meet the voltage scaling criterion. To the surprise of many, the result was a processor that out-performed the Pentium 4 under energy optimization and at maximum performance.

The main enhancements in the Pentium M are:

1. Improvements to branch prediction leads to fewer mispredicted speculative loads, a decrease in the number of instructions per task, and a decrease in run time. This combination increases performance and lowers power.
2. Decreasing the number of micro-ops per instruction by micro-op fusion lowers performance slightly while lowering total energy consumption.
3. Decreasing the number of transistor switches per micro-op lowers performance slightly while lowering total energy consumption.
4. Decreasing energy per transistor switch by building higher quality transistors raises performance and lowers power.

The improvement in branch prediction was achieved with three new features. The first is the loop stream detector (trace cache, discussed in presentation 5). The second is the indirect branch predictor (slide 32 in presentation 4). The third is called a **jump detector**. On the first encounter with a branch instruction, it misses in the branch predictor. It is predicted not taken if it jumps forward (usually and if-conditional) but predicted taken if it jumps backward (generally a loop). The Pentium-M misprediction rate is 20% lower than the rate for the Pentium 4.

The Pentium 4 is a superscalar with dynamic scheduling. IA-32 instructions are decoded to RISC type micro-ops that execute out-of-order for high throughput. But the required resources (decoders, instruction pool registers, registration station) grow with the number of micro-ops, so that the independent processing of micro-ops consumes more power. The Pentium M decodes certain IA-32 instructions to fewer micro-ops (fused micro-ops). Multiple inherently sequential micro-ops are stored as a single fused micro-op with independent sections. The dispatch unit acts on one independent section of a micro-op, executed once for each section. This reduces the number of micro-ops by about 10%.

Similarly, in the Pentium 4 stack operations are decoded as a read or write micro-op on the stack and a separate update of the stack pointer. In the Pentium M, the various changes to the stack pointer are accumulated in a dedicated counter, and the stack pointer is only updated on retirement of a stack operation.

Slides 19 – 20 show comparisons of the Pentium 4 and Pentium M under energy efficient operation (speed close to Pentium 4 and much higher efficiency) and under full speed operation (higher speed and efficiency).

Slide 21 Intel Core Duo

The Intel Core Duo (Intel's second generation dual core CPU) was designed as a chip containing two enhanced Pentium-M cores. The design requirements set by Intel were

1. Similar or better performance than Pentium M on a single thread code
2. Improved performance for multithreaded and multi-processs code
3. Average power consumption of dual core same as Pentium M
4. Meets thermal requirements of target systems

The Core Duo contains independent thermal control per core.

Slide 22 – 25 ACPI

Even with energy efficient processors, the most effective way to save power is to deactivate system components when they are not required. The **Advanced Configuration and Power Interface** (ACPI) is an industry standard for the OS interface to processor configuration, developed by HP, Intel, Microsoft, Phoenix, Toshiba. ACPI defines operating states for the global system and devices.

The global (system) states are:

G0 (working), G1 (sleeping), G2 (software off), G3 (mechanical off)

and the device states are:

D0 (on), D1/D2 (device dependent) , D3 (off)

Specific CPU power states are

C0 (active), C1 (halt), C2 (stop clock), C3(sleep), C4 (deep sleep), ...

The OS controls the CPU power state to save power. When the scheduler detects an empty task queue, it transitions the processor into an idle state with a BIOS instruction sequence. The processor remains idle until it receives an interrupt that indicates a new task to be executed. The OS returns the processor to the C0 state. The use of sleep states for the Core Duo is shown in slide 23.

The OS is given range of sleep options. One possible idle management transition diagram is shown in slide 24. The definitions of the sleep states for the Core Duo is given in slide 25.