Overview

הקורס **ארכיטקטורות מחשבים** עוסק בשיקולים וחשיבה מאחורי התכנון של מעבדים למחשבים, טאבלטים וסמארטפונים. ההרצאות משלבות תיאוריה עם דוגמאות מעשיות במטרה להבין את הפלטפורמות העדכניות.

הקורס מחולק לעשרה פרקים:

1. **מבוא לחקר ביצועים**: נציג מושגי יסוד בביצועי מחשבים וגורמי השפעה בחומרה ותוכנה
2. **רענון ארגון המחשב**: נציג מבנה von Neumann והמרכיבים שלו — CPU, ALU, זיכרון, פקודות, קלט/פלט ובקרה. גם נדבר על תכנון קבוצת פקודות :סוגי פקודה ,מבני פקודות , שיטות מיעון, והגישה הקלאסית לתכנון מחשבים מורכבים.
3. **תורת הביצועים המתמטית**: מדידה והשוואה של ביצועים, benchmarks, חוק Amdahl, משוואת ה- CPU וכלים שונים לקביעת הביצועים של מעבד.
4. **תכנון קבוצת פקודות מסוג RISC**. בפרק הזה נלמד את השיקולים לתכנון מעבדים מודרניים ושיפור הביצועים ביחס ממחשבים הקלאסיים מסוג CISC.
5. **מודל ה- DLX**: נלמד גרסה בסיסית למעבד מסוג RISC ושיטות המימוש שלו.
6. **שיפור מודל ה- DLX**: נציג כיצד ארכיטקטים מנתחים ביצועי מעבד בזמן ריצה ומכניסים שיפורים כמן חפיפה (pipelining) לתכנון קיים.
7. **חשבון מחשב**: הצגת נתונים ,חיבור וחיסור ,כפל ,חילוק ,חשבון במספרים צפים
8. **היראכיה של זיכרון**: אוגרים, מטמון (cache), אחסון לטווח ארוך. נציג שיטות ניהול זכרון והשיקולים בתכנון מערכות זכרון קיימים. גם נלמד על מערכות קלט/פלט.
9. **מערכות מודרניות**: superscalar, dynamic scheduling, superpipeline, multicore, parallel programming ועוד שיטות עדכניות במעבדים של היום.
10. **RISC בשוק**: סקירה על מעבדי RISC מסחריים כמו MIPS (הבסיס למודל DLX) ומעבד ARM שקיים ביותר מ- 95% מהטלפונים והטבלטים.

מי שמסיים את הקורס בהצלחה יהיה מסוגל להסביר את הפיתוח של מחשבים והארכיטקטורה שלהם משנות ה- 1970 עד לטכנולוגיות האחרונות. הקורס יקל על לימוד טכנולוגיות מתפתחות חדשות, ולקבל החלטות נבונות בבחירת מוצרים. הקורס גם מספק שפה משותפת לאנשי תוכנה ומהנדסי חומרה שתקל על שיתוף פעולה במחקר ופתוח.

הקורס בנוי על שילוב של מצגות וסיכומים כתובים בטקסט מלא. מטרת המצגות היא להציג איורים, תוכניות, והגדרה בסיסית של מושגים בקיצור נמרץ. השקפים לא מהווים תחליף לקריאה בסיכומים וביצוע תרגילים. <u>לימוד מהשקפים לבדם לא מומלץ</u>.

<u>המלצות ללימוד עצמי</u>

כפי שניתן לראות באתר הקורס, מבנה הלימוד מוגדר בפריסה שבועית של הנושאים והתרגילים. קורס מתוקשב מתבסס על הציפייה שסטודנטים בשלב מתקדם מכירים את שיטות הלימוד המוצלחות להם ויודעים להפעיל את המשמעת העצמית הדרושה ללמוד באופן שוטף ולהגיש תרגילים בהתאם למועדי ההגשה. אבל מהניסיון, הפיתוי לדחות לימוד קורס מתוקשב חזק (עד כדי להשאיר אותו לתקופת המבחנים). הכניעה לפיתוי זה לא מביאה לתוצאות מספקות. ממבט ראשון, נדמה שיש יותר מדי נושאים בקורס ללמוד בסמסטר אחד. אבל כשלומדים באופן עקבי וסופגים לאט, רואים שבעצם מדובר במספר לא גדול של מושגים קשורים. כל שיטה מהווה שיפור מדורג על שיטה הקודמת ברצף והנושאים מתגלגלים אחד לשני באופן טבעי.

בכל נושא, מומלץ לעבור על המצגות עם הסיכומים (הרצאות מודפסות) פתוחים ועין על התרגיל שמיועד להשלים את נושא על ידי תרגול. חלק מהנושאים קל לקלוט מהמצגת, ובחלק אחר חשוב לקרוא את הטקסט הנלווה כדי להבין את ההקשר. בכל מקרה שמשהו עדיין לא ברור תפנה/י אלי להסבר נוסף בדואר אלקטרוני — אשתדל לענות לכל הודעה תוך 24 שעות.

Chapter 1

Introduction to Computer Architecture

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

Slides 2 – 4 What is Computer Architecture

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

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

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

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

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

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

Instruction Set Architecture (ISA)

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

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

Processor design proceeds in 4 steps:

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

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

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

Important **market factors** are:

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

Slides 5 — 6 Computing Platform by Application

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

Workstation applications

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

Mobile applications

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

Online Transaction Processing (OLTP)

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

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

Supercomputer applications

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

Mainframe

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

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

Cloud computing

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

Slide 8 **Basic Definitions**

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

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

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

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

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

Slide 9 Run Time and Clock Cycles

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

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

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

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

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

Slide 10 Speedup and Clock Rate

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

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

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

Slide 11 Factors Affecting Run Time

CPU hardware

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

Memory

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

Internal communication and I/O

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

Operating system efficiency

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

Compiler

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

Special hardware

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

Application code

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

Slides 13 — 16 CPU Hardware and Memory Examples

The most obvious example today of the influence of the CPU hardware on system performance is the number of **CPU cores** in the main processor. Most personal computers and telephones now include a CPU with 2 to 8 cores.

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

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

Slide 14 shows an example of an operation for a **vector processor**. In this example, each of two 64-bit registers holds eight byte-long operands — byte 0 to byte 7. The **Parallel Addition** adds byte 0 from the source (SRC) register to byte 0 from the destination (DEST) register and stores the result in byte 0 of DEST. At the same time it adds byte 1 (SRC) to byte 1 (DEST) and so on. There is no carry from one byte to another — these are eight separate byte-oriented additions. Some vector processors can perform parallel operations on eight 64-bit operands at once.

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

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

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

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

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

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

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

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

Slides 17 — 22 Compiler Efficiency Example

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

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

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

Slide 19 shows the assembly program along with the instruction type and required CC. The first and last instructions (light green) run only once at the start and end of the program. Four instructions (dark green) form the loop defined in the C program — compare, increment, branch. Three instructions (light blue) are the ALU action in each loop iteration.

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

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

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

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

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

Slide 24 **Benchmarks**

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

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

Slides 25 – 31 **SPEC Benchmark**

SPEC

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

SPEC sells testing suites for various types of performance, including CPU integer (int) operations, CPU floating point (fp) operations, system as a file server, web server, mail server, graphics processor. The suites are updated every few years to reflect realistic conditions. The current versions for CPUint and CPUfp were released in 2006.

How SPEC Works (slide 26)

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

The programs included in SPEC 2006 are listed on **slide 27**. The CINT suite includes programs with integer-intensive calculations and the CFP suite includes programs with float-intensive calculations. As described on **slide 28**, the reference machine is a Sun Ultra Enterprise 2, a computer that was considered very fast when it was introduced in 1997. In order to prepare the machine to run the tests of SPEC 2006 it was necessary to add physical memory — the original machine could not run the new programs. The run times for the various programs are shown on **slide 29**. The run times are very long — up to 22,000 seconds (over 6 hours) — so that runtime can be measured precisely on a standard clock.

Typical SPEC Report

The first part of the report lists $\mathbf{T_i^{ref}}$, $\mathbf{T_i^{test}}$, 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. **Slide 30** shows a report for the Sun SPARC Enterprise M8000, an updated version of the reference computer. The report shows that the runtimes are, on average, about 10% of the runtimes on the reference machine — the **SPECint score** is about 10. Notice that one program runs 23 times as fast while another runs less than 4 times as fast. These differences represent the structural improvements in the newer system that are advantageous for some programs but not for others.

Information regarding the system configuration is reported following the test results. **Slide 31** shows technical details for the system under test: a SPARC64 processor running at 2.28 GHz clock rate, 32 cores on 16 chips with 2 threads per core, 128 MB of instruction L1 cache (I), 128 MB of L1 data cache (D), 5 MB of unified L2 cache (cache is covered in presentation 8), 64 GB of main memory (RAM) and a 73 GB hard drive. The OS is Solaris (a UNIX version). To get a sense of this computer manufactured in 2007, consider that very few personal workstations have 64 GB of RAM (4 to 8 GB is typical today), but no disk as small as 73 GB is available today (typical is 500 to 2000 GB today). This machine had as much memory as was practically available.

Slides 32 – 33 Representative Results

The tables on slides 32 and 33 show some representative results for CPUs on Cint (SPECint) and Cfp (SPECfp). The fastest CPU for integer instructions is the **Intel Core i7-5960X**, a processor designed for powerful servers, workstations and powerful gaming machines. The tested machine contains 1 processor chip with 8 cores per chip, for a total of 8 running software threads. This chip also permits automatic parallelization of software. The SPEC software sources are written sequentially with no parallel algorithms — the compiler can automatically recognize opportunities to run parallel threads on multiple cores. The base score on Cint is 79.7

The **Intel Core i7-6700K** processor has a clock speed similar to the **Core i7-5960X** and with a score of 77.4 is almost as fast, running only 4 threads. The comparison suggests that automatic parallelization may not be effective for more than a few threads. This limitation is seen again in the **Xeon E5-2699**, which achieves a score of Cint = 74.0 (4% lower) while running 44 threads (11 times higher) at 2.2 GHz (half the speed). If performance scaled linearly with the number of threads, this processor would have a Cint score in the neighborhood of 420. Similarly, the **Core**

2 Duo E6850 running with automatic parallelization is seen to be only 5% faster than the same chip with parallelization disabled.

Because the multithreading is performed automatically by the compiler, only the most obvious parallelization opportunities are recognized. Even so, parallelization does not guarantee improved performance — a machine with dual **Xeon E5-2699** processors (88 threads instead of 44) achieved a slightly lower score than the single processor machine. Optimized utilization of multicore systems requires the intervention of a skilled programmer knowledgeable in parallel algorithms.

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

Slide 34 **Actual Sources of Performance Improvement**

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

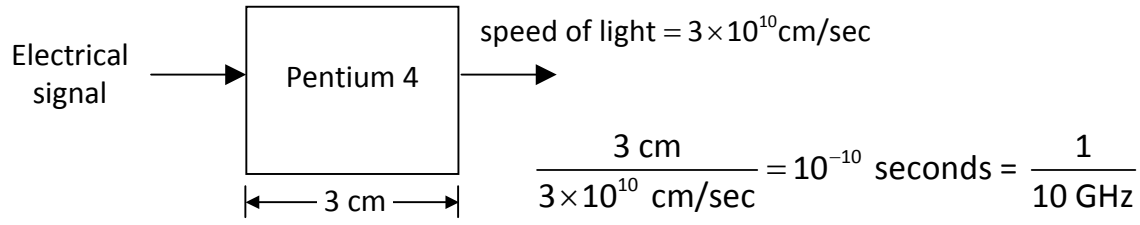
$$S = \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{\text{clock rate}^{new}}{\text{clock rate}^{old}} = \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{4\text{ GHz}}{4\text{ MHz}}$$

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

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

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

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



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

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

Chapter 2

Instruction Set Architecture (ISA)

מטרת הפרק הזה היא הגדרת **ארכיטקטורה של קבוצת הפקודות** והבנת השיקולים שהדריכו תכנון מחשבים בין 1950 – 1990 ועדיין תקפים לגבי המחשב האישי. נסקור את האפשרויות שניתן לבחור מהן בהרכבת ISA והשיטות לממש אותן ב- microarchitecture.

Slide 2 **Overview of Chapter**

What is a processor?

- Von Neumann architecture
- Stages in the design of a processor

Instruction set

- Structure of instructions
- Operands and data
- Data storage and memory types
- Operations on data

Considerations in design of an instruction set

- Complex Instruction Set Computers (CISC)

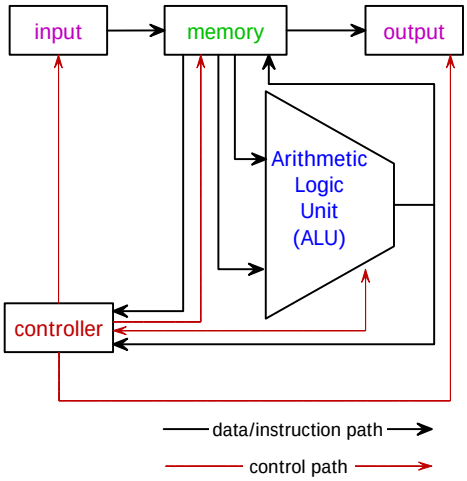
Implementing instructions in a microarchitecture

- Microcode

Slide 3 **Von Neumann Architecture**

In a 1947 paper, **John von Neumann** and others specified the features for an electronic digital computer:

- Digital computation in ALU
- Programmable via set of standard instructions
- Internal storage of data
- Internal storage of program
- Automatic Input/Output
- Automatic sequencing of instruction execution by decoder/controller



The activities in a digital are divided into a sequence of **instructions** — actions performed on data. Instructions move and manipulate data to produce new data according to a specific sequence.

Slide 4 Stages in Computer Design

Instruction Set Architecture (ISA)

The design of a computer begins with the specification of the ISA:

1. Look at the universe of problems to be solved and define the desired capabilities
2. Define a set of atomic operations at level of a system programmer (assembly language)
    - A set of small and orthogonal operations (each performs different task)
    - Instructions in the set can be combined to perform any desired operation
3. Specify the instruction set for the machine language
    - Choose a minimum set of basic operations from all the possibilities
    - Minimize the number of ways to solve the same problem

Implementation

1. Design the machine as a microarchitecture implementation of the ISA
2. Evaluate the machine's theoretical performance
3. Identify problem areas in the machine's performance
4. Improve processor efficiency by redefining operations

Slides 5 — 7 Instruction Set Architecture

Definitions

An **instruction** is a description of an **Operation** performed on **Operands**

An **Operation** is a specific action performed on data.

An **Operand** is a representation of data.

Source operands are the data **inputs** to an operation. **Destination** operands are the data **outputs** from an operation. Operands are specified by an **Addressing Mode** that determines the location of the data in the machine and by the **Data Type** that indicates whether the data is represented as an Integer, Long, Floating Point, Decimal, String, Constant, etc.

As an abstraction, a **general instruction** in an instance of the data structure

Operation	Operand	Operand	...	Operand

where the first field is taken from the set of legal (well-defined) actions on data and the remaining fields are instances of legal addressing modes.
A typical machine instruction has the form

```
ADD destination, source_1, source_2
```

which is interpreted to mean

```
destination ← source_1 + source_2
```

Two data operands are read from source operand locations and added. The sum is stored in the destination operand location.

General operations may act on any number of source operands.

A **unary** operation acts on one source operand.

A **binary** operation acts on two source operands.

An **n-ary** operation acts on n source operands.

An **address specifier** is a special field that describes the format of an operand. It may specify the addressing mode and the operation model (described on slides 13 – 14).

Various names are given to the width of an integer operand. In Intel documentation, an operand may be a byte, word (two bytes), dword (double word = 4 bytes), or quadword (8 bytes). In other architectures, a word is the standard integer length, 32 or 64 bits. **We will state the width of data operands explicitly**.

In slides 8 to 16 we define the basic aspects and features of an instruction set: **operands** (memory and registers), **operation models**, **addressing modes** and **operations**.

Slide 8 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 (monetary cost per byte) with the 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. Cache addressing is similar to RAM addressing — cache 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. Register widths are defined by the standard integer for the CPU.

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 (the data contents 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**.

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 9 Register Naming

The **registers** are part of the CPU design and are named in the design process. Information stored in registers is called **architectural state** and describes machine status and program status.

Registers are divided into **general purpose** and **special purpose**.

General Purpose (GP) registers hold data for instructions. The width of the data register is the width of the standard integer defined in the CPU architecture (usually 32 or 64 bits). Access to registers is by reference to names or numbers.

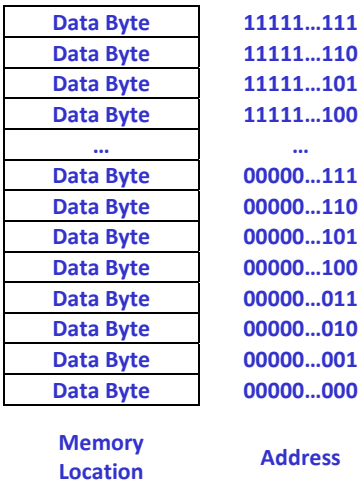
Intel x86 registers are named: **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP**

Registers in other ISAs are numbered: **R0, R1, … , R127**

Special Purpose (SP) registers include **machine status** registers and **Operating System** registers (reserved for use by the OS in supervisor mode).

Slides 10 – 11 Flat Memory Organization

Main memory is organized by an **N-bit physical address** $\mathbf{A = A_{N-1}\ A_{N-2}\ \ldots\ A_1\ A_0}$. The value of A runs from 0 to $2^N - 1$. Each address specifies the storage location of one byte of data. The CPU accesses data in main memory by sending the N-bit address of the first byte for the data. The CPU must contain an N-bit register to hold the physical address.



Since most integers are longer than one byte the ISA must specify the order in memory of the bytes that belong to the integer.

In a **little endian** ISA the least significant byte is stored at lowest address for the integer. The 32-bit integer **69 b3 36 7d** (in hexadecimal notation) is stored at address 0 as

stored byte					69	b3	36	7d
address	07	06	05	04	03	02	01	00

In a **big endian** ISA the most significant byte is stored at lowest address for the integer. The 32-bit integer **69 b3 36 7d** (in hexadecimal notation) is stored at address 0 as

stored byte					7d	36	b3	69
address	07	06	05	04	03	02	01	00

Intel x86 processors are little endian machines.

Slide 12 Specifying Operands

Access to operands is specified by **Addressing Modes**, which are formalized in the following rules:

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. For example,

`LW reg1,IMM(reg2)`

is formalized as

`REGS[reg1] ← MEM[REGS[reg2] + IMM]`

where **`reg2`** holds a pointer to memory and the constant **`IMM`** is added to the pointer by the CPU before the memory access is performed. Slide 16 lists some common addressing modes.

Slides 13 – 14 Structured Operation Models

The operation model in an ISA is the system-level programming model. It specifies the type of ALU to be used in the implementation of defined instructions.

Stack

A stack-oriented ALU maintains a **stack pointer** and uses instructions that **auto-increment** or **auto-decrement** the pointer (add or subtract **`d`** = width of integer):

`Push`	**`Pointer ← Pointer – d`** **`Stack[Pointer] ← memory/register`**
`Pop`	**`memory/register ← Stack[Pointer]`** **`Pointer ← Pointer + d`**
`Binary Op`	**`Stack[Pointer + d] ← Stack[Pointer + d] Op Stack[Pointer]`** **`Pointer ← Pointer + d`**

The high level instruction **`Z = X + Y`** (where **`X, Y, Z`** are pointers to locations in main memory) is compiled to the sequence

```
push X
push Y
ADD        ; adds top two elements of stack to top of stack
pop Z
```

The stack programming model is used in Java byte code.

Accumulator

An accumulator-oriented ALU maintains a special register **A** which is one source and the destination of every ALU instruction. The high level instruction **Z = X + Y** is compiled to the sequence

```
load X    ; copies X to A
add Y     ; adds Y to A
store Z   ; copies A to Z
```

The accumulator programming model is used in hand held calculators.

Register-Memory Model

A Register-Memory ALU stores operands in general registers or in main memory. The high level instruction **Z = X + Y** is compiled to the sequence

```
load R1, X
add R1, R1, Y
store Z, R1
```

The Register-Memory programming model is used in the Intel x86 ISA.

Register-Register Model

A Register-Register ALU must load operands from main memory to general registers before an ALU operation. The high level instruction **Z = X + Y** is compiled to the sequence

```
load R1, X
load R2, Y
add R1, R1, R2
store Z, R1
```

The Register-Register programming model is less convenient for assembly language programming but permits faster implementation in hardware. It is used (at least in internal operations) in all modern CPUs.

Slide 15 **Typical Addressing Modes**

Addressing modes specify the location of an operand. Compilers use certain addressing modes as standard strategies to implement the programming models of high-level languages. Some addressing modes are:

Mode	Assembly Syntax	Operand Location Accessed	Use
Register	**R3**	**Regs[R3]**	Register data used in short-term ALU operations
Immediate	**#3**	**3**	Constant (literal) value. Encoded in instruction — cannot be changed at run time.
Direct (absolute)	**(1001)**	**Mem[1001]**	Static data — placed by OS at load time
Register deferred	**(R1)**	**Mem[Regs[R1]]**	Register R1 holds a pointer to a memory location
Displacement	**100(R1)**	**Mem[100+Regs[R1]]**	Local variables — R1 holds a pointer to the start of a local data frame and 100 is the offset to a named variable

Indexed	**(R1 + R2)**	**Mem[Regs[R1]+Regs[R2]]**	Array addressing — R1 points to the start of a data array and R2 holds the offset to an array element
Memory indirect	**@(R3)**	**Mem[Mem[Regs[R3]]]**	Pointer to pointer
Auto Increment	**(R2)+**	**Mem[Regs[R2]]** **Regs[R2] ← Regs[R2]+d**	Stack access (typically pop)
Auto Decrement	**-(R2)**	**Regs[R2] ← Regs[R2]-d** **Mem[Regs[R2]]**	Stack access (typically push)
Scaled	**100(R2)[R3]**	**Mem[100+Regs[R2]+Regs[R3]*d]**	Complex array indexing — R2 holds array base, 100 is an offset, and R3 is an index that is multiplied by the operand length **d**
PC-relative	**(PC)**	**Mem[PC+value]**	Store data relative to program counter (instruction address)
PC-relative deferred	**1001(PC)**	**Mem[PC+Mem[1001]]**	Store data relative to program counter (instruction address)

Slide 16 Typical Operations

An instruction set can define many types of operation on data, generally classified as:

Data transfer

Load (reg ← mem), store (mem ← reg), move (reg/mem ← reg/mem), convert data types

Arithmetic/Logical (ALU)

Integer arithmetic (+ – × ÷¸ compare shift) and logical (AND, OR, NOR, XOR)

Decimal

Integer arithmetic on decimal numbers

Floating point (FPU)

Floating point arithmetic (+ – × ÷¸ sqrt trig exp …)

String

String move, string compare, string search

Control

Conditional and unconditional branch, call/return, trap

Operating System

System calls, virtual memory management instructions

Graphics

Pixel operations, compression/decompression operations

Classic Computer Organization

In the previous section we saw examples of possible features for an instruction set. Given the various instruction formats, types of operands and addressing modes, possible programming models and instruction types, the next question is **what elements to chose and on what basis?**

In order to understand the choices made in contemporary CPUs, we will discuss the choices made historically in the order these strategies emerged. It will be seen that very few of these strategies have disappeared from modern instruction sets, and very little time will be wasted on "ancient history".

Slides 18 — 21 Considerations in Classic Computer Design

Before the mid-1970s all computers were large, expensive and typically owned by large businesses and institutions. By the late 1960s smaller computers were developed for special purposes. In the mid-1970s "minicomputers" were developed as general-purpose alternatives to large "mainframe" computers.

The first highly successful minicomputer was the VAX introduced in 1977 by the Digital Equipment Corporation (DEC). The VAX designers worked in a technical context that included:

Expensive memory

The wholesale price of RAM in 1977 was about $5000 per MB.

Poor compilers

Compilers were very simple with very limited error messaging and few optimization abilities. As a result, fast and efficient code was usually written, or optimized, in assembly language.

Semantic Gap Argument

The leading theoretical approach to programming language argued that an effective computer language must imitate natural language (spoken language). It should have a large vocabulary of operations and operands, and a high redundancy, meaning that it provides several different ways of programming the same task.

The result of these considerations was the development of powerful and **complex assembly languages**. The classic ISA defines many different types of instruction **syntax** with many **operations** and **addressing modes**. Although learning assembly language was a more difficult task, an experienced programmer could write efficient code easily, choosing the most appropriate methods from equivalent various alternatives. Because each instruction is **complex** and powerful (one instruction can perform many sub-operations), fewer instructions are necessary and program listings are shorter and occupy less memory.

An instruction set architecture designed under this approach is now called **CISC** (Complex Instruction Set Computer). A typical CISC ISA contains:

- More than 300 instruction types
- More than 15 addressing modes
- More than 10 data types
- Automated procedure handling — a single instruction to implement an function call
- Complex machine implementations — a consequence of the complexity of the instruction set. Each defined instruction must be implemented in dedicated hardware.

CISC machines were the conventional wisdom in the mainframe computers of the 1960s and 1970s. There was no other type of general-purpose computer and the term CISC did not yet exist (until there were alternatives in the 1990s). By 1980 all computers could be categorized as:

Mainframes

Mainframes are large and expensive computers, generally owned by big businesses and government agencies. In the 1980s the mainframe of an international bank occupied two entire floors in the World Trade Center. Some manufacturers in the 1970s were IBM, Control Data, Burrows, and Honeywell. Until the 1990s all mainframes were CISC machines.

Minicomputers

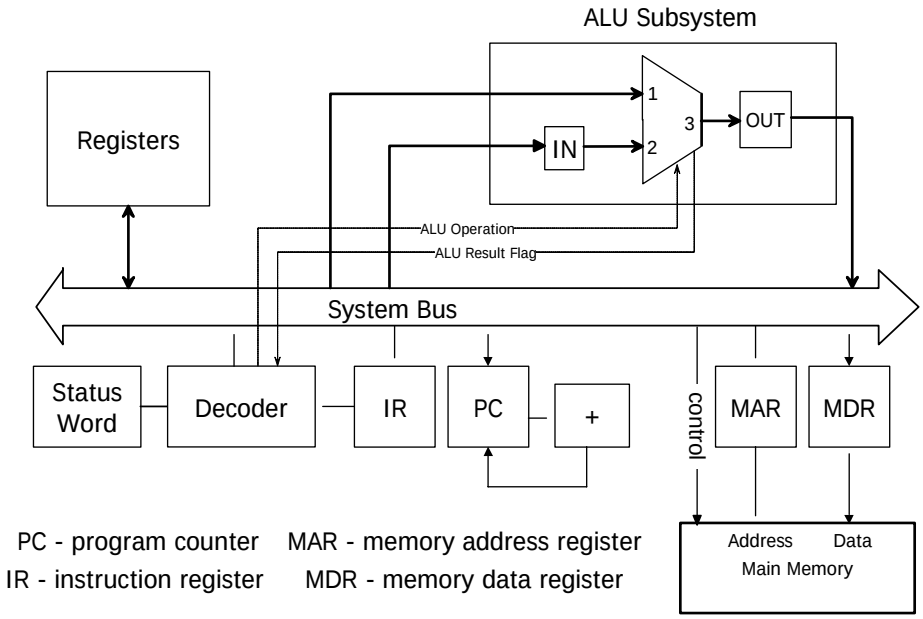
Minicomputers were smaller computers (about the size of a refrigerator) designed for smaller organizations. Unlike mainframes, they could typically run one OS at a time and serve up to about 30 users performing simple tasks. Two manufacturers were Digital (PDP/VAX) and Data General (Eclipse). Because a university department could own a minicomputer, this development promoted the emergence of academic computer science as a separate discipline (by mathematicians, physicists and electrical engineers). The smaller machines required smaller operating systems leading to the development of Unix. Because several small computers may be working on a single large task, it became important to connect them, leading to developments in computer networking such as TCP/IP.

Microcomputers

Microprocessors (a CPU on a single integrated circuit chip) were developed in the 1970s, based on the ISA of a minicomputer. Intel designed the 8086 and 8088 (1979) to operate like a tiny VAX. The Apple II personal computer and IBM's PC took advantage of these CISC type microprocessors. The Intel x86 family used in PCs and servers is the only CISC ISA still widely manufactured.

Slides 22 — 24 Physical Implementation

In order to implement the complex ISA of CISC, the microarchitecture was designed to be generic and easily expandable. Much like the workbench a medieval artisan's workshop, all work passes across the **System Bus** located at the center.



Attached to the System Bus are 5 subsystems:

Registers — the user accessible general-purpose registers defined in the ISA. These are generally numbered R_0 to R_{n-1} (for n registers).

ALU subsystem — the ALU with 2 temporary registers, IN and OUT.

The register IN can store a source operand. The other source operand is provided directly by the System Bus and must be held stable during the ALU operation.

The register OUT holds the ALU result until it can be moved to memory or a register.

Memory subsystem — 2 temporary registers, MAR and MDR

MAR is the memory address register. The MAR holds addresses sent to main memory (external to the CPU) for reading and writing.

MDR is the memory data register. The MDR holds data to be written to memory or data read from memory.

The **control** line specifies a read or write operation to the memory. On read data is transferred from the memory to the MDR. On write data is transferred to the memory from the MDR.

Program counter — the PC register holds the address of the next instruction to be fetched. After a fetch the PC is updated by adding the length of the fetched instruction.

Decoder subsystem — decoder, IR and status word

The **instruction register** (IR) holds the fetched instruction for the decoder.

The **status** word stores flags related to the results of ALU operations (zero, negative, overflow).

The **decoder** converts each instruction to a state machine sequence for performing the instruction.

The **state machine** provided by the decoder is a sequence of commands to the other subsystems. Each command tells the various subsystems to read from an input line or write to an output line. As shown on slide 24, this is accomplished through **Output Enable** (OE) and **Input Enable** (IE) connections on each device.

Device A is connected to Device B through two signal **amplifiers**. In order for data to transfer from Device A to Device B, the decoder will set the OE on Device A (enabling write) and the Input Enable on Device B (enabling read).

For Device 1 to transfer data to Device 2 but not Device 3, the decoder will set OE on Device 1, IE on Device 2 and leave all other IE and OE lines unset. This activates the output amplifier on Device 1, the input amplifier on Device 2, and leaves Device 3 electrically isolated from the system bus (so that it does not participate in the transfer).

A **decoder command** is a word of the type 100010000...00 where each bit sets or unsets the OE or IE of one device. A **machine language instruction** is interpreted (translated) by the decoder into a sequence of **decoder commands** to implement the instruction.

Slides 26 — 26 Instruction Fetch

Fetching an instruction requires a 4 step state machine controlled by the decoder. The steps are:

(1) MAR ← PC

The address of the instruction is transferred to the memory address register (MAR)

(2) READ

The instruction is transferred to the memory data register (MDR)

(3) IR ← MDR

The instruction address is transferred to the instruction register (IR) for the decoder

(4) PC ← PC + length(instruction)

The program counter is updated

These steps are detailed on slides 26 – 29.

Slides 30 — 39 Atomic Operations

The **fetched instruction** is stored in the **instruction register** (IR) and **decoded**. Decoding means translation from machine language to a sequence of atomic operations within the CPU. Each atomic operation includes write (OE) and reads (IE) controlled by the decoder.

As an example consider the machine instruction **`SUB R1, R2, 100(R3)`** defined in the ISA. The source operands are **R2** and **`100(R3)`**. The instruction is formally written:

`Regs[R1] ← Regs[R2] + Mem[ 100 + Regs[R3] ]`

and the sequence of atomic operations is:

`ALU_IN ← R3`	Copy **R3** to the temporary register **`IN`** in the ALU subsystem
`ALU ← 100`	Write **`100`** to the immediate input in the ALU subsystem
ADD	Perform **ADD** on **R3** and **`100`** in the ALU
`MAR ← OUT`	Copy the ALU result from the temporary register **OUT** to **MAR**
READ	Read the memory operand to **MDR**
`ALU_IN ← MDR`	Copy the memory operand from **MDR** to the **`IN`** register in the ALU
`ALU ← R2`	Write **R2** to the immediate input in the ALU subsystem
SUB	Perform **SUB** on **R2** and the memory operand in the ALU
`R1 ← OUT`	Copy the ALU result from the temporary register **OUT** to **R1**

These steps are detailed on slides 31 – 39.

Slides 40 — 42 Microcode

The sequence of atomic operations in the CPU is called a **microprogram** and is written in a syntax of primitives called **microcode**. The decoder **interprets** each machine instruction to a microprogram. The microcode sequence for each machine instruction is stored in the decoder in read only memory (ROM). This method was developed my Maurice V. Wilkes in 1951.

Each line of a microprogram is atomic — it must complete before the next line can begin. The primary reason for this requirement is that only one data value can be written on the system bus at one time.

The **clock cycle** for the CPU must be long enough so that the most complex microcode instruction can be completed in one clock cycle. Since each line of microcode executes in 1 clock cycle, the **number of clock cycles required to execute 1 machine language instruction** is just the number of lines of microcode plus the number of cycles to fetch the instruction.

For example, the instruction **`SUB R1, R2, 100(R3)`** shown above requires 4 CC to fetch and 9 CC to execute. Therefore, this instruction will execute in 13 clock cycles.

The Intel 8086 includes a special subsystem that prefetches instructions whenever the memory is not being used for data access. If the **SUB** instruction is prefetched then it will run in 9 CC instead of 13 CC, a significant optimization.

The **run time** for a program can now be calculated. The total number of clock cycles for the program is the sum of the required CC for each instruction. Many instructions can be divided into types that use system resources in the same way (for example **`ADD R1, R2, R3`** and **`SUB R1, R2, R3`**). So the total number of clock cycles is given by

$$\mathrm{CC}^{\text{program}} = \sum_{\substack{i=\text{instruction} \\ \text{types}}} \left(\text{Instructions of type } i\right) \times \left(\mathrm{CC}^{\text{Instruction of type } i}\right)$$

The **total program run time** is now just $\mathrm{CC}^{\text{program}}$ × seconds per CC = $\mathrm{CC}^{\text{program}}$ / clock rate.

Slide 43 CISC Creates Anti-CISC Revolution

The increased development of CISC-type minicomputers and microprocessor-based personal computers lead to the end of the CISC era. The first 32-bit minicomputer was the **Eclipse** introduced by Data General in 1974. Digital introduced the 32-bit **VAX** in 1977 and it became a major success in the market. Large institutions used the VAX to offload certain applications from the mainframe systems. Intel was still running the assembly line of their Jerusalem factory on VAX systems in the 1990s.

By 1990 minicomputers had turned into powerful servers and workstations, powering the development of UNIX as an operating system for small computers and TCP/IP to interconnect the growing number of machines. Computer Science emerged as separate academic discipline and students needed topics for projects, theses and dissertations. One area for academic research was the performance of small computers. The results were surprising.

Research on minicomputer performance showed that CISC machines use their resources inefficiently. As compilers improved, it turned out that most of the instruction types and addressing modes were never used in converting high level language to machine language. And because CISC machine were designed to be generic and complete they ran more slowly than necessary, carrying the weight of the need to support unnecessary features.

Chapter 3

Quantitative Performance Theory

בפרק 2 ראינו שהמחשבים הקלאסיים שתכננו וייצרו בין 1950 — 1990 היו מסוג CISC, ארכיטקטורות מורכבות עם מימושים מורכבים. המורכבות הזאת נבעה חלקית מאילוצים טכנולוגיים כמו מחסור בזכרון ומהדרים (קומפיילרים) לא מתוחכמים. סיבה נוספת הייתה השערה ששפה עשירה דומה לשפה טבעית תאפשר תכנות יותר טובה. אבל ממחקר על ביצועי מחשבי CISC התברר שחלק מהיכולות של CISC לא בא לידי ביטוי בזמן ריצה ולכן המורכבות תרמה להאטה של המערכת ללא תמורה. הפרק הזה מציג מבוא לחקר הביצועים. בפרקים הבאים נשתמש בכלים האלה לחקור מבנים שהחליפו את ה- CISC, גישה שנקראת RISC — Reduced Instruction Set Computer.

Slides 2 — 3 Amdahl Equation for Multiprocessors

One of the first studies of computer performance was made by **Gene Amdahl** at IBM who studied the disappointing performance of parallel computing in the 1960s.

A **Symmetric Multiprocessor (SMP)** divides a programming task among N identical and equivalent CPUs. A communication network connects the processors, so that the operating system runs on one processor (or more) and the OS assigns tasks to the other processors by some scheduling method. A multicore CPU is a modern example of an SMP. It was once naively expected that programs would run N times as fast on N CPUs, but this goal was never reached.

Amdahl noticed that run time is divided between processes that can be run in parallel and processes that must run sequentially. Only the processes that can be run in parallel are enhanced by increasing the number of processors; the sequential code is the limiting factor in speedup. **Amdahl's equation** expresses this observation:

$$S = \frac{1}{\left(1 - F_P\right) + \frac{F_P}{N}}$$

F_P = fraction of work that can be enhanced (parallelized)

N = speedup for part to be enhanced (number of processors)

Applying Amdahl's equation to a small server with 8 Intel Xeon processors and a workload for which 80% of the work can be run in parallel, the calculated speedup is

$$S = \frac{1}{\left(1 - 0.80\right) + \frac{0.80}{8}} = \frac{1}{0.20 + 0.10} \cong 3.33$$

Even if the number of processors were unlimited, the speedup would be limited by the 20% of the workload that runs sequentially

$$S = \frac{1}{\left(1 - F_P\right) + \frac{F_P}{N}} \xrightarrow[N \to \infty]{} \frac{1}{1 - F_P} = \frac{1}{1 - 0.80} = 5$$

and the maximum speed would be 5. This result shows that future enhancements to programming speed must come from increasing the degree of parallelization, that is, from better parallel programming.

Slide 4 Basic Performance Measures

As said in chapter 1, the most important performance characteristic is processing speed, measured by **Response Time**, which is the elapsed time T from start to finish of a defined program task. **Run time** is the response time for a complete program, and includes both useful work performed by the CPU and time spent waiting for external events.

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

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

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

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

Slide 5 Processor Performance Enhancements

For a single CPU, performance can be improved by making changes to the processor hardware and to the software. Some possible changes are:

Hardware Enhancements

Increasing the clock rate

Improving instruction implementation to reduce the number of clock cycles per instruction

Improving memory organization to reduce latency in data access

Increasing the number of processing elements — multiple CPUs, multiple ALUs in each CPU, additional registers to reduce memory access

Software Enhancements

Use of run time optimizations that use CPU features more effectively

Improving the compiler to produce more efficient code

Improving the operating system to reduce run time of system code and more effectively manage threads

General outlook

It may already be clear to students of computer science that time-to-market and generic code reusability are basic goals in software development today. Software performance and code efficiency are generally of interest in specific sectors — gaming, high performance scientific computing and embedded processor-based systems. Hardware must become increasingly faster to permit increasingly bloated code to run at all.

Enhanced Run Time

The total run time is the sum of all the partial run times. Any given enhancement may affect partial run times so that some are longer, some are shorter, and some are unchanged. The speedup will be greater than 1 (net improvement) as long as the sum of new partial run times is less than the sum of old partial run times. This is seen in slide 6, where blue sections of run time are enhanced while green sections remain unchanged.

Slides 7 — 8 Amdahl Equation

Another form of Amdahl's equation can be found by comparing the sums of partial run times. The basic definitions are:

T	total run time of a task = the sum of the partial run times
T'	total run time of a task after enhancement = the sum of the partial run times after enhancement
t_e	partial run time that can be enhanced = sum of partial run times that can be enhanced
t_e'	partial run time that can be enhanced, after enhancement = sum of partial run times that were enhanced $t_e' < t_e$
t_0	partial run time that cannot be enhanced and does not change

From these definitions, we define two relative quantities:

F_e	Relative fraction of the run time that can be enhanced = $t_e / T \leq 1$
S_e	Speedup of the portion of run time that can be enhanced = $t_e / t_e' > 1$

By simple algebra we find Amdahl's equation in the form

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e} = \frac{1}{\frac{t_0}{T} + \frac{t'_e}{T}} = \frac{1}{\frac{T - t_e}{T} + \frac{t'_e}{t_e}\frac{t_e}{T}} = \frac{1}{1 - F_e + \frac{1}{S_e}F_e}$$

which expresses the speedup in terms of relative quantities. The actual run times are not required to calculate speedup if the relative enhancements are known.

The form of Amdahl's equation for SMP is found by assigning

F_e = fraction of program that can be parallelized = F_p

S_e = speedup of parallelizable part of program = number of processors N

As an example of how Amdahl's equation works, consider the following enhancement to the float instructions:

	Before		After	
Integer	t_{INT}	100 ms	t_{INT}	100 ms
Float	t_{FP}	300 ms	t_{FP}'	200 ms
Total	T	400 ms	T'	300 ms

The partial run time of the floats is decreased from 300 ms to 200 ms and so the total run time is decreased from 400 ms to 300 ms. The speedup is S = T/T' = 400 ms / 300 ms = 4/3.

The relative quantities and speedup from Amdahl's equation are

$$F_e = \frac{t_{FP}}{T} = \frac{300\ \text{ms}}{400\ \text{ms}} = \frac{3}{4} = 75\%$$

$$S_e = \frac{t_{FP}}{t_{FP}'} = \frac{300\ \text{ms}}{200\ \text{ms}} = \frac{3}{2} = 1.50$$

$$S = \frac{1}{1 - F_e + \frac{F_e}{S_e}} = \frac{1}{1 - \frac{3}{4} + \frac{3/4}{3/2}} = \frac{1}{\frac{1}{4} + \frac{1}{2}} = \frac{4}{3}$$

Slide 9 Application of Amdahl Equation

To see how the Amdahl equation can be applied, we consider two alternative enhancements to a CPU and use Amdahl's equation to determine which provides better speedup.

For the CPU under study, Float (FP) instructions account for 50% of total run time. Among all the FP instructions, just the square root (FP) alone accounts for 20% of total run time.

Two enhancements are proposed:

Speedup of $S_e = 2$ for all FP instructions

Speedup of $S_e = 10$ for just the square root instruction

The information provided is in the form required for Amdahl's equation.

Enhancement 1

$F_e = 0.50$ and $S_e = 2$ leads to

$$S_1 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.50 + \frac{0.50}{2}} = \frac{1}{0.50 + 0.25} \cong 1.33 \Rightarrow 33\% \text{ speedup}$$

Enhancement 2

$F_e = 0.20$ and $S_e = 10$ leads to

$$S_2 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.20 + \frac{0.20}{10}} = \frac{1}{0.80 + 0.02} \cong 1.22 \Rightarrow 22\% \text{ speedup}$$

This shows a general rule of thumb known informally as **Amdahl's "law"** — a smaller partial speedup with a wide area of application provides a larger overall speedup than a large partial speedup with a small area of application.

Slides 10 — 11 Generalized Amdahl Equation

Most practical enhancements will **degrade** some partial run times.

Besides t_e = the run time that can be **enhanced** ($t_e' < t_e$) and t_0 = the **unchanged** run time there will also be t_d = the run time that is **degraded** ($t_d' > t_d$).

In relative terms, $F_d = t_d / T$ (the relative fraction of run time that is degraded) and $S_d = t_d / t_d' < 1$ (the speedup of the degraded partial run time). Now Amdahl's equation is

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e + t'_d} = \frac{1}{\frac{T - t_e - t_d}{T} + \frac{t'_e}{t_e}\frac{t_e}{T} + \frac{t'_d}{t_d}\frac{t_d}{T}} = \frac{1}{1 - (F_e + F_d) + \frac{F_e}{S_e} + \frac{F_d}{S_d}}$$

As an example, the shift from a CISC ISA to a RISC ISA makes 94% of run time 5 times faster ($S_e = 5$), degrades 1% of the run time by a factor of 10 ($S_d = 1/10$), and leaves 5% of run time unchanged. From the Amdahl equation we find

$$S = \frac{1}{1 - (F_e + F_d) + \frac{F_e}{S_e} + \frac{F_d}{S_d}} = \frac{1}{1 - (0.94 + 0.01) + \frac{0.94}{5} + \frac{0.01}{1/10}} = \frac{1}{0.05 + 0.19 + 0.10} \cong 2.94$$

The speedup is about 3 times faster than CISC, even though some operations are slower.

Slides 12 — 13 Detailed Analysis of CPU Run Times

The use of the Amdahl equation requires us to know the relative run time data. This may be useful in qualitative comparisons, but the relative numbers are not generally available.

The detailed run time information — as seen in the SPEC benchmark — is generally found from measurements on running programs. But to make measurements on running programs requires an implementation of the CPU, and we are not usually interested in building the CPU until we can predict its performance. **CPU analysis** predicts run time without having to build the CPU.

Assumptions in CPU analysis

1. All instructions can be grouped together according to resource usage. For example, ADD R1, R2, R3 and SUB R1, R2, R3 differ only in the ALU operation
2. All instructions in a group run in the same number of clock cycles
3. Every clock cycle measures same the unit of time
4. Instruction run time = seconds per clock cycle × number of clock cycles per instruction
5. Group run time = instruction run time × number of instructions in the group
6. Total run time = sum of instruction group run times

The **basic definitions** are

$T =$ total run time of program

$t_i =$ total run time of instructions in group i

$IC_i =$ number of instructions in group i (Instruction Count)

$CPI_i =$ number of clock cycles to run 1 instruction in group i (Cycles Per Instruction)

$N_i =$ number of clock cycles to run all instructions in group i

$\tau =$ seconds per clock cycle = clock period

$R =$ clock rate $=$ clock frequency $=$ clock cycles per second $=$ Hertz (Hz) $= 1/\tau$

Using these definitions we may calculate the following quantities:

$IC =$ total number of instructions in the program

$N =$ total number of clock cycles to run the program

$CPI =$ average number of clock cycles per instruction for the program = $\frac{N}{IC}$

In general, we write the value of some *quantity* after enhancement as *quantity*'.

Slides 14 — 15 CPU Equation

The **average CPI** (cycles per instruction) characterizes the effort required by a CPU in processing instructions. A CPU with a low CPI requires fewer clock cycles to run a typical instruction than a CPU with a high CPI. The **CPU equation** permits calculation of the average CPI (cycles per instruction) from detailed information about the CPU.

The **number of clock cycles** required to run all instructions of **type *i*** is given by

$$N_i = \text{number of instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type i}} = IC_i \times CPI_i$$

Summing over all of the instruction groups, the **total number of clock cycles** required to run all instructions is

$$N = \sum_{\text{all groups } i} N_i = \sum_i IC_i \times CPI_i$$

The average number of clock cycles per instruction is defined as

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

and using the expression for N above leads to

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

Notice that the ratio $\frac{IC_i}{IC}$ is the proportion (percentage) of instructions from group i in the full program and since the sum of these ratios is 1, the set of ratios forms a **probability distribution** for the instructions in the program. The average CPI is a weighted average of the individual values of CPI_i for each group i.

Slide 15 shows an **example** of how these calculations fit together. A certain **program** has 5000 integer instructions, 1000 branch instructions, and 4000 load/store instructions. The total number of instructions is the sum IC = 10,000. The program is thus 50% integer, 10% branch and 40% load/store.

On a certain CPU the CPI_i (number of clock cycles required to run one instance of an instruction from group i) is 4 CC for integer, 12 CC for branch, and 8 CC for load/store. All the integer instructions together require 4 CC per instruction × 5000 instructions = 20,000 CC to run. Similarly, the branch instructions run in 12,000 CC and the load/store instructions run in 32,000 CC. Adding these together, the complete program will run in N = 64,000 CC.

The average number of cycles per instruction is then CPI = N / IC = 60,000 / 10,000 = 6.4

The same number is found from the CPU equation as

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.50 + 12 \times 0.10 + 8 \times 0.40 = 6.4 \text{ cycles per instruction}$$

Slides 16 — 17 CPU Run Time

The **time required to run one instance of an instruction of type *i*** is just the number of clock cycles to run that instruction multiplied by the number of seconds in a clock cycle:

$$\text{single instruction run time} = \frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}} = CPI_i \times \tau$$

Multiplying by the number of instruction in the group, the **run time for all instructions of type *i***

$$t_i = \text{instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}} = IC_i \times CPI_i \times \tau$$

Adding the contributions from all groups, the **total run time for the program** is

$$T = \sum_{\textbf{all groups } i} t_i = \sum_i CPI_i \times IC_i \times \tau = \left(\sum_i CPI_i \times \frac{IC_i}{IC} \right) \times IC \times \tau$$

Since the quantity in parentheses is the average CPI (from the CPU equation), this is

$$T = CPI \times IC \times \tau = \text{clock cycles per instruction} \times \text{number of instructions} \times \text{seconds per clock cycle}$$

Slide 17 shows an **example** of the run time calculation. The instructions are divided into 5 instruction groups. For each group we are shown the percentage of instructions in that group and the number of clock cycles required to run one instruction from that group. Using the CPU equation, the average CPI is found to be 10.5 clock cycles per instruction. We are given the total number of instructions in the program is 1,000,000 and the clock speed R = 100 MHz. Since the clock cycle time is 1 / R, we can calculate the program run time as 0.105 seconds.

Slides 18 — 20 C Code to Runtime Example

Another example begins with a program written in C.

The program is **compiled** to assembly language and the assembly code is sorted into **instruction types**: load, $ALU_{\text{auto-increment}}$, ALU, and JMP. The first two instructions run once each (2 instructions) and the three instructions in the loop run 5 time each (15 instructions) for a total IC =17. In relative terms, 1/17 = 6.5% and 5/17 = 29%. So the loads are 13% and the other three types are 29% each.

On slide 19 we show a sample interpretation of an assembly instruction into a **microprogram**. Assuming an instruction queue mechanism (as in the 8086) that prefetches instructions, this instruction will run in $CPI_{\text{ALU-AI}}$ = 9 clock cycles (9 lines of microcode).

Slide 20 shows the results of similar analysis for the other instructions. With the distribution of instructions in the program and the CPI_i of each instruction type, the average CPI is found from the **CPU equation** to be 7.2. The total **number of clock cycles** is N = CPI × IC = 7.2 × 17 = 122. Multiplying by the clock period (10^{-9} sec at 1 GHz) gives the **run time** as 0.122 microseconds.

Slides 21 Applying CPU Equation

The main application of the CPU equation is to calculate speedup. The run time before enhancements is calculated as in the examples above. Following some architectural enhancement, there will be new values of IC', CPI' and τ'. These permit calculation of the new run time and the speedup as

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'}$$

Slides 22 — 30 CPU Equation — Example 2

As another example, on a certain CPU:

25% of all instructions in an average program are floating point (IC_{FP} / IC = 0.25). The FP group includes 5 operations: ADD, SUB, MULT, DIV, SQRT.

Each FP instruction runs in a different number of clock cycles. The average FP instruction runs in 4 clock cycles (CPI_{FP} = 4).

The SQRT instruction is the most complex. Only 2% of all instructions in the typical program are square root (IC_{SQRT} / IC = 0.02) but each SQRT runs in 20 cycles (CPI_{SQRT} = 20).

The average CPI for all other instructions in program is 4/3 clock cycles (CPI_{other} = 4/3). Since the FP instructions are 25% all the other instructions are 75%.

Based on this information we can calculate

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = CPI_{FP} \times \frac{IC_{FP}}{IC} + CPI_{other} \times \frac{IC_{other}}{IC} = 4 \times 0.25 + \frac{4}{3} \times (1 - 0.25) = 2.00$$

Slide 23 describes two possible enhancements:

Enhancement 1 improves performance of all FP instructions

Reduce average CPI_{FP} = 4 cycles to CPI_{FP}' = 2 cycles

There is no change in the program content so $IC_i' = IC_i$ for all instruction types

There is no change to clock rate $\tau' = \tau$

Enhancement 2 improves performance of SQRT (FP) instruction

Reduce CPI_{SQRT} = 20 cycles to CPI_{SQRT}' = 2 cycles

There is no change in the program content so $IC_i' = IC_i$ for all instruction types

There is no change to clock rate $\tau' = \tau$

Slide 24 shows the straightforward **calculation for enhancement 1**. CPI_{FP} = 4 cycles is replaced by CPI_{FP}' = 2 cycles and all other factors remain unchanged. The average CPI changes from 2 CC per instruction to 1.5 CC per instruction. The speedup is then found from S = T / T' where since $T' = CPI' \times IC' \times \tau' = CPI' \times IC \times \tau$ the speedup is just the ration of CPIs = 1.33. Enhancement 1 provides a 33% speedup in performance.

Slide 25 shows that calculation of the **speedup for enhancement 2** requires splitting the FP group into SQRT and FP/SQRT which is the group of FP instructions without SQRT, {ADD, SUB, MULT, DIV}. Since FP is 25% of all instructions and SQRT is 2% of all instructions, the group FP/SQRT is 23% of all instructions. Now the CPI' is found from

$$CPI' = \sum_i CPI'_i \times \frac{IC_i'}{IC'}$$

$$= CPI'_{FP/SQRT} \times \frac{IC_{FP/SQRT}'}{IC'} + CPI'_{SQRT} \times \frac{IC_{SQRT}'}{IC'} + CPI'_{other} \times \frac{IC_{other}'}{IC'}$$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{other} \times \frac{IC_{other}}{IC}$$

where only CPI'_{SQRT} is changed from its previous value. But there is one missing piece of information that must be calculated first: We do not know the $CPI_{FP/SQRT}$ of the group FP/SQRT.

To find $CPI_{FP/SQRT}$ we must use the definition of average CPI. We were told that the average CPI_{FP} for the group of FP instructions is 4. This means that the total number of CC for FP instructions divided by the total number of FP instructions is 4.

$$4 = CPI_{FP} = \frac{\text{total } FP \text{ cycles}}{FP \text{ instructions}} = \frac{N_{FP}}{IC_{FP}} = \frac{\sum_{k \in FP} N_k}{IC_{FP}} = \sum_{k \in FP} \frac{CPI_k \times IC_k}{IC_{FP}} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

The parameter that we must find is $CPI_{FP/SQRT}$ which is defined as

$$CPI_{FP/SQRT} = \frac{\text{total } FP/SQRT \text{ cycles}}{FP/SQRT \text{ instructions}} = \frac{N_{FP/SQRT}}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} \frac{N_k}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}}$$

On slide 27 we combine these expressions:

1. Write the definition of CPI_{FP}

$$CPI_{FP} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

2. Split the sum into FP/SQRT and SQRT

$$= \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} + CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}}$$

3. Insert two factors of 1

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} \right] \times \frac{IC_{FP/SQRT} / IC}{IC_{FP/SQRT} / IC} + \left[CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}} \right] \times \frac{1/IC}{1/IC}$$

4. In the first term exchange IC_{FP} with $IC_{FP/SQRT}$. In the second term move $1/IC$ inside.

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}} \right] \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

5. The factor in parentheses in the first term is precisely the definition of $CPI_{FP/SQRT}$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

Plugging in the numbers we know we find

$$4 = CPI_{FP/SQRT} \times \frac{0.23}{0.25} + 20 \times \frac{0.02}{0.25} \quad \Rightarrow \quad CPI_{FP/SQRT} = 2.61$$

We notice that the large $CPI_{SQRT} = 20$ raises the average CPI_{FP} from 2.61 to 4 even though SQRT is only 0.02 / 0.25 = 8% of all FP instructions.

On slide 28 we find the **speedup for enhancement 2**. Using

$$CPI' = CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{other} \times \frac{IC_{other}}{IC} = 1.64$$

the speedup is S = T / T' = 2 / 1.64 = 1.22.

We see that the dramatic improvement in SQRT has less overall effect than the more moderate improvement to all the FP instructions in enhancement 1.

Slide 29 shows a trick for calculating $CPI_{FP/SQRT}$ in the case that the instruction count does not change:

Add zero = $CPI - CPI$

$$CPI' = CPI - \left(CPI - CPI'\right)$$

Expand CPI and CPI'

$$= CPI - \left(\sum_i CPI_i \times \frac{IC_i}{IC} - \sum_i CPI'_i \times \frac{IC'_i}{IC'} \right)$$

Change order of addition and subtraction

$$= CPI - \sum_i \left(CPI_i \times \frac{IC_i}{IC} - CPI'_i \times \frac{IC'_i}{IC'} \right)$$

When $IC'_i = IC_i$ remove common factor of $\frac{IC_i}{IC}$

$$CPI' = CPI - \sum_i \left[\left(CPI_i - CPI'_i\right) \times \frac{IC_i}{IC} \right]$$

For the case of $SQRT$

$$= CPI - \left[\left(CPI_{SQRT} - CPI'_{SQRT}\right) \times \frac{IC_{SQRT}}{IC} \right]$$

$$= 2.00 - \left[\left(20 - 2\right) \times 0.02 \right]$$

$$= 1.64$$

This method only works if the structure of the program does not change.

It should be recalled that the enhancements in this example are similar to those in the example of the **Amdahl equation** (slide 9) and the speedups are identical. Slide 30 shows that the relative quantities given in the case of Amdahl can be derived from the parameters in this example.

From the definitions of F_e and S_e (slide 7) we can write

$$F_e^{FP} = \frac{t_{FP}}{T} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI \times IC \times \tau} = \frac{4 \times \left(0.25 \times IC\right) \times \tau}{2 \times IC \times \tau} = 50\%$$

$$S_e^{FP} = \frac{t_{FP}}{t'_{FP}} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI'_{FP} \times IC'_{FP} \times \tau'} = \frac{4}{2} = 2$$

$$F_e^{SQRT} = \frac{t_{SQRT}}{T} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI \times IC \times \tau} = \frac{20 \times \left(0.02 \times IC\right) \times \tau}{2 \times IC \times \tau} = 20\%$$

$$S_e^{SQRT} = \frac{t_{SQRT}}{t'_{SQRT}} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI'_{SQRT} \times IC'_{SQRT} \times \tau'} = \frac{CPI_{SQRT}}{CPI'_{SQRT}} = \frac{20}{2} = 10$$

which recovers the parameters given in the Amdahl equation example.

Slides 31 — 32 Changing Instruction Mix

Up to now we have discussed examples in which an architectural enhancement changes CPI_i for some group but the **instruction distribution** represented by the ratios IC_i / IC is unchanged. Because CPI is a weighted average of the CPI_i values for the various groups, changing the instruction mix will generally change CPI.

As an example, slide 31 shows a CPU that runs a **program of 10,000 instructions** for which the average CPI is calculated to be 6.4 cycles per instruction. The (unrealistic) enhancement made in this case is that **2,000 ALU instructions are erased** from the program (with no reasonable explanation).

For the new program, the total IC is now 10,000 – 2,000 = 8,000. As a result, all of the relative weights IC_i / IC are changed. For example, IC_{branch} = 1000 / 10000 = 10% becomes IC'_{branch} = 1000 / 8000 = 12.5%.

The removal of many fast ALU instructions (CPI_{ALU} = 4) means that this group has a smaller weight in the program: from 50% down to 37.5%. Calculation of the **average CPI** shows that it rises from 6.4 to 7.0 because the slower instructions (with more cycles per instruction) are a bigger part of the instruction distribution.

As shown on slide 32 the **speedup** is 14%, because the rise in CPI is compensated by the fall in the instruction count. This example shows that speedup must be calculated from comparative run times and not only from CPI.

Slides 33 — 35 The Instructions Per Second Myth

Two measures often used to describe computing power are

MIPS = million instructions per second

FLOPS = floating point operations per second

But these measures cannot provide a fair comparison because

$$\text{MIPS} = \frac{\text{instructions} / 10^6}{\text{run time}} = \frac{IC}{10^6 \times T} = \frac{IC}{10^6 \times CPI \times IC \times \tau} = \frac{R}{10^6 \times CPI} = \frac{R(\text{in MHz})}{CPI}$$

which does not include instruction count.

For example, consider similar processors **CPU-1** and **CPU-2** which both run ALU instructions in 1 cycle and all other instructions in 2 cycles. Both have clock speed of 1 GHz = 1000 MHz. The processors differ in their compiler:

CPU-1 compiler produces 50% ALU instructions and 50% other

CPU-2 compiler is more efficient and produces 25% fewer ALU instructions than CPU-1

The CPU equation provides $CPI_1 = 1 \times 0.50 + 2 \times 0.50 = 1.50$ which means that

$$\text{MIPS}_1 = \frac{1000}{1.50} \cong 667 \text{ million instructions/sec}$$

To find the CPI for CPU-2 we first find the instruction distribution (slide 34):

We know that on CPU-1, $IC_1^{ALU} / IC_1 = 50\%$ and so $IC_1^{ALU} = 0.5 \times IC_1$.

We also know that on CPU-2 the compiler produced 25% fewer ALU instructions which means that $IC_2^{ALU} = (1 - 0.25) \times IC_1 = 0.375 \times IC_1$.

The other instructions are not affected and so $IC_2^{other} = IC_1^{other} = 0.5 \times IC_1$ (the other 50% of instructions). Thus

$IC_2 = IC_2^{ALU} + IC_2^{other} = 0.375 \times IC_1 + 0.5 \times IC_1 = 0.875 \times IC_1$

meaning that CPU-2 has 87.5% as many instructions as CPU-1 (12.5% fewer instructions).

This allows us to calculate

$IC_2^{ALU} / IC_2 = 0.375 \times IC_1 / (0.875 \times IC_1) = 0.375 / 0.875 = 0.43$

$IC_2^{other} / IC_2 = 0.5 \times IC_1 / (0.875 \times IC_1) = 0.5 / 0.875 = 0.57$

Now the average $CPI_2 = 1 \times 0.43 + 2 \times 0.57 = 1.57$ which is larger than $CPI_1 = 1.5$. It is not surprising that CPU-2 has a **lower MIPS** of 1000 / 1.57 = 637 score than CPU-1.

Nevertheless, we expect that CPU-2 is faster than CPU-1 and calculating the speedup

$$S = \frac{T_1}{T_2} = \frac{CPI_1 \times IC_1 \times \tau_1}{CPI_2 \times IC_2 \times \tau_2} = \frac{CPI_1 \times IC_1}{CPI_2 \times IC_2} = \frac{1.50 \times IC_1}{1.57 \times (0.875 \times IC_1)} \cong 1.09$$

we see that this is true.

Slides 36 — 39 Replacing Instruction Types

The problem of changing the instruction distribution begins with summing the IC_i for the instruction groups. Generally,

$IC = IC_1 + IC_2 + ... + IC_n$

Where Type 1 = ALU, Type 2 = Conditional Branch, and so on.

Replacing an instruction in the ISA may affect more than one instruction type. For example, the Intel instruction **`loop target`** replaces the 3 instructions

```
DEC CX          ; CX ← CX – 1 (ALU type on register CX)
CMP CX, 0       ; compare CX == 0   (ALU type)
JNZ target      ; branch to target if CX != 0 (branch type)
```

Adding the loop instruction to the ISA will change both IC'_1 and IC'_2.

Slides 37 – 39 show another example, the **addition of a floating point unit** (FPU) to a CPU which did not have one. If no FPU is present in a CPU then FP instructions are executed by **emulation**. This means that the FP operation is converted to a sequence of integer instructions.

For example, to multiply $(2.165 \times 10^4) \times (3.247 \times 10^{-3})$ the compiler produces code that performs integer multiplication of 2165 × 3247 = 7029755, calculates exp = 4 – 3 + (–3) – 3 = –5 and shifts the result to $7.030 \times 10^6 \times 10^{-5} = 7.030 \times 10^1$. In this situation, the compiler produces a large number of ALU instructions that emulate FPU operations.

Slide 37 shows the instruction distribution for a CPU with no FPU. ALU instructions run in 1 CC per instruction are since they are 75% the average CPI is small.

Slide 38 shows the new distribution when an FPU is added to the CPU.

First we split the old ALU instructions into two groups: ALU^{INT} represents actual ALU instructions and $ALU^{EMULATION}$ represents ALU instructions for FPU emulation. We are told that 2/3 of the old ALU instructions emulate FPU instructions. Therefore

$ALU^{EMULATION}$ = (2/3) × 75% = 50% of all instructions

ALU^{INT} = (1/3) × 75% = 25% of all instructions

The ALU^{INT} instructions will still be in the program after addition of the FPU, but the $ALU^{EMULATION}$ instructions will be replaced. We are told that 1 new FPU instruction replaces 10 $ALU^{EMULATION}$ instructions. Therefore

IC'_{ALU} = 0.25 IC

IC'_{FPU} = (1/10) × 0.50 IC = 0.05 IC

The values of IC for the other groups are unchanged so that IC'_{other} = (1 – 0.75) IC = 0.25 IC and

IC' = IC'_{ALU} + IC'_{FPU} + IC'_{other} = 0.55 IC

Slide 39 shows the calculation of the new distribution and the average CPI'. For example

IC'_{ALU} / IC' = 0.25 IC / 0.55 IC = 0.25/0.55 = 0.46

The average CPI' = 0.95/0.55 is found from the CPU equation.

The speedup is found by comparing the run times

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{1.25 \times IC}{\frac{0.95}{0.55} \times (0.55 IC)} = \frac{1.25}{0.95} \cong 1.32$$

where we see that the factor of 0.55 cancels as expected.

Slides 40 — 43 Load-Store versus Register-Memory

As another example, we consider the **register-memory instructions**.

CPU-1 is a load-store machine — all ALU operands must come from a register, so that a memory operand must be loaded to a register before an ALU operation uses it. Slide 40 shows the **instruction distribution**, the $\mathbf{CPI_i}$ for the instruction groups and the calculation of the average **CPI**.

We are told that 25% of all ALU memory operands are used only in only 1 ALU operation. This means that 75% of memory operands are used in more than 1 operation and it is more efficient to keep them in a register for multiple accesses.

To consider the register-memory instruction for the 25% of memory operands that are used just once, we define CPU-2 as an **ideal register-memory machine** that operates as follows:

> For 75% of memory operands used in multiple ALU operations, the perfect compiler loads these **"multiple" memory operands** to registers.
>
> For 25% of ALU memory operands used in only a single ALU operation, the perfect compiler does not load these **"single" memory operands** to registers, but uses them directly from memory.

To calculate the speedup we split the ALU operations into ALU^{multi} and ALU^{single} so that we replace ALU^{single} with $ALU^{register\text{-}memory}$ and cancel 1 register load for every ALU^{single}.

Of the 40% of instructions that are ALU, 75% are ALU^{multi} so $IC^{ALU\text{-}multi}$ = 0.30 IC and $IC^{ALU\text{-}single}$ = 0.10 IC. This split is shown on slide 42.

We must also split the load instructions into $IC^{load\text{-}single} = IC^{ALU\text{-}single}$ =0.10 IC (one load instruction for each ALU^{single} instruction) and $IC^{load\text{-}multi} = IC^{load} - 0.10\ IC = (0.25 - 0.10)\ IC = 0.15\ IC$.

Since the load and ALU take 4+5 clock cycles, the new instruction $ALU^{register\text{-}memory}$ runs in 9 clock cycles.

Slide 43 shows the new table after replacing ALU^{single} and $load^{single}$ with $ALU^{register\text{-}memory}$. The total instruction count is IC – 10% IC – 10% IC + 10% IC = 0.90 IC.

Calculating CPI' we see a small rise which exactly compensates the smaller number of instructions. All together, the speedup is 1 meaning no change in performance.

Slides 44 – 50 Analysis of 8086 Example

In chapter 1 (slides 17 – 22) we considered compiler optimization for an Intel 8086 CPU. Slide 44 shows the program before optimization, the number of clock cycles per instruction and the number of time each instruction runs. Adding the entries in the Runs column we see that the total number of instructions is IC = 7N – 3. The IC_i / IC for each row is just the entry in Runs divided by 7N – 3.

In order to compute IC_i / IC we must first find averages for the instructions that require different numbers of cycles in different conditions. On slide 45 we calculate the averages from their definitions.

For example JGE runs N – 1 times in 4 CC and 1 time in 13 CC. The total number of runs is N – 1 + 1 = N and the total number of cycles is 4(N – 1) + 13. We write $CPI_{JGE} = (4(N - 1) + 13)/N$. We have a similar result for the store.

Slide 46 shows the instruction distribution for the program and the CPI_i for each row. Slide 47 shows the numerical values for N = 100. Using these values we calculate CPI = 9.45 and using R= 4 MHz, find the run time T = 1.646 ms.

Slide 48 shows the optimized program (using register variables which improves CPI_i for access to operands) and the slide 49 shows the new distribution after replacement of memory access instructions by register access instructions. The updated CPI' = 4.31 is found on slide 50 and the run time T' = 0.7515 ms. The speed up is thus S = T / T' = 2.19.

Slides 51 – 60 Software Content — Instruction Distribution

The method of CPU analysis is summarized on slides 51 – 60.

- Slide 51 shows a sample instruction distribution with the CPI_i value for each group.
- Slide 52 calculates the total IC
- Slide 53 calculates the relative values IC_i / IC
- Slide 54 calculates the relative run time per instruction group $CPI_i \times IC_i$ / IC
- Slide 55 calculates the average CPI
- Slide 56 calculates the total number of clock cycles N and the run time T
- Slide 57 calculates the number of clock cycles N_i per instruction group
- Slide 58 compares the results of slides 56 and 57
- Slide 59 calculates the relative fraction of run time F_i
- Slide 60 shows the relationship of CPU analysis and Amdahl's law

Slide 61 **Instruction Distributions**

CPU analysis permits performance analysis of a new machine design "on the drawing board" and allows us to evaluate a proposed design without building a CPU implementation. The procedure can be summarized as:

1. Specify Instruction Set Architecture (ISA). The ISA describes the machine language for proposed CPU and provides a human-readable assembly language. The proposed microarchitecture associated with the ISA determines CPI_i for each instruction group i, by allowing us to count the number of clock cycles required to implement a single instruction in the ISA.
2. Write C, C++, Fortran compilers for the proposed machine language
3. Compile representative programs to machine language. To insure that the programs are representative, we can use programs from SPEC CINT and CFP.
4. Sort the instructions in the compiled machine language code into groups and calculate the relative instruction count IC_i/IC for each group i.
5. Calculate average CPI and the run time T based on a predicted clock speed.
6. Compare the run time with a reference machine to decide whether the CPU design is worth implementing.

Chapter 4

From CISC to RISC

בפרק 2 ראינו שהמחשבים הקלאסיים שתכננו וייצרו בין 1950 עד 1990 היו מסוג CISC, ארכיטקטורות מורכבות עם מימושים מורכבים. בפרק 3 למדנו גישה לביצועי מעבד ושיטות לניתוח ביצועים. ממחקר על ביצועיי מחשבי CISC התברר שחלק מהיכולות של CISC לא בא לידי ביטוי בזמן ריצה ולכן המורכבות תרמה להאטה של המערכת ללא תמורה. בשנות ה- 1980 הציעו גישה חלופית בשם RISC על בסיס "חוק" אמדהל שמדגישה פעולות נפוצות על חשבון פעולות אחרות. בפרק הזה נציג תוצאות מחקר על מחשבי CISC כדי לבחור מרכיבים ל- ISA בסוג RISC — Reduced Instruction Set Computer.

Slides 2 – 3 **CISC Creates the Anti-CISC Revolution**

In 1977 the Digital Equipment Company (DEC) introduced the VAX, the first commercially successful 32-bit CISC minicomputer. Because minicomputers were much cheaper than large mainframes, serious computers became available to small organizations for the first time in the 1970s and 1980s. As CISC minicomputers were more widely used they opened new possibilities and led to new developments. The UNIX operating system was developed as a minicomputer operating system and TCP/IP protocols were developed to support networks of minicomputers. In the late 1970s computer science emerged as separate academic discipline, bringing together mathematicians, physicists and electrical engineers into new academic departments. CS students needed topics for final projects, theses and dissertations — some performed research on CISC performance.

The results of the performance studies on CISC machines were surprising. The complexity of the CISC ISA was driven by technical limitations, such as limited memory space and very simple compilers. But there was also a belief that a complex machine language, similar to natural language, would produce efficient code. Research revealed that most of the machine language instructions defined in the CISC ISA are never used. In order to implement the complex instruction set, CISC implementations give up speed in favor of generality. Thus, it was found that CISC machines run slowly in order to support unnecessary features.

In particular, CISC machines had three main **limitations**:

CISC instruction set requires microcode

CISC ISA contains many different instruction types and each instruction requires a somewhat different implementation. Also, many CISC operations are complex with many separate steps. These instructions require complex decoding and sequencing, which can best be implemented on a generic microarchitecture using microcode.

Central bus organization

The implementation of atomic microcode operations is most easily carried out using a central System Bus, which becomes a bottleneck in CPU operation. Because only one value can be transferred at a time on the bus, microcode operations must be performed sequentially. Therefore, the machine instructions made up of microcode operations are executed sequentially and each machine instruction executes in multiple clock cycles.

Memory access

The complexity of the CISC operations means that instructions are of non-uniform length, from 1 byte to 16 or more bytes per instruction. As a result, the process of instruction fetch is made complex — multiple clock cycles are required to fetch one instruction. Worse still, determining the number of bytes to load requires decoding of the first byte.

Slide 4 RISC "Philosophy"

By the mid-1980s, based on research that revealed the limitations and inefficiency of CISC, computer architects, including John **Hennessy** at Stanford and David **Patterson** at Berkeley, proposed a new approach called **RISC** (Reduced Instruction Set Computer). The basic idea is to apply Amdahl's "Law" to instruction set architecture — speed up the operations accounting for most of run time and ignore any performance degradation to other instructions, so that the overall run time will be improved.

The RISC proposal was based on significant technological developments since the VAX was introduced in the 1970s. The price of RAM fell from $5000 / MB in 1975 to $5 / MB in 1990 (today RAM can be ordered online for about $0.0035 / MB). As compilers became more powerful and efficient with extensive optimization procedures, and as UNIX, C, and TCP/IP made practical portable code widespread, only specialists programming critical path tasks used assembly language. Therefore, choices made by compilers became more important than the convenience of the assembly programmer.

The principal research result on **CISC performance** can be summarized as two rules of thumb:

About 90% of run time is accounted for by about 10% of the instructions in the VAX ISA

About 90% of the VAX instruction set accounts for about 10% of program run time

Beginning in 1984 the RISC model was proposed. The idea is to speed up the operations that account for most of run time and ignore performance degradation to other instructions that may come from the enhancements. The RISC ISA reduces the CISC ISA, keeping only the most important instructions. This means that CISC instructions not included in RISC will be implemented as multiple RISC instructions. The goal is to build a simpler hardware implementation that will permit faster execution.

Slides 5 – 6 RISC Microprocessors

Simpler ISA

The RISC ISA defines fewer machine instructions and all instructions are defined to be of the same length. This makes the task of microarchitecture design much simpler.

Simpler hardware design

The simpler ISA permits a microarchitecture without microcode — all instructions can be implemented in a similar way. As a result there is no dedicated system bus and the CPU can process several instructions at the same time. As a result the CPU has a lower CPI and can be driven at a higher clock speed. One instruction completes its execution on almost every clock cycle.

High level program compiled to RISC

Because some CISC instructions are implemented a several RISC instructions, programs have a larger IC. There are more machine instructions after compilation than in CISC. But the longer machine programs run more quickly than the same high level program on CISC.

All processors today use RISC technology

Pure RISC CPUs include the IBM Power, SPARC, MIPS, ARM, and others

RISC technology for CISC language includes Intel x86 (Pentium, Centrino, Core, Xeon)

Explicitly parallel RISC includes Intel IA-64 (Itanium) and IBM mainframe CPUs

A comparison of CISC and RISC features is shown in the table:

	CISC	RISC
Instruction Types	300	50
Addressing Modes	15	5
Data Types	10	2
Procedure Handling	Automated and complex	Must be hand coded
Implementations	Complex	Simple
Memory Organization	Complex	Simple

Analysis of x86 and SPEC programs on the VAX and MIPS RISC CPU lead to a significant **speedup**:

$$S = \frac{T}{T'} = \frac{CPI^{CISC} \times IC^{CISC} \times \tau^{CISC}}{CPI^{RISC} \times IC^{RISC} \times \tau^{RISC}} = \frac{CPI^{CISC}}{CPI^{RISC}} \times \frac{IC^{CISC}}{IC^{RISC}} \times \frac{\tau^{CISC}}{\tau^{RISC}} = (\approx 6) \times \left(\approx \frac{1}{2} \right) \times \frac{\tau^{CISC}}{\tau^{RISC}} \approx 3 \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

On average the compiled binary of a C program to RISC is about twice as large (IC) as for CISC. But the CISC CPU requires about 6 times as many clock cycles to execute a typical instruction as a RISC CPU, so the speedup is at least 3, even before raising the clock rate.

Clock rates were more easily raised in RISC implementations because the simpler design requires fewer and smaller digital circuits. In general, digital circuits consume power on transitions between 0 and 1 values, so that the faster the clock rate the higher the bit transition rate and thus the higher the power consumption (the so-called **thermal bound**). As circuits get physically larger it becomes more difficult to remove heat from the center of the CPU package and so smaller circuits can be more easily operated at higher clock rates. Companies like Intel and IBM that specialize in the solid state physics of silicon have succeeded in making individual circuit features so small (around 10^{-5} millimeter) that other factors have become limitations for large circuits.

Designing a RISC ISA

In the rest of this chapter we review research results on CISC machines in order to make intelligent choices for a RISC ISA.

Slide 8 **Considerations for a RISC ISA**

A RISC ISA should be:

Simple — no instruction should require more steps than others

Complete — the ISA should be able to perform any desired computation

Orthogonal — there should be only one way to encode any given computation

Design of the ISA requires the following choices

Computation model — register-register or register-memory

Range and type of operations — which operations to include from all CISC possibilities

Operands — which data types and sizes to include from all CISC possibilities

Addressing modes — which types and sizes of displacement parameters to include

Branches — which types (conditional, unconditional, call/return) and offset (jump length)

These choices can be made by studying their relative importance in the run time of CISC machines that allow many possible combinations of these features.

Slide 9 Instruction Types

In order to choose the instruction (operation) types for a RISC ISA, Hennessey and Patterson chose 5 programs from the SPEC CINT benchmark suite (1992 version) and compiled them for the Intel x86 CISC ISA. By counting the number of times each instruction ran (IC_i) and the number of clock cycles required to run each instruction (CPI_i) they found the relative proportion of run time for each instruction ($F_i = [CPI_i \times IC_i] / [CPI \times IC]$). The results appear in the table:

Instruction	Relative Proportion of Total Run Time
Load	22%
Conditional branch	20%
Compare	16%
Store	12%
Add	8%
And	6%
Sub	5%
Move reg-reg	4%
Call	1%
Return	1%
Other	5%
Total	**100%**

The main result is that the first 10 instructions (out of about 300 defined by the ISA) account for 95% of run time. A design that runs these instructions quickly will speed up 95% of run time.

The first 10 instructions do not form a complete set — additional operations are required, such as Boolean operations. Also, the call and return instructions (2% of run time) are complex and difficult to implement, so they will be eliminated. Any reasonable set of 30 to 50 simple instructions from the x86 ISA (excluding register-memory ALU instructions and call/return) will produce an efficient RISC ISA.

Slides 10 – 11 Addressing Modes

Slide 10 shows a graph of the frequency of use of addressing modes in three SPEC programs compiled for the VAX ISA. The numbers represent the relative proportion of instructions in the 3 programs that use a particular addressing mode. The programs are TeX (phototypesetting), spice (electronic circuit emulation) and gcc (GNU compiler).

The table on slide 11 summarizes the results. Of the 12 addressing modes available in the VAX ISA, 3 modes account for 99% (TeX), 75% (spice) and 90% (gcc) of all accesses to operands.

A RISC ISA can be restricted to these 3 addressing modes — **register deferred**, **immediate**, **displacement**. A further simplification can be obtained by implementing register deferred as a special case of displacement: implement `mem[ reg ]` as `mem[ reg + 0 ]`

Slide 12 Instruction Length

In order to achieve an efficient fetch and execute procedure, instructions should be of uniform length. This enables completion of instruction fetch in 1 clock cycle. It also simplifies instruction decoding because the instruction length is known and instruction fields are in the same place for most instruction types. Most RISC machines for servers and workstations (including 64-bit ISAs) use 32-bit instructions. Some special purpose RISC machines use longer instructions, and the Intel Itanium and some IBM mainframes use 128-bit instructions.

When the ISA defines uniform 32-bit instructions it is obvious that no single field in the instruction can be 32 bits long. This means that address displacements, immediate values, the length of branch jumps will be smaller than 32-bit, and we must **choose the size of the field** that specifies these values.

Slides 13 – 14 Length of Immediate Operand

An immediate value is a constant (literal) hard coded into the instructions of a program. Integer values represented by an n-bit field range in size from -2^{n-1} to $+2^{n-1} - 1$. But most constants are not large integers. Slide 13 shows a graph of the relative frequency of the **number of bits required** to represent actual immediate values used in TeX, spice and gcc compiled for VAX.

From the table on slide 14 it can be seen that **allocating 16 bits** in the 32-bit instruction for immediate operands covers 70% (TeX), 78% (spice) and 80% (gcc) of immediate values used.

Slides 15 – 16 Displacement Length

The displacement addressing mode **`mem[reg + displacement]`** requires a displacement value coded as an immediate in the instruction. The displacement generally represents an offset from the base of a local data frame and so most are not large integers. Slide 15 shows a graph of the relative frequency of the **number of bits required** to represent actual immediate values used in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 16 it can be seen that **allocating 16 bits** in the 32-bit instruction for displacements covers 100% of values used in INT and FP programs.

Slides 17 – 18 Branch Instructions

Slide 17 shows a graph of the relative use of branch instructions in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 18 it can be seen that the call and return instructions are used in more than 10% of branch instructions, typically for implementing function calls in high level languages. However these instructions are complex and difficult to implement in RISC architecture. The call instruction must store the return address and register values on a local stack and then jump to a target address. The return must recover the return address, jump to that address, recover register values and clean up the stack. In a RISC machine it is preferable to implement call and return in a different way.

Slide 18 also shows that the unconditional branch is used in fewer than 6% of branch instructions. But this instruction is required for completeness of the instruction set and must be included in the RISC ISA. Including conditional and unconditional branch instructions covers about 90% of branches used in INT and FP programs.

Slides 19 – 20 Branch Offset

In a CPU, the program counter (PC) is the register that holds the address of the next instruction to be fetched. Branch instructions change the PC as **`PC ← PC + offset`** and require an offset value coded as an immediate in the instruction. Slide 19 shows a graph of the relative frequency of the **number of bits required** to represent actual offset values used in SPEC CINT and CFP programs compiled for VAX.

From the table on slide 20 it can be seen that **allocating 16 bits** in the 32-bit instruction for the offset covers 100% of values used in INT and FP programs.

Slide 21 Summary — RISC ISA By the Numbers

In this chapter we have seen research results on the actual usage at run time of the many complex features available in CISC ISA. Most of these features are rarely used. A RISC ISA is designed by including only the most widely used features from CISC and implementing the remaining features as combinations of the resulting reduced set. In summary, we expect a RISC ISA to include the following features:

Instruction Types

10 instructions account for 95% of run time in CISC machines. We may choose 30 – 50 of the most necessary and convenient instructions and operations for the RISC instruction set.

Addressing Modes

Three addressing modes — register, immediate and displacement account for over 75% of all operands accessed in CISC.

Instruction Length

The RISC ISA defines uniform 32-bit instructions.

Branch Instructions

Conditional and unconditional branch instructions account for over 75% of all branch instructions used in CISC.

Length of immediate values

It was seen that an immediate field of length 16-bits is adequate for over 70% of immediate operands and 100% of displacement and branch offset values.

An example of a RISC ISA is presented in the coming chapters.

Chapter 5

DLX Architecture — A Model RISC Processor

בפרק 4 ראינו תוצאות מחקר על מחשבי CISC ובחרהו מרכיבים ל- ISA בסוג RISC. בפרק זה נציג מודל למעבד RISC בשם DLX. המודל הזה מבוסס על מעבד מסחרי בשם MIPS 2000 ומפושט כדי להקל על לימדת הנושא. לא קיים מעבד DLX בסיליקון ולא כל הפרטים סגורים עד הסוף. לפירוט נוסף ניתן ללמוד ממעבדי MIPS.

Slides 2 – 3 DLX Architecture — General Features

The DLX uses a flat memory model (no segmentation) with a 32-bit address. The ISA defines 3 data types:

Integers (32-bit)

Single precision Floating Point (32-bit)

Double precision Floating Point (64 bits)

The CPU uses a register-register operation model — all ALU operands have source and destination in the registers.

There are 32 integer registers, each 32 bits wide. Integer registers are named R0, R1, ... , R31 and are addressed as 00000 to 11111 in the register address space. Reading register R0 always returns zero — Reg[R0] = 0 (constant). Writes to R0 are legal but have no effect. All other registers are identical (no special purpose registers).

There are 32 FP registers, each 32 bits wide. FP registers are named F0, F1, ... , F31. The DLX uses the IEEE 754 standard FP format. Accesses to double precision FP values use a register pair (even and odd).

The DLX uses three basic addressing modes — register, immediate and memory. The memory addressing mode (**displacement**) can implement **register deferred** (by specifying a zero displacement) and **absolute** (by using R0 that always returns 0).

Slides 4 – 7 Instruction Set

The complete instruction set is shown on slides 4 to 7.

Transfer instructions are shown on slide 4. A typical instruction is of the form

`LW R1, 30(R2)`

read as **Load Word** from the memory address **`30(R2)`** to register **`R1`**. The formal definition is

$$\mathbf{Reg[R1] \leftarrow_{32} Mem[30 + Reg[R2]]}$$

which indicates a 32-bit (4-byte) transfer from the memory address formed by adding 30 to the pointer value stored in R2 (and 3 additional bytes from the next 3 addresses).

The **Store Word** instruction has a similar syntax for the opposite direction.

The **Load Byte** instruction transfers one byte from the single memory location and performs a sign extension to a 32-bit representation. Bits in DLX are numbered from 0 = most significant bit (MSB), so that $\mathbf{Mem[30 + Reg[R2]]_0}$ refers to the high-order bit in the byte stored at that address (the sign bit). The notation $\mathbf{(byte_0)^{24}}$ **`## byte`** means that the 32-bit value is formed from 24 copies of the sign bit followed by the 8 bits of the memory byte.

Notice that **Store Byte** copies the 8 low-order bits from the register to a memory location and that **Load Byte Unsigned** sign extends using leading 0 instead of the sign bit.

The instruction LF is similar to LW but for FP values and registers. The instructions MOVFP2I and MOVI2FP transfer values between integer and FP registers but do not convert any values.

The standard **integer ALU instructions** are defined on slide 5. These include the four basic arithmetic instructions +, –, ×, ÷, the Boolean operations AND, OR, XOR, and six compare operations. The operations ADD, SUB, AND, OR, XOR have register-register versions and versions that operate on a register value and an immediate value. There is also a Load High instruction that writes a 16-bit immediate value to the upper 16-bits of a register. This permits putting the value 0x11223344 into register R2 with the sequence

```
LHI R2, #1122
ADDI R2, R0, #3344
```

The **standard FPU instructions** are defined on slide 6. These include the four basic arithmetic instructions +, –, ×, ÷, and six compare operations for single and double precision. The IEEE 754 standard is presented in chapter 7.

The **control (branch) instructions** are defined on slide 7. The first four instructions are unconditional branches:

J performs an absolute jump.

JAL write the PC to R31 before the jump.

JR jumps to the address stored in a register. R31 can then be used as a return instruction.

JALR is similar to **JAL** but permits a choice of storage register.

There are two conditional branch instructions:

BEQZ performs a conditional jump if the value of a register is 0.

BNEZ performs a conditional jump if the value of a register is not 0.

The **TRAP** instruction performs a software interrupt. The details are not worked out for the DLX but the instruction is well-defined for the MIPS family.

To get a feel for programming in the DLX assembly language we consider an implementation of a simple C program:

```
for ( i = 0 ; i < 256 ; i++)
   a[i] = a[i] + b[i] – c[i] + d[i]
}
```

The arrays contain 256 elements = 256 × 4 = 1024 bytes = 0x400 bytes (hexadecimal notation).

For this example we suppose that the compiler places the 4 arrays at addresses:

```
a[] = 000 – 3FF
b[] = 400 – 7FF
c[] = 800 – BFF
d[] = C00 – FFF
```

The assembly language code is:

```
ADDI R1, R0, #0x400     ; Since R0 = 0 this writes the immediate value 400 to R1
LW   R2, -4(R1)         ; load word from a[] to R2  ( 400 – 4 = 3FC = top of a[] )
LW   R3, 3FC(R1)        ; load word from b[] to R3  (400 + 3FC = 7FC)
ADD  R4, R2, R3         ; add the values in R2 and R3 to R4
LW   R2, 7FC(R1)        ; load word from c[] to R2  (400 + 7FC = BFC)
SUB  R4, R4, R2         ; subtract the value of R2 from R4
LW   R2, BFC(R1)        ; load word from d[] to R2  (400 + BFC = FFC)
ADD  R4, R4, R2         ; add the value of R2 to R4
SW   -4(R1), R4         ; store sum from R4 to a[]
SUBI R1, R1, #4         ; i--  (the index i points to 4-byte memory locations)
BNEZ R1, -0x28          ; if R1 <> 0 jump 10 back instructionsNotice that the
```

assembly code works from the top of the array to the bottom, an optimized implementation of the C program that reduces branch instructions and ALU operations.

The BNEZ instruction jumps back in the loop as long as R1 does not reach 0. While BNEZ executes the PC already points to the next instruction below it. An infinite loop is written as

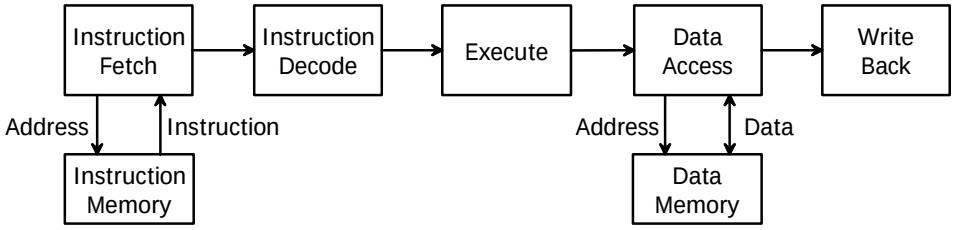
```
J –0x04        ; subtract 4 from the address of the instruction below J.
```

The length of the jump is 10 instructions and all instructions are 4 bytes (32-bits) long. The number 40 is hex is 0x28.

Slide 9 Implementation

The general approach to implementation is division into 5 independent stages. This hardware organization is based on an industrial assembly line with uniform operations. There is no central system bus. There are separate memory caches for instructions and data.

High level design



Instructions move through 5 stages from left to right. The instruction completes in each stage after 1 clock cycle. The first two stages **FETCH** and **DECODE** are identical for all instructions. The last three stages operate differently according to the instruction:

EXECUTE performs ALU instructions and address calculations

MEMORY ACCESS executes Load and Store instructions

WRITE BACK performs register updates for Load and ALU instructions

Slide 10 RISC Performance

Analysis of x86 and SPEC programs on the VAX and MIPS 2000 CPU lead to a significant **speedup**:

$$S = \frac{T}{T'} = \frac{CPI^{CISC} \times IC^{CISC} \times \tau^{CISC}}{CPI^{RISC} \times IC^{RISC} \times \tau^{RISC}} = \frac{CPI^{CISC}}{CPI^{RISC}} \times \frac{IC^{CISC}}{IC^{RISC}} \times \frac{\tau^{CISC}}{\tau^{RISC}} = (\approx 6) \times \left(\approx \frac{1}{2} \right) \times \frac{\tau^{CISC}}{\tau^{RISC}} \approx 3 \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

On average the compilation of a C program to RISC is about twice as large (IC) as for CISC. But the CISC CPU requires about 6 times as many clock cycles to execute a typical instruction as a RISC CPU, so the speedup is at least 3, even before raising the clock rate.

Slides 11 – 14 Instruction Formats

All DLX instructions are 32-bits with bits numbered 0 (left) to 31 (right). There are three instruction formats:

Type	0-5	6-10	11-15	16-31	
	6	5	5	5	11
R	opcode	rs1	rs2	rd	function
I	opcode	rs	rd	immediate	
J	opcode	offset			

In all instructions bits 0 – 5 specify the opcode (operation code) that determines the instruction.

The **J-type** specifies the unconditional branch instructions Jump and Jump And Link. The branch offset field is 26 bits allowing long jumps.

The **R-type** specifies register-register ALU instructions with two source registers (rs1, rs2), a destination register (rd) and a function field that expands the ALU types. The typical operation is of the form **`rd ← rs1 function rs2`**.

The **I-type** is used for all other instructions with one source register (rs), destination register (rd) and a 16-bit immediate value. The typical instructions are:

Loads

```
rd ← imm(rs)
```

Stores

```
imm(rs) ← rd
```

ALU operations with immediate operand

```
rd ← rs op immediate
```

Conditional branch instructions

```
if rs eq/ne 0 then PC ← PC + imm  (rd unused)
```

Jump register

```
PC ← rs
```

Jump and link register

```
rd ← PC
PC ← PC + immediate
```

Implementation Details

We first describe the implementation of instructions in the 5 stages of the CPU and then give detailed examples of execution in the detailed subsystems of the CPU microarchitecture.

Slide 16 **Execution Stages by Instruction Type**

The actions of each instruction type in each CPU stage are shown in the table:

Stage	ALU	Store	Load	Branch
Instruction Fetch (**IF**)	Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory
Instruction Decode (**ID**)	Decode operation and operands	Decode operation and operands	Decode operation and operands	Decode operation and operands
Execute (**EX**)	Calculate ALU operation	Calculate memory address	Calculate memory address	Calculate branch condition Calculate branch address
Memory (**MEM**)		Update PC	Load data from memory	Update PC
Writeback (**WB**)	Write result to register Update PC		Write loaded data to register Update PC	

We see that all instructions undergo the same actions in the IF and ID stages.

The EX stage uses the ALU in different ways depending on the instruction — it may perform basic ALU operations or calculate address locations for memory and branch instructions. The MEM stage performs memory access for load and store operations. The WB stage updates registers are ALU and load operations.

Slide 17 Temporary Registers for Implementation

In order to carry out instruction execution certain temporary registers are required:

IR — Instruction Register

Register in the IF stage that holds the fetched instruction during execution

PC — Program Counter

Register in the IF stage that holds the memory address of the next instruction

NPC — Next Program Counter

Register in the IF stage that holds a temporary update of the PC

A, B, I

Registers in the ID stage that buffer operand for use in the ALU

ALU_{out} — ALU output

Register in the EX stage that holds the result of an ALU operation

Cond — Condition Flag

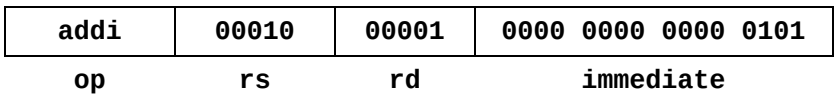
Register in the EX stage that holds the result of the test for conditional branch

LMD — Load Memory Data

Register in the MEM stage that holds data loaded from memory

Slide 18 Example Type-I ALU Instruction

We begin with a simple type-I ALU instruction. The analysis of the other instructions types will follow this pattern. Line 1 presents the **assembly syntax** for the instruction, line shows 2 the **formal description** of the operation and line 3 shows the **instruction encoding**. The encoding shows that the opcode is addi, rs = R2 (00010 binary), rd = R1 and the immediate = 5.

1	Instruction	`addi R1, R2, #5`
2	Operation	`Reg[R1] ← Reg[R2] + 5`
3	Encoding	
4	Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
5	Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R1] */ $I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$
6	Hardware Stage 3	$ALU_{out} \leftarrow A + I$
7	Hardware Stage 4	
8	Hardware Stage 5	$Reg[IR_{11\text{-}15}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + I */ `PC ← NPC`

The execution lines of the table are:

4	Hardware Stage 1	**IR ← Mem[PC]** **NPC ← PC + 4**
5	Hardware Stage 2	**A ← Reg[IR_{6-10}]** /* A ← Reg[R2] */ **B ← Reg[IR_{11-15}]** /* B ← Reg[R1] */ **I ← $(IR_{16})^{16}$ ## IR_{16-31}**
6	Hardware Stage 3	**ALU_{out} ← A + I**
7	Hardware Stage 4	
8	Hardware Stage 5	**Reg[IR_{11-15}] ← ALU_{out}** /* Reg[R1] ← A + I */ **PC ← NPC**

Line 4 describes two actions in stage 1 — the **instruction fetch** stage:

The IF stage reads the address of the next instruction from PC, accesses this address in the instruction cache and stores the 4-byte instruction in IR.

The IF stage adds 4 to the PC and stores the result in NPC.

Line 5 describes three actions in stage 2 — the **instruction decode** stage:

The ID stage reads the 5-bit number of rs from the instruction, accesses this register and copies the 4-byte value to A.

The ID stage reads the 5-bit number of rd from the instruction, accesses this register and copies the 4-byte value to B.

The ID stage reads the 16-bit immediate from the instruction, sign-extends to 32 bits and copies the 4-byte value to I.

Line 6 describes the action in stage 3 — the **execution stage**:

The EX stage adds the values of A and I and writes the 4-byte value to ALU_{out}. Notice that the value in B is ignored.

Line 7 shows that no action is taken in stage 4 — the **memory** stage. This stage can be skipped by instructions of this type.

Line 8 describes the two actions in stage 5 — the **writeback** stage:

The WB stage reads the 5-bit number of rd from the instruction, and writes the value of ALU_{out} to this register.

The WB stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 19 Example Type-R ALU Instruction

This instruction is similar to the type-I example except that is uses R3 instead of an immediate. The execution stages are also identical, except in the execution stage:

The EX stage adds the values of A and B and writes the 4-byte value to ALU_{out}. This time the value in I is ignored.

Type-I ALU uses **I** and type-R ALU uses **B**. It is simpler to prepare both **I** and **B** in stage ID than to add extra hardware that makes the choice. The extra register read is unnecessary but harmless.

Slide 20 Example Type-I Store Instruction

This instruction is similar to the type-I ALU instruction except that the sum is used as a memory address instead of being written to a register. The actions in stages 1 – 3 are identical to those of the type-I ALU instruction. But here

The MEM stage writes the value of B to memory using the 4-byte address value from ALU_{out}.

The MEM stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 21 Example Type-I Load Instruction

This instruction is similar to the type-I load instruction except that direction of the transfer is reversed:

The MEM stage reads from memory the 4-byte value at the address from ALU_{out} and copies the operand to LMD.

The WB stage copies the loaded operand from LMD to the register named in the instruction in rd.

The WB stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 22 Example Type-I Conditional Branch Instruction

The conditional branch operates differently from the previous instructions. As always, the actions in stage IF and ID are the same as for every instruction. Then

The EX stage adds the values of NPC and I and writes the 4-byte value to ALU_{out}. This value is the address of the branch target instruction if the branch condition is valid.

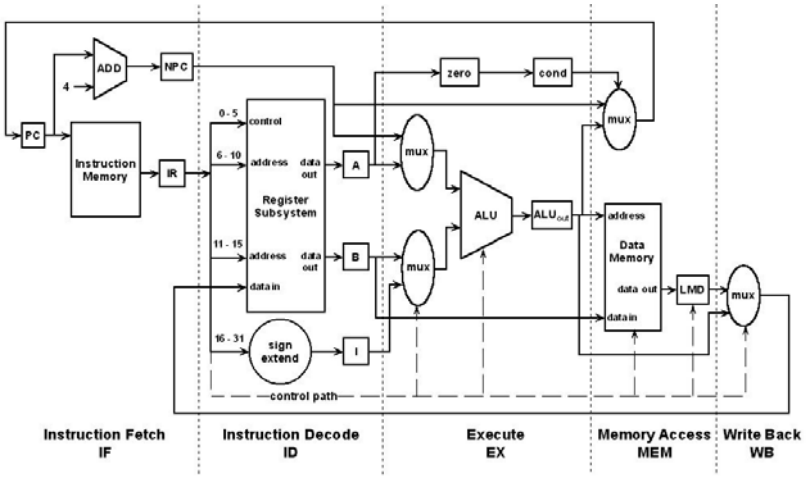
The EX stage compares the value in A with 0 and writes a 1-bit flag to the condition register cond.

The MEM stage updates the PC register. If the condition flag cond is 1 then it copies ALU_{out} to PC (this is the branch taken). If cond = 0 then it copies NPC to PC (this is the branch not taken).

Slides 23 – 44 DLX Hardware Drawing — Version 1

The hardware drawing of DLX version 1 should be read as a graphic representation of the formal description given above.

The 5 stages are separated by dotted lines. The solid lines are data paths among basic elements (memory, registers, ALU) and temporary registers — they implement the actions described on slides 18 – 22.



Stages EX, MEM and WB contain **multiplexors** labeled as **mux**. These multiplexors implement the choices described in the description of each stage — the control path (dashed line) provides a signal that chooses one input from the left and sends it to the output at the right.

The EX stage has two multiplexors. The lower multiplexor has inputs I and B — it selects I for type-I instructions and selects B for type-R ALU instructions. The upper multiplexor selects input A for all instruction types except conditional branch when it selects NPC.

The multiplexor in MEM selects NPC, except for a conditional branch taken when it selects the calculated target address from ALU_{out}.

The multiplexor in WB selects ALU_{out}, except for load instructions when it selects LMD.

Slides 24 to 27 show the stage-by-stage execution of the Type-I ALU Instruction.

Slide 24 shows the actions performed in stage 1 — IF and the information flow

Slide 25 shows the actions performed in stage 2 — ID and the information flow

Slide 26 shows the actions performed in stage 3 — EX and the information flow

Slide 27 shows the actions performed in stage 5 — WB and the execution completion

Slides 28 to 31 show the stage-by-stage execution of the Type-R ALU Instruction.

Slides 32 to 35 show the stage-by-stage execution of the Type-I Store Instruction.

Slides 36 to 40 show the stage-by-stage execution of the Type-I Load Instruction.

Slides 41 to 44 show the stage-by-stage execution of the Type-I Branch Instruction.

Slide 45 Performance

As seen in the detailed analysis of instruction execution, all instructions DLX version 1 can execute in 4 clock cycles, except for loads, which require 5 clock cycles.

Hennessey and Patterson report that compilation of SPEC CINT programs for the DLX instruction set can be sorted to produce the following table for IC_i / IC along with CPI_i:

Instruction Type i	IC_i / IC	CPI_i
ALU	40%	4
Load	25%	5
Store	15%	4
Branch	20%	4

The average CPI can be calculated as

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.40 + 5 \times 0.25 + 4 \times 0.15 + 4 \times 0.25 = 4.25$$

The goal of future versions of the DLX is to reduce this value of CPI to as close to 1 as possible.

Chapter 6

Speeding Up DLX

בפרק 5 הצגנו את מעבד ה- DLX כמודל למעבד RISC וביצענו ניתוח של ה- microarchitecture שלו. גרסה 1 של ה- DLX פועלת על פקודה אחת בחמישה שלבי ביצוע ומסיים את הטיפול בפקודה הייצוגית ב- 4.25 מחזורי שעון. בפרק הזה נציג גרסה משופרת שמסוגלת לסיים ביצוע פקודה כמעט על כל מחזור שעון. הכלי הבסיסי לשיפור נקרא pipeline (צנרת) והוא מאפשר חפיפה של פקודות. כאשר פקודה מסיימת בחטיבת IF ועוברת לחטיבת ID פקודה חדשה נכנסת לחטיבת IF וכך הלאה. ברוב המקרים המעבד פועל על 5 פקודות, בשלבי ביצוע שונים, בכל עת. כדי לאפשר לימוד מסודר נכניס שיפור אחד לפי הצורך ואז נדון בשיפור נוסף. כך נעבור מגרסה 1 לגרסה 2, 3, 4, 5, 6A, 6B, 6C.

Slides 2 – 3 **DLX Execution Stages — Version 1**

Slide 2 reviews the executions stages of DLXv1:

Clock Cycle 1

I_1 (instruction number 1) enters Instruction Fetch (IF)

Clock Cycle2

I_1 moves to Instruction Decode (ID)
Instruction Fetch (IF) holds its state fixed

Clock Cycle3

I_1 moves to Execute (EX)
Instruction Fetch (IF) holds its state fixed
Instruction Decode (ID) holds its state fixed

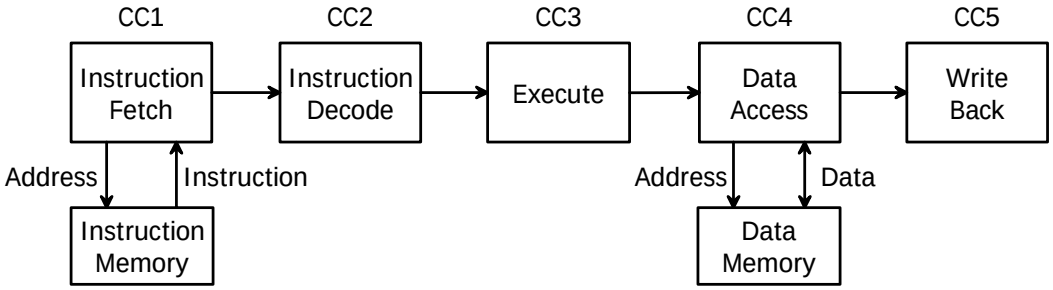
Clock Cycle4

I_1 moves to Memory Access (MEM)
Instruction Fetch (IF) holds its state fixed
Instruction Decode (ID) holds its state fixed
Execute (EX) holds its state fixed

Clock Cycle5

I_1 performs Write Back (WB) using instruction (IR) stored in IF stage
PC updated and stages IF, ID, EX, MEM are reset

There is no central system bus and instructions move from execution stage to execution stage. This description suggests the main possibility of improvement. DLXv2 is based on an industrial **assembly line**, which permits pipelining — in each stage new work begins when old work passes to the next stage.



Slides 4 – 7 DLX — Version 2

In DLXv2 each stage begins new work when the old work is passed to the next stage. The "work" is formalized as **Instruction State** — the 32-bit instruction and all intermediate results from earlier stages required to complete execution.

The operations on the first 5 instructions (in this ideal case) are:

CC1	I_1 enters Instruction Fetch (IF)
CC2	I_1 and its execution state move to Instruction Decode (ID) I_2 enters Instruction Fetch (IF)
CC3	I_1 and its execution state move to Execute (EX) I_2 and its execution state move to Instruction Decode (ID) I_3 enters Instruction Fetch (IF)
CC4	I_1 and its execution state move to Memory Access (MEM) I_2 and its execution state move to Execute (EX) I_3 and its execution state move to Instruction Decode (ID) I_4 enters Instruction Fetch (IF)
CC5	I_1 moves to Write Back (WB) I_2 and its execution state move to Memory Access (MEM) I_3 and its execution state move to Execute (EX) I_4 and its execution state move to Instruction Decode (ID) I_5 enters Instruction Fetch (IF)

The steps in execution can be visualized in the **Processor View** table. Five columns show the instruction content in one of the 5 executions stages, and the rows show the state in a given clock cycle.

clock cycle ↓ stage →

	IF	**ID**	**EX**	**MEM**	**WB**
1	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1
6	I_6	I_5	I_4	I_3	I_2
7	I_7	I_6	I_5	I_4	I_3
8	I_8	I_7	I_6	I_5	I_4

After CC4, 5 instructions are being processed at one time and each instruction is in a different stage of execution.

A different presentation is the **Instruction View** showing the progress of each instruction in a given clock cycle.

instruction ↓ clock cycle →

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	
I_4				IF	ID	EX	MEM	WB
I_5					IF	ID	EX	MEM
I_6						IF	ID	EX
I_7							IF	ID
I_8								IF

The **performance advantage in DLXv2** is intuitively obvious — the CPU is working on multiple instructions at the same time and should complete the program in a shorter time. This is called the **ideal case** because it does not take account of **dependencies** between instructions that may cause the pipeline to **wait** for results of previous instructions before continuing with the next instruction.

The formal calculation of speedup follows from the observation that I_1 finishes after N=5 clock cycles, I_2 finishes after N=6 clock cycles, I_3 finishes after N=7 clock cycles and so on. In general, a program with **IC instructions** will complete in N=IC+4 clock cycles. The average CPI in the ideal case is then

$$CPI = \frac{\text{clock cycles}}{\text{finished instructions}} = \frac{IC+4}{IC} = 1 + \frac{4}{IC} \xrightarrow[IC>>4]{} 1$$

This means that on average, one instruction completes on every clock cycle.

Slides 8 – 10 **Pipelining — Functional Requirements**

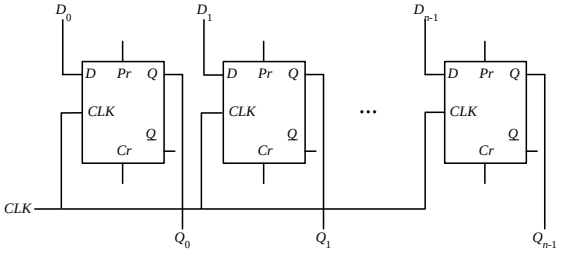
In DLXv2 each stage receives a new instruction on every clock cycle. The 5 stages cannot all hold the intermediate results for 5 instructions. So, each stage must transfer the **Instruction State** (the 32-bit instruction and all intermediate results) to the next stage after execution in that stage. For example,

IF stage
- Loads an instruction to **IR**
- Finds **NPC** for the next instruction
- Passes **IR** and **NPC** (intermediate results) to the **ID** stage

ID stage
- Stores the received **IR** and **NPC** for the incoming instruction
- Decodes **IR** to **A**, **B**, and **I**
- Passes **IR**, **NPC**, **A**, **B**, and **I** to the **EX** stage

In order to manage the temporary storage of these intermediate results, a **stage buffer** is placed between each pair of 5 pipeline stages.

Stage buffers are collections of **D-flip/flops** (also called **edge-triggered latches**) that perform **synchronous transfer**.

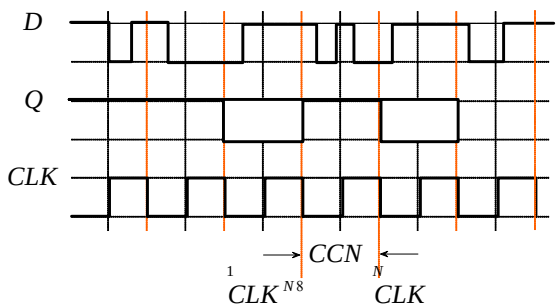


A buffer contains n D-flip/flops to store n bits:

Input D — the output of some digital system

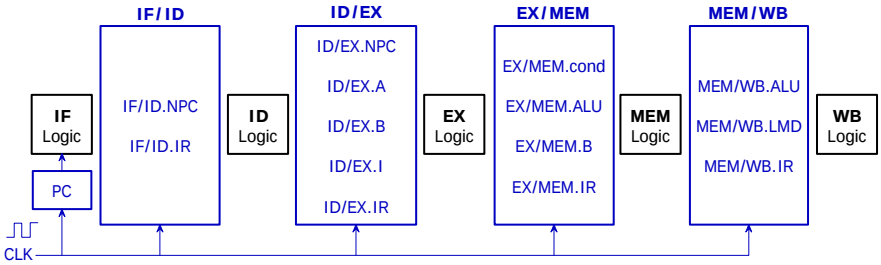
Output Q — samples D and changes only on the falling edge (1-to-0 transition) of the signal **CLK**

CLK (called a **Clock** or **Trigger**) is the same for all flip/flops — the n transitions are synchronized



Clock Cycle N defined as follows:
- CC N begins on CLK^{N-1}
  - Changes on Input D have no effect on the latch value or Output Q during the CC
- CC N ends on CLK^{N}
  - Latch samples Input D on falling edge of CLK
  - Stores instantaneous value of input D in latch
  - Forwards stored value to Output Q

DLXv2 redrawn with emphasis on the stage buffers appears as



A single system clock **CLK** is fed to all buffers, which synchronizes the transitions in the stages. In this version, the **PC** (address of next instruction) must also be transferred synchronously.

The logic elements in the 5 execution stages are built from two types of digital circuit

Combinational logic — output = function(present input)

Asynchronous memory — output = function(present input, past input)

The 4 stage buffers and the PC are built from another type of digital circuit

Synchronous sequential logic — output = function(present input, past input, external clock)

This arrangement insures **Synchronous Transfer**. When a clock cycle begins (falling edge of CLK) the synchronous stage buffers provide new inputs to the combinational logic and asynchronous memory in the pipeline stages. During the clock cycle the output of the stage logic changes asynchronously, depending on the propagation delays in each circuit. The period of the CLK signal (the period τ of the clock cycle) is chosen to insure that all of these asynchronous changes become stable before the end of the cycle. When the clock cycle ends (falling edge of CLK) the synchronous stage buffers sample their inputs (the outputs of the asynchronous circuits of the pipeline stage logic) store these values and transfer them to the next stage.

The stage buffers are named for the pipeline stages they buffer, using the dot-notation for members of a C structure:

IF/ID buffers NPC and IR from the IF stage for the ID stage.

ID/EX buffers the NPC and IR received from IF and the A, B, I results of stage ID.

EX/MEM buffers the IR and B results received from ID and the ALU and cond results of EX.

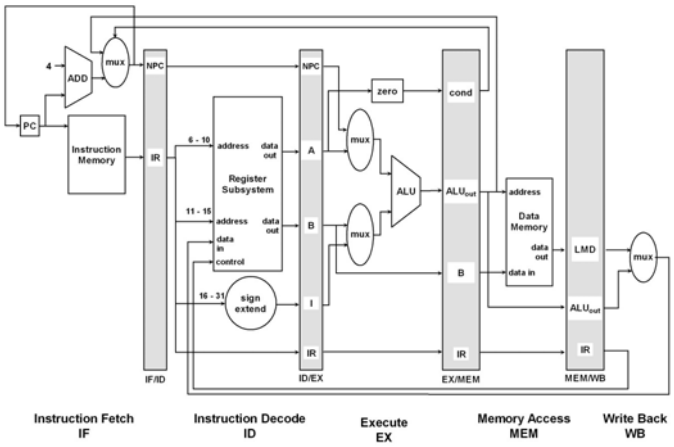
MEM/WB buffers the IR and ALU results received from EX and the LMD result of MEM.

Slides 11 – 13 DLX Drawing — version 2

The microarchitecture drawing of DLXv2 is shown on slide 11.

The stage buffers are shown in gray.

Notice that the multiplexor that chooses the next instruction address has been moved from the MEM stage to the IF stage. It is necessary to permit fetch of the next instruction on each clock cycle.



This version does not correctly handle branch instructions. That problem will be solved in DLXv6 (slides 45 – 48).

A formal specification of DLXv2 appears on slides 12 – 13.

Slide 14 Instruction Transfer Timing

As an example of synchronous transfer we consider the transfer of instructions across the pipeline. We write that a stage buffer "sees" an input while the asynchronous changes stabilize but the input has not been sampled or stored in the latch.

Trigger	Event	Transfer	Inputs
CLK 0	CC 1 begins	Memory ← PC(I_1)	IF/ID.IR "sees" Mem[PC(I_1)]
CLK 1	CC 2 begins	IF/ID.IR ← Mem[PC(I_1)] Memory ← PC(I_2)	ID/EX.IR "sees" Mem[PC(I_1)] IF/ID.IR "sees" Mem[PC(I_2)]
CLK 2	CC 3 begins	ID/EX.IR ← Mem[PC(I_1)] IF/ID.IR ← Mem[PC(I_2)] Memory ← PC(I_3)	EX/MEM.IR "sees" Mem[PC(I_1)] ID/EX.IR "sees" Mem[PC(I_2)] IF/ID.IR "sees" Mem[PC(I_3)]
CLK 3	CC 4 begins	EX/MEM.IR ← Mem[PC(I_1)] ...	MEM/WB.IR "sees" Mem[PC(I_1)] ...
CLK 4	CC 5 begins	MEM/WB.IR ← Mem[PC(I_1)]	Mem[PC(I_1)] controls Write Back

On CLK 0 clock cycle CC1 begins and a new value of PC for I_1 is presented to the IF logic. This value is used to read instruction I_1 from memory. During CC1 (between CLK 0 and CLK 1) the outputs of the memory circuits stabilize. Register IF/ID.IR "sees" these fluctuating signals during CC1 — when CLK 1 occurs these values have stabilized and are sampled and stored by the register latch. Following CLK 1 (in CC2) instruction I_1 is presented to the ID stage in the same way. At the end of CC2 (CLK 2) instruction I_1 is sampled and stored by ID/EX.IR.

In this way, instruction I_1 is stored in:

IF/ID.IR at the end of CC1
ID/EX.IR at the end of CC2
EX/MEM.IR at the end of CC3
MEM/WB.IR at the end of CC4

and can be used in the WB stage during CC5.

Slides 15 – 19 Simple 5-Instruction Program for DLX

Slide 16 shows the progress of simple 5-line program (slide 15). To avoid complications, this program contains no dependencies between lines (for that reason the program is trivial and accomplishes nothing).

Each column shows the actions performed in one pipeline stage in subsequent clock cycles. For example, column IF shows:

Fetch of I_1 from PC = 00 and preparation of NPC = 04 in CC1
Fetch of I_2 from PC = 04 and preparation of NPC = 08 in CC2
Fetch of I_3 from PC = 08 in and preparation of NPC = 0C in CC3
and so on.

Columns EX, MEM and WB show the specific actions taken for the different instruction types.

As an example of synchronous transfer (slide 17) we see that on:

CLK0 Memory ← PC =00 so that F/ID.IR "sees" Mem[00] and IF/ID.NPC "sees" 04 as inputs
CLK1 Memory ← PC =04 so that F/ID.IR "sees" Mem[04] and IF/ID.NPC "sees" 08 as inputs

Slide 18 shows the values "seen" by each stage buffer just before CLK4 ends CC4.
Slide 19 shows the values "seen" by each stage buffer just after CLK4 begins CC5.

Slides 21 – 22 Instruction Dependencies: Definitions

Up to now we have ignored dependencies between instructions. An **instruction dependency** occurs when the result of one instruction is needed to execute a later instruction. Naturally, dependencies are what give meaning to programs.

Hazard

A hazard exists when the processor runs smoothly but provides the wrong answers

Pipeline hazard

Several instructions are in various stages of execution in the CPU pipeline. If the pipeline uses a resource value before it is correctly updated by an earlier instruction, this is a pipeline hazard. This condition causes an error.

Example

PC ← NPC on each clock cycle
A branch instruction requires PC ← NPC + I
The correct evaluation of NPC + I is not available on the next clock cycle
The CPU fetches the wrong instruction

Hazard Types

Structural Hazard — caused by a conflict in the pipeline over access to a resource

Data Hazard — an instruction result is not ready when it is needed

Control Hazard — the branch address is not ready when it is needed

In order to prevent processing errors at run time, hazard conditions must be stopped. The simplest way to prevent the hazard is to pause the pipeline and wait for the required resource to become available. This is called a **WAIT STATE** or a **PIPELINE STALL**.

Stalls degrade processor performance because they add **stall clock cycles** to the instruction execution. This is formalized by considering the effect on CPI

$$CPI = \frac{\text{processing clock cycles (ideal) + stalled clock cycles}}{\text{completed instructions}}$$

$$= \frac{N^{ideal} + N^{stall}}{IC} = CPI^{ideal} + CPI^{stall} \xrightarrow[IC \to \text{ large on DLX}]{} 1 + CPI^{stall}$$

Similarly we characterize the performance degradation as

$$\text{performance degradation} = 1 - \frac{CPI^{ideal}}{CPI^{ideal} + CPI^{stall}} = \frac{CPI^{stall}}{1 + CPI^{stall}}$$

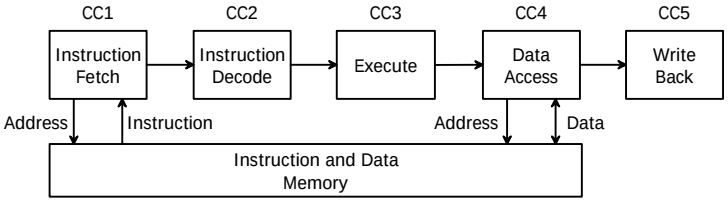
To avoid this degradation the next step is to eliminate the cause of the stall by analyzing the cause of the stalls and improving the implementation. This is the main professional activity of hardware architects.

Slides 23 – 25 Structural Hazards

A structural hazard is a conflict over access to resources. There are no structural hazards in DLXv2 but we must understand them to avoid introducing them while attempting to improve other aspects of the microarchitecture.

A typical structural hazard is the **unified cache hazard**. The DLX has a **split cache**: instructions are fetched from Instruction Memory (cache) in the IF stage and data is accessed in Data Memory (cache) in the MEM stage.

A pipeline with a unified cache has a single memory that is accessed from the IF stage and from the MEM stage:



Since instructions and data are in the same memory device, it is not possible to access data and fetch instruction on same clock cycle. This requires that instruction fetch waits (stalls) for 1 clock cycle on every data memory access (loads and stores). The implementation of the stall is seen in the processor view execution diagram as:

	IF	ID	EX	MEM	WB
CC1	I_1				
CC2	LW	I_1			
CC3	I_2	LW	I_1		
CC4	I_3	I_2	LW	I_1	
CC5	ϕ	I_3	I_2	LW	I_1
CC6	I_4	ϕ	I_3	I_2	LW
CC7		I_4	ϕ	I_3	I_2
CC8			I_4	ϕ	I_3
CC9				I_4	ϕ
					I_4

The Load Word (LW) instruction is fetched in CC2. The instructions proceed without a problem until CC5 when LW enters MEM and requires memory access. The data access of LW blocks the IF stage from accessing a new instruction. No instruction is fetched on CC5, symbolized by ϕ (null). There is no instruction to forward to ID on CC6, and instead IF forwards ϕ to represent **No Operation** (NOP). The NOP is also called a bubble. The NOP is forwarded to EX on CC7 and so on. No instruction completes execution on CC9.

To calculate the effect of this stall on CPI we expand the definition of CPI^{stall} as the **number of stall cycles per stall** multiplied by the **number of stalls per instruction**. There may be various **types of stall** so we must sum the stall cycles caused by different stall types i. There may also be various **instructions that cause stalls** so we expand stalls per instruction as **stalls of type i per instruction j** multiplied by the number of **instructions of type j per instruction** and sum over j.

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

$$= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instructions}}$$

$$= \sum_{i=\text{type}} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \left(\frac{\text{stalls}}{\text{instructions}} \right)_i$$

$$= \sum_{i,j} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \frac{\text{stalls of type i}}{\text{instructions of type j}} \times \frac{\text{instructions of type j}}{\text{instructions}}$$

← This is the most general formula for CPI^{stall}.

$$= \sum_{i} \left(\frac{\text{stall cycles}}{\text{data stall}} \right)_i \times \left(\frac{\text{data stall}}{\text{instructions}} \right)_i \times \frac{IC_i}{IC}$$

← In the case of unified cache, there is only one stall type.

The factors in the expression for the unified cache stall are known:

There is 1 stall cycle for every stall of this type
There is 1 stall for every data access (load or store)
Load instructions are 25% of all instructions and store instructions are 15% of all instructions

Plugging in these values gives

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

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

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

$$= 0.40 \frac{\text{stall cycles}}{\text{instruction}} \Rightarrow CPI = CPI^{ideal} + CPI^{stall} = 1.40$$

Most modern CPUs use split cache to avoid this stall. The performance degradation associated with $CPI^{stall} = 0.4$ is

$$\text{degradation} = \frac{CPI^{stall}}{1 + CPI^{stall}} = \frac{0.4}{1.4} = 0.29 = 29\%$$

Slide 26 Data Hazards

A data hazard occurs when an instruction result is not ready when it is needed. If the execution continues without correction then operations are performed in the wrong order. The types of data hazard are classified and named for the correct order of operations.

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

RAW hazards

A DLX register is updated in stage WB. The next instruction may attempt to read the same register in stage ID before it is updated. This is a possible hazard that must be avoided.

WAW hazards cannot occur

The DLX writes to memory and registers in the order of the instructions. Memory is always updated in MEM and registers are always updated in WB. Since all updates are performed in the order of execution, I_2 cannot perform either WB or MEM before I_1 performs WB or MEM. There is no possible error in the order of writing.

WAR hazards cannot occur

Loads are performed in MEM and registers are read in ID. A store is performed in MEM and registers updated in WB. Therefore I_2 cannot perform a register update in WB or a memory write in MEM before I_1 performs its register read in ID or a memory read in MEM.

Although WAW and WAR hazards are not possible in DLX, we must be sure not to create them while trying to improve the pipeline.

As an example of RAW hazards we consider the program

```
I1  ADD R1,R2,R3        I1 has R1 as destination
I2  SUB R4,R5,R1
I3  AND R6,R7,R1        I2 – I4 have R1 as source
I4  OR  R8,R9,R1
```

If the program runs without correction

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

I_1 updates **R1** in **WB** during **CC5**

I_2 reads old **R1** in **ID** during **CC3**

I_3 reads old **R1** in **ID** during **CC4**

I_4 reads **R1** in **ID** during **CC5**

I_2 and I_3 read the incorrect value of R1 in CC3 and CC4. I_4 reads the correct updated value of R1 in CC5. This is because the update of R1 occurs at the beginning of CC5 and the final read (sample and latch) of R1 for I_4 in ID occurs at the end of CC5. See slide 29 for a detailed analysis.

To prevent the register-register RAW hazard, **DLXv2 must stall the pipeline** for 2 clock cycles. This is seen in the processor view execution table

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	AND	SUB	ϕ	ADD	
CC5	AND	SUB	ϕ	ϕ	ADD
CC6	OR	AND	SUB	ϕ	ϕ
CC7		OR	AND	SUB	ϕ
CC8			OR	AND	SUB
				OR	AND
					OR

The DLX control system must be able to identify all hazards and insert stall cycles when necessary.

Wait states during CC3 and CC4

ID/EX freezes internal state on **SUB**
IF/ID freezes internal state on **AND**
ID performs **NOP** (no operation) to avoid reading old value of **R1**
ID/EX passes ϕ (**NOP**) to **EX**

Continuation — no hazard in CC5

WB performed at start of clock cycle
SUB result locked at end of clock cycle

The pipeline stall in instruction view is

Clock Cycle ⟶

	1	2	3	4	5	6	7	8
ADD R1,R2,R3	IF	ID	EX	MEM	WB			
SUB R4,R5,R1		IF	ID	ID	ID	EX	MEM	WB
AND R6,R7,R1			IF	IF	IF	ID	EX	MEM
OR R8,R9,R1						IF	ID	EX

SUB is held in ID for CC3 and CC4 (ID/EX is frozen on SUB).

The performance degradation is found from

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

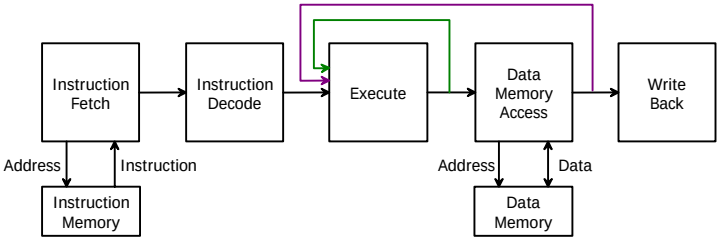
$$= \frac{\text{2 stall cycle}}{\text{stall}} \times \frac{\text{0.5 register dependencies}}{\text{ALU instruction}} \times \frac{\text{0.4 ALU}}{\text{instruction}}$$

$$= 2 \times 0.5 \times 0.4 \frac{\text{cycles}}{\text{instruction}} \quad \Rightarrow \quad CPI = 1.4 \Rightarrow \text{(29\% degradation)}$$

and is seen to be serious.

Slides 32 – 34 Forwarding or Bypass (DLX Version 3)

The register-register RAW stall is prevented in **DLX Version 3** by a technique called **forwarding** (or bypass). The basic idea is that the **ADD** instruction calculates its ALU result in CC3 but does not update R1 until CC5. If we force an early write to R1 we may create new data hazards. Instead, we **allow** the **SUB** and **AND** instructions to read **incorrect values** in ID (normal register reads) but we **replace the values** in EX when the instructions need them. We must provide the correct value from **EX/MEM.ALU** and **MEM/WB.ALU** directly to the ALU input. DLXv3 has new data paths from the buffers that store the result of instruction 1 directly to the ALU.

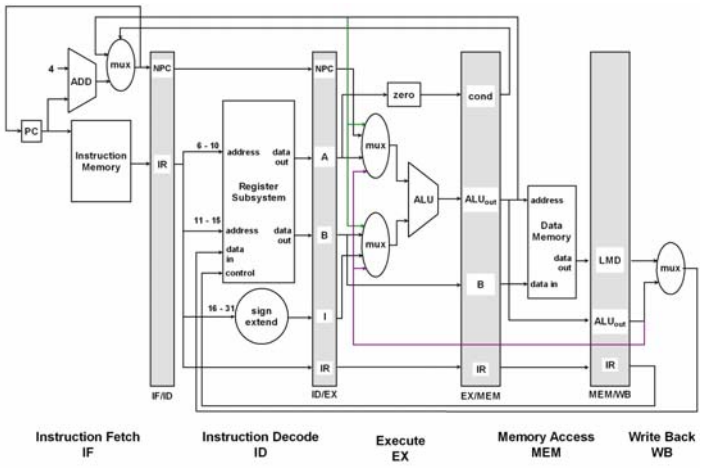


The processor view execution diagram is

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

The arrows show the forwarding of the result of ADD to the ALU.

The drawing of the DLXv3 microarchitecture (slide 33) shows the new data paths added.



The MUXes in EX have two new choices: EX/MEM.ALU_{out} (the previous ALU result) and MEM/WB.ALU_{out} (the ALU result from before the previous). Either can replace A or B in forwarding.

The pipeline stall in instruction view is

Clock Cycle ⟶

	1	2	3	4	5	6	7	8
ADD R1,R2,R3	IF	ID	EX	MEM	WB			
SUB R4,R5,R1		IF	ID	ID	ID	EX	MEM	WB
AND R6,R7,R1			IF	IF	IF	ID	EX	MEM
OR R8,R9,R1						IF	ID	EX

Instruction I_2 is held in IF for CC3 and CC4 (IF/ID is frozen on SUB).

The performance degradation is found from

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

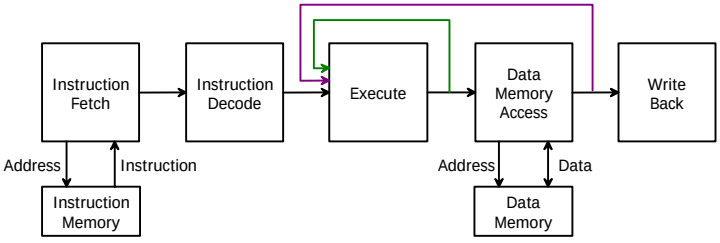
$$= \frac{\text{2 stall cycle}}{\text{stall}} \times \frac{\text{0.5 register dependencies}}{\text{ALU instruction}} \times \frac{\text{0.4 ALU}}{\text{instruction}}$$

$$= 2 \times 0.5 \times 0.4 \frac{\text{cycles}}{\text{instruction}} \quad \Rightarrow \quad CPI = 1.4 \Rightarrow \text{(29\% degradation)}$$

and is seen to be serious.

Slides 32 – 34 Forwarding or Bypass (DLX Version 3)

The register-register RAW stall is prevented in **DLX Version 3** by a technique called **forwarding** (or bypass). The basic idea is that the **ADD** instruction calculates its ALU result in CC3 but does not update R1 until CC5. If we force an early write to R1 we may create new data hazards. Instead, we **allow** the **SUB** and **AND** instructions to read **incorrect values** in ID (normal register reads) but we **replace the values** in EX when the instructions need them. We must provide the correct value from **EX/MEM.ALU** and **MEM/WB.ALU** directly to the ALU input. DLXv3 has new data paths from the buffers that store the result of instruction 1 directly to the ALU.

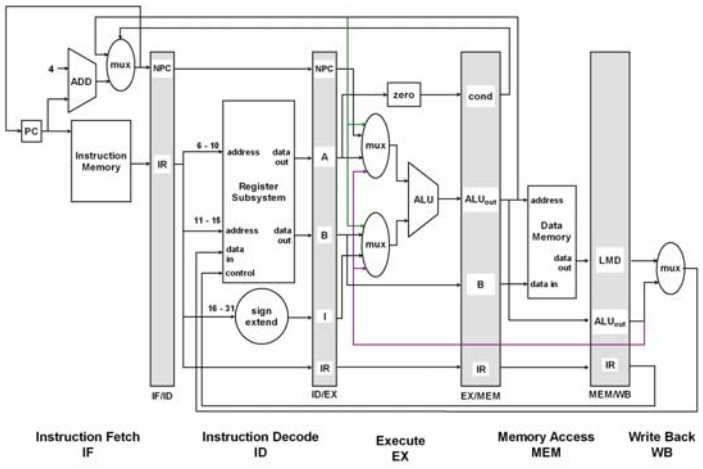


The processor view execution diagram is

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

The arrows show the forwarding of the result of ADD to the ALU.

The drawing of the DLXv3 microarchitecture (slide 33) shows the new data paths added.



The MUXes in EX have two new choices: EX/MEM.ALU_{out} (the previous ALU result) and MEM/WB.ALU_{out} (the ALU result from before the previous). Either can replace A or B in forwarding.

The instruction view execution diagram is

Clock Cycle ⟶

	1	2	3	4	5	6
ADD R1,R2,R3	IF	ID	EX	MEM	WB	
SUB R4,R5,R1		IF	ID	EX	MEM	WB
AND R6,R7,R1			IF	ID	EX	MEM
OR R8,R9,R1				IF	ID	EX

The processor moves the state of the **ADD** instruction from buffer to buffer

SUB needs the **ALU result** in **CC4**

ADD provides the **ALU result** from **EX/MEM.ALU**

AND needs the **ALU result** in **CC5**

ADD provides the **ALU result** from **MEM/WB.ALU**In DLXv3 there are no stalls for register-register dependencies and so $CPI^{stall} = 0$.

Slides 35 – 38 Register-Load RAW Dependencies in DLXv3

Another type of RAW dependency occurs when a load to a register does not complete before the register is needed. This case differs from the register-register load dependency because the required result is not available in the clock cycle following the EX stage.

Slide 35 replaces the instruction I_1 ADD R1, R2, R3 with the instruction I_1 LW R1, 32(R2). In the case of ADD, the value that is written to R1 in CC5 is available after CC3 and can be provided directly to the ALU for I_2 in CC4 and I_3 in CC5. In the case of LW, the address of the load is available after CC3, but the loaded data is not available until after CC4.

Slide 36 shows the new forwarding data paths in **DLX version 4** and the timing of the forwarding in processor view. Since the loaded data in only available for forwarding in CC5 the SUB instruction following the load must **stall for one cycle** in the ID stage. Slide 37 shows the new data path on the DLXv4 microarchitecture — the red line from LMD (load memory data) back to the multiplexors in EX.

Slide 38 shows the **forwarding** and the **stall** in instruction view. The data loaded by LW in MEM in CC4 is made available to the SUB in CC5. The SUB must wait in the ID stage during CC4 causing the stall.

The CPI^{stall} calculation requires:

1 stall cycle per stall

0.25 load instructions per program instruction (IC_{load} / IC)

The number of stalls per load instruction is just the number of times a loaded data operand is an ALU source in the next instruction. This happens about 50% of the time — in the other 50% another operand is usually loaded to a different register.

CPI^{stall} = 1 stall cycle per stall × 0.5 stalls per load × 0.25 loads per instruction = 0.125.

We will see (slide 41) that this performance degradation can be improved with a compiler optimization.

Slides 39 – 40 Register-Store RAW Dependencies in DLXv4

Another type of RAW dependency occurs when an ALU result is written to memory by the next instruction (slide 39). This dependency can also be treated with forwarding as seen in slide 40.

In **DLX version 5** a new multiplexor is added as the data input for memory store instructions. Normally the data written to memory is put into B during the ID stage of the SW instruction. When the program performs an ALU operation and stores the result immediately, the new multiplexor in the MEM stage can choose MEM/WB.ALU_{out} (the value that updates the register in the WB stage) instead of B (the previous value).

Slide 41 Compiler Scheduling to Prevent RAW Hazards

On the first pass a compiler that receives the C instructions

```
I = I + 123;
J = J - 567;
```

Will first compile

```
I = I + 123; →    LW   R2, I
                  ADD R2,R2, #123
                  SW   I, R2
```

and then compile

```
J = J - 567; →    LW   R3, J
                  SUB R3, R3, #567
                  SW   J, R3
```

If the program runs this way, there will be a stall cycle following each LW instruction.

An optimizing compiler will make a second pass and reschedule (rearrange) the instructions in the following order:

```
I1 LW   R2, I
I2 LW   R3, J
I3 ADD R2,R2, #123
I4 SW   I, R2
I5 SUB R3, R3, #567
I6 SW   J, R3
```

The new program order does not change the numerical results because the new order respects the data dependencies in the original version. But as can be seen in the execution diagrams on slide 41 this version has no stall cycles. Instruction I_2 is fetched before I_3 so that when I_3 reaches EX the value of R2 loaded by I_1 is available for forwarding. When I_5 needs the value of R3 it is already in the register.

The first version runs in 12 CC and the second version runs in 10 CC.

Slides 42 – 44 DLX Control Hazard

In versions 2 – 5 of the DLX, PC ← NPC on each clock cycle so that there is a new PC for a new instruction fetch in every clock cycle. The **control hazard** occurs whenever there is a conditional branch instruction because the next instruction (taken or not taken) is not yet known.

Definitions

Fall-through — the instruction immediately following a branch in the program listing

Target — the instruction pointed to by the branch

Branch taken — the condition for the branch is true and the branch causes a jump to the target instruction

Branch non-taken — the condition for the branch is false and the branch continues to the fall-through instruction

The stages of execution for a branch are

CLK	CC	Latched State	Action during CC
0	1	`Memory ← PC(branch)`	IF/ID.IR "sees" IR(branch) and PC(branch)
1	2	`IF/ID.IR ← branch`	Decode of branch instruction, NPC, I
2	3	`ID/EX.NPC ← NPC` `ID/EX.I ← I`	Calculate address NPC + I and cond
3	4	`EX/MEM.ALU ← ALU` `EX/MEM.cond ← cond`	PC "sees" correct address via MUX using cond to choose NPC or NPC+I
4	5	`PC ← branch address`	IF/ID.IR "sees" correct next instruction

When the instruction fetched on CC1 is a branch, the correct address for the next instruction (target or fall-through) is not known until the end of CC4 and cannot be fetched until CC5.

The simplest treatment for the DLX control hazard is to stall and **flush the pipeline**:

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ϕ	ϕ	IF	ID	EX	MEM	WB
I_3				ϕ	ϕ					
...	...	...	...	...	...	...	...	...	...	...
I_T	Target					IF	ID	EX	MEM	WB

CC1 fetch BEQZ

CC2 BEQZ is identified in ID as a conditional branch
The fall-through is fetched in IF (automatically)

CC3 The fall-through is flushed (deleted) from the pipeline
EX calculates the target address in the ALU and evaluates the condition

CC4 PC "sees" correct next address from EX/MEM.ALU and EX/MEM.cond

CC5 PC latches correct next address and fetches fall-through or target (depends on cond)

The performance degradation is severe:

CPI^{stall} = 3 CC per stall × 1 stall per branch × 0.20 branches per instruction = 0.60

This is equivalent to a 38% degradation in performance.

The branch stall will be treated in 3 steps: versions 6a, 6b and 6c.

Slides 45 – 48 DLX Version 6

The first enhancement (**DLXv6a**) is to add a special CLK to the PC to prevent waiting during CC4. At the end of CC3 an extra CLK pulse triggers PC to read from PC+4 or EX/MEM.ALU (depending on the value in EX/MEM.cond). The new instruction is now fetched in CC4 (instead of CC5). This reduces the number of stall cycles from 3 to 2 as shown on slide 45.

The second enhancement (**DLXv6b**) is more complex. Instead of computing the target address in the general ALU (stage EX), a special dedicated ADDER is placed in the ID stage. Also the comparator to set the cond flag is moved to ID. Instead of storing NPC in ID/EX for use in the general ALU, the special ALU adds IF/ID.NPC to the value that will be stored in I. Buffer ID/EX has two new registers, NNPC = NPC + I and cond. These values are now available to the IF stage at the end of the ID stage of the conditional branch. This reduces the number of stall cycles from 2 to 1 as shown on slide 46. In the execution diagram we see that if the condition is false then the fall-through instruction is cancelled in CC2 and then fetched again in CC3.

The third enhancement (**DLXv6c**) is implemented in the controller. If the condition is false then the fall-through instruction is the correct instruction, so it is not cancelled in CC2. The pipeline can continue in this case as seen on slide 47. This means that there is a stall of 1 clock cycle when the branch is taken and no stall when the branch is not taken. The complete **microarchitecture** drawing for DLXv6c is shown on slide 48.

Slides 49 – 52 Version 6c Branch Processing

The execution of a **taken conditional branch** begins on CC1 with the fetch of the BEQZ instruction. Slide 49 shows PC providing the address of the BEQZ instruction and to the adder that produces the address of the fall-through instruction. The PC (a buffer) and IF/ID.NPC "see" the fall-through address as their inputs (but do not latch the new address until triggered by CLK). ID/IF.IR "sees" the instruction as its input. On the CLK trigger, PC, IF/ID.NPC and IF/ID.IR sample and store their inputs.

On CC2 (slide 50) the adder in ID calculates the target address from IF/ID.NPC and the immediate (that will be stored in I at the end of the clock cycle). The address of the fall-though is provided to memory which provides the fall-through instruction to the input of IF/ID.IR (fetch of fall-through). At the end of the clock cycle (CLK trigger) the address of the target and cond are latched into ID/EX.NNPC and ID/EX.cond

On CC3 (slide 51) the fall-through instruction is flushed from ID, which now holds a NOP. PC "sees" the address of the target as input. The system controller quickly recognizes the conditional branch (slide 52) and sends a second "special" CLK trigger to the PC (since the pipeline was flushed this does not have any effect on the rest of the CPU). Without wasting any more time, the PC latches the address of the target instruction — this causes the fetch of the target instruction in CC3.

On CC4 the target instruction is latched into IF/ID.IR and the execution continues.

Slide 53 Branch Performance of Version 6c

The treatment of conditional branch in DLXv6c is called **Predict-Not-Taken** because it provides better performance on a branch not taken (no pipeline stall):

Branch taken — Flush entire pipeline
Branch not taken — Continue instruction in IF

From statistics on programs from SPEC CINT the actual probabilities are:

Not taken 33%
Taken 67%

Calculating CPI^{stall} for branches

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{stall cycles}}{\text{taken branch}} \times \frac{\text{taken branch}}{\text{branch instruction}} \times \frac{IC^{branch}}{IC}$$

$$= 1 \times 0.67 \times 0.20 \frac{\text{cycles}}{\text{instruction}} = 0.13 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.13 \Rightarrow \text{(12\% degradation)}$$

This value is an improvement over the initial value of 38% degradation for pipeline flush, but is still in need of improvement. Large CPUs use a complex technique of branch prediction (slide 61) not implemented for DLX. About half of all transistors (and chip "real estate") in an Intel x86 CPU are used for branch prediction.

Slides 54 – 56 DLXv6c Pipeline

The DLXv6c pipeline is shown on slide 54. The drawing includes the data paths added for forwarding and a Floating Point Unit (FPU).

Execution of float instructions is similar to integer execution, except that instructions are sent to the **FPU** instead of the EX stage that contains an **integer ALU**. The FPU is complex because floating point is a more complex data structure than 2's complement signed integers.

The **forwarding mechanisms** treat ALU-to-ALU RAW data dependencies with no stall and Load-to-ALU RAW data dependencies with a 1 CC stall. The ALU-to-Store data dependency is handled with no stall if the Store follows the ALU immediately but causes a stall if another instruction comes between ALU and Store. All other data dependencies cause a stall until the required data appears in its register following a write-back. These situations are summarized on slides 57 – 60.

A **formal specification** of DLXv6c is given on slides 55 – 56.

Slides 57 – 60 Forwarding

Slide 57 shows the instruction view execution table for **ALU-to-ALU** forwarding. The arrows show the forwarding data paths used to provide ALU result back to the ALU, instead of the previous values read during the ID stage. The third instruction requires forwarding from both the first and second instruction. The second instruction forwards its result directly from EX after

the ALU operation completes. The first instruction forwards its result from MEM where it stores it to be written to R1 in WB.

Slide 58 shows the instruction view execution table for **Load-to-ALU** forwarding.

In the first case the loaded data is used in the next two instructions, causing a 1 CC stall between LW and the second instruction. The third instruction requires no forwarding — it reads R1 correctly in ID following the writeback.

In the second case we see another 1 CC stall between LW and ADD. But notice that the second and third instructions can be exchanged with no change to result. The rescheduled program is seen to execute without a stall.

Slide 59 shows the instruction view execution table for **ALU-to-Store** forwarding. There is no stall when the Store instruction follows the ALU instruction immediately. But as seen in the following diagram, inserting an unrelated instruction between the ALU and the Store creates a 1 CC stall. There is no forwarding mechanism to bring the ALU result from MEM to the Store input. The Store must stall until R1 is written to the register in WB.

Slide 60 shows the instruction view execution table for **ALU-to-Branch** operations. There is no forwarding mechanism to bring the ALU result from EX to the NNPC input in ID. The branch must stall for 2 CC until R1 is written to the register in WB. This case is best handled by compiler optimization — placing independent work between the ALU instruction and the branch.

Slide 61 Improvement by Re-Scheduling in DLXv6c

The result of compiler optimization is shown on slide 61 for the sample program given on slide 8 of chapter 5. The first version of the program loads the first two operands and performs an ALU operation, then another load and another ALU, and so on. At the end of the Load/ALU block it updates the loop index R1 and performs a branch. Each LW-ALU pair causes a 1 CC stall. The ALU-Branch causes a 2 CC stall.

The optimized code issued by the compiler after a second pass first updates the loop index R1, then performs all of the loads at once, then all the ALU instructions at once, and then the Branch. All of the data dependencies have been "distanced" from one another so that the latency (number of CC between the start of an operation and the availability of the result) is "hidden" by other work being performed. This version runs with no stall cycles (except for the 1 CC stall between BNEZ and the SUBI in the next loop iteration because of predict-not-taken), saving 5 CC per loop.

The speedup from this optimization is significant. The loop runs 256 times and so we may neglect the first instruction (ADDI R1, R0, #400), which runs only once. In each loop iteration, 10 instructions (SUBI to BNEZ) run.

CPI_1^{stall} = 256 x (3 CC for LW-ALU + 2 CC for ALU-Branch + 1 CC for Branch) / (256 x 10) = 0.60

CPI_2^{stall} = 256 x (1 CC for Branch) / (256 x 10) = 0.10

$$S = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI}{CPI'} = \frac{1 + CPI^{stall}}{1 + CPI^{stall\,'}} = \frac{1.60}{1.10} = 1.45$$

which is a 45% improvement.

Slides 62 – 65 General Branch Prediction

The branch statistics from SPEC CINT are

Branch not taken	33%
Branch taken	67%

So that most branch instructions are taken. This makes sense because branches are used to build loops and usually run more than once.

Branch prediction is an advanced technique that improves the execution of taken branches at the cost of doubling the complexity of the CPU. It is not implemented in the DLX model but is used in most modern RISC processors and in Intel x86 CPUs since the original Pentium (1995).

The extra hardware includes a **BRANCH PREDICTOR** that records statistics on branch instructions:

Source address of branch instruction
Target address
Taken/not-taken

Using these statistics the hardware can predict branch behavior based on previous behavior.

If branch prediction were added to the DLX it would appear as on slide 63:

The IF stage would include the **branch predictor** that assembles a **database** of branch instructions indexed by the source address (unique location of branch instruction in memory at run time). The entry for each record is information on its history of taken/no-taken and the target address it points to.

When a recognized branch instruction is fetched (on the 2^{nd} or 3^{rd} time the branch runs) the branch predictor makes a **prediction** of the next instruction address to fetch (target if taken or fall-through if not-taken). The IF fetches the predicted instruction.

When the branch instruction enters ID it is executed in the usual way to **validate** the prediction. If the prediction was correct then execution continues and the statistics are updated in the branch prediction database. If the prediction was incorrect — **misprediction** — then the pipeline is flushed, the correct instruction is fetched and the statistics are updated in the branch prediction database.

Slide 64 shows the execution diagrams for the first and second execution of a certain branch for a simple loop (highly simplified model). The branch is unknown on the first fetch and is predicted not-taken (no other guess can be made). The branch is taken and the misprediction is handled. On the second execution the branch is recognized and predicted as taken.

Slide 65 shows a very simple program with BNEZ as the control instruction for a loop. On the first execution of BNEZ it is unknown and mispredicted as not-taken. On the next N – 2 loop iterations the BNEZ is correctly predicted as taken. On the last loop iteration BNEZ is mispredicted as taken — the loop has finished and the branch is not-taken. Out of N iterations of B instructions per iteration, there are only 2 CC of stall, which is negligible for large N.

Slides 66 – 67 More Compiler Optimizations

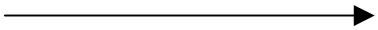
Another compiler optimization is **common sub-expression elimination**. When the compiler encounters instructions of the type

```
B = 10*(A/3);
C = (A/3)/4;
```

It calculates **(A/3)** into a register and then uses the register value in later calculations.

First Pass Compilation

```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R1,R1,R2
SW B,R1
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#4
DIV R1,R1,R2
SW C,R1
```



Second Pass Compilation

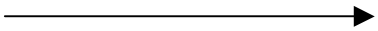
```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R3,R1,R2
SW B,R3
ADDI R2,R0,#4
DIV R3,R1,R2
SW C,R3
```

Loop unrolling is an optimization in which the compiler replicates instructions instead of implementing a loop. This is considered "inelegant" in high-level programming, but may be more efficient to implement because it eliminates the overhead of testing loop control variable.

Similarly, **inlining** is an optimization that replaces a procedure call with the code of the procedure or macro.

First Pass Compilation

```
00  ADDI R2,R0,#0x05
04  ADDI R1,R0,#0x08
08  LW R3,0x1000(R1)
0C  JAL 10
10  SW 2000(R1),R3
14  SUBI R1,R1,#0x04
18  BNEZ R1,-0x14
1C  ADDI R2,R0,#3
20  ADD R3,R3,R2
24  JR R31
```



Second Pass Compilation

```
00  ADDI R2,R0,#0x05
04  LW R3,0x1008(R0)
08  ADD R3,R3,R2
0C  SW 2008(R1),R3
10  LW R3,0x1004(R0)
14  ADD R3,R3,R2
18  SW 2004(R1),R3
1C  ADDI R2,R0,#3
```

In the first pass an operand is loaded to R3 and passed to the procedure at address 20. The ALU result is returned and stored in the instruction at address 10. The index is updated and the program loops back to LW at address 08. The loop runs twice. The Load-ALU pairs and ALU-Branch pairs cause many stall clock cycles for a short program.

Slide 68 More Hardware Optimizations

After RISC became established in the 1990s, so-called second generation RISC introduced new and complex hardware systems that further enhance pipeline efficiency and lower CPI below 1.

Superscaling

A superscalar CPU operates 2 or more pipelines in parallel. Instructions without dependencies can execute in parallel so that more than one instruction completes on each CC and CPI < 1. This method is used in most RISC processors and in Intel x86 processors since the original Pentium (1995).

Dynamic Scheduling

Instead of relying on compiler optimization for efficient instruction scheduling, the processor performs dynamic instruction scheduling at run time. The CPU chooses instructions with no unresolved dependencies from a pool of instructions, a very efficient method when combined with superscaling. This technique has been used in IBM mainframes since 1967 and is used in all Intel x86 processors since the Pentium II (1997).

Register Aliasing

Independent tasks and threads require access to the same logical registers (R0, R1, … as defined in ISA). Many processors provide physical registers than can be allocated per task from a large register pool. Then, multiple tasks can use the same logical register in parallel without causing data dependencies. This technique has been used in IBM mainframes since 1967 and is used in all Intel x86 processors since the Pentium II (1997).

Instruction Predication

A predicated instruction can be run or just cancelled according to a predicate flag associated with the instruction. Predication flags can be set from standard test-and-set instructions (SLT, SGT, SEQ, …). An IF-THEN-ELSE structure can be implemented as

```
if (test) set flag
(flag) action for true        // cancelled if flag = false
(!flag) action for false      // cancelled if flag = true
```

This eliminates the use of conditional branch instructions to jump over instructions and saves many stall clock cycles. Predication is used on the ARM RISC CPU, some microcontrollers and the Intel Itanium.

Chapter 7

Computer Arithmetic

עד עכשיו לא דיברנו על פרטי ה- ALU ו- FPU. רק הזכרנו שהרכיבים האלה קיימים במעבד ומשחקים תפקיד מרכזי. בפרק הזה נעשה חזרה קצרה על חשבון במחשבים, תחום ששיך למערכות ספרתיות.

Slide 2 **Unsigned Integers**

The standard n-bit representation of an unsigned binary number will be written as

$$k_2 = a_{n-1}\ \ a_{n-2}\ \ a_{n-3}\ \ \dots\ \ a_1\ \ a_0 \qquad a_i \in \{0,1\}$$

$$\underbrace{00\ \dots\ 0}_{n} \le k_2 \le \underbrace{11\ \dots\ 1}_{n}$$

We write the decimal value of k_2 as $k_{10} = K(k_2)$

$$k_{10} = K(k_2) = a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + a_{n-3}\times 2^{n-3} + \ \dots\ + a_1\times 2^1 + a_0\times 2^0$$

$$(k_2)_{\min} = 00\ \dots\ 0 \quad \Rightarrow \quad K\left((k_2)_{\min}\right) = 0$$

$$(k_2)_{\max} = \underbrace{11\ \dots\ 1}_{n} \quad \Rightarrow \quad (k_2)_{\max} + 1 = 1\underbrace{00\ \dots\ 0}_{n}$$

$$K\left((k_2)_{\max} + 1\right) = \left(1\times 2^n + 0\times 2^{n-2} + \ \dots\ + 0\times 2^0\right) = 2^n$$

$$K\left((k_2)_{\max}\right) = 2^n - 1,$$

$$0 \le k_{10} = K(k_2) \le 2^n - 1$$

A value $k > 2^n - 1$ cannot be represented by in n-bits and may cause overflow in an ALU.

Slides 3 – 5 **Signed Integers**

Signed numbers are written and stored in digital systems in the same way as unsigned numbers but are interpreted according to the 2's-complement method.

The value $k_2 = a_{n-1}\ \ a_{n-2}\ \ a_{n-3}\ \ \dots\ \ a_1\ \ a_0 \qquad a_i \in \{0,1\}$

may represents a non-negative value

$$0 \le k_{10} \le 2^{n-1} - 1$$

$$k_{10} = K(k_2) = K(0\ \ a_{n-2}\ \ a_{n-3}\ \ \dots\ \ a_1\ \ a_0)$$

$$= 0\times 2^{n-1} + a_{n-2}\times 2^{n-2} + a_{n-3}\times 2^{n-3} + \ \dots\ + a_1\times 2^1 + a_0\times 2^0$$

or a negative value (minus the 2's-complement)

$$0 > k_{10} \ge -2^{n-1} \Rightarrow k_{10} = -K\left((k_2)'+1\right)$$

$$k_2 = 11\ \dots\ 1 \Rightarrow k_{10} = -K\left((11\ \dots\ 1)'+1\right) = -K\left((00\ \dots\ 0)+1\right) = -1$$

$$k_2 = 11\ \dots\ 0 \Rightarrow k_{10} = -K\left((11\ \dots\ 0)'+1\right) = -K\left((00\ \dots\ 1)+1\right) = -2$$

$$\vdots$$

$$k_2 = 10\ \dots\ 0 \Rightarrow k_{10} = -K\left((10\ \dots\ 0)'+1\right) = -K\left((01\ \dots\ 1)+1\right) = -2^{n-1}$$

The high-order bit — the sign bit — is 0 for non-negative and 1 for negative values.

The decimal value of a 2's-complement signed number with sign bit = 1 is

$$0 > k_{10} \geq -2^{n-1} \Rightarrow k_{10} = -K\left(\left(k_2\right)'+1\right)$$

where k' = 1's-complement of k (reverse all 1 and 0 values). Therefore, for any value k_2

$$k_2 = a_{n-1}\ \ a_{n-2}\ \ a_{n-3}\ \ \ldots\ \ a_1\ \ a_0$$

$$\left(k_2\right)' = (1-a_{n-1})\ \ (1-a_{n-2})\ \ (1-a_{n-3})\ \ \ldots\ \ (1-a_1)\ \ (1-a_0)$$

$$K\left(k_2+\left(k_2\right)'+1\right) = K\left(a_{n-1}+(1-a_{n-1})\ \ a_{n-2}+(1-a_{n-2})\ \ \ldots\ \ a_0+(1-a_0)\right)+1$$

$$= K\left(\underbrace{11\ldots1}_{n}\right)+1$$

$$= \left(2^n-1\right)+1$$

$$= 2^n \text{ which has no representation in } n \text{ bits}$$

$$= 0$$

$$K\left(\left(k_2\right)'+1\right) = 2^n - K\left(k_2\right) = -K\left(k_2\right) + \text{overflow bit (ignored for signed)}$$

The sum k + (-k) in 2's-complement signed numbers is always 0 as required for arithmetic.

Slide 5 shows a general expression for the decimal value of a signed binary number.

Slides 6 – 8 Multiplication and Division

Binary numbers can be multiplied in long multiplication, for example

```
          0  1  1
      ×   1  0  1
    ---------------
          0  1  1
       0  0  0
    0  1  1
    ---------------
    0  1  1  1  1
```

From this procedure it is possible to generalize to the multiplication algorithm for unsigned numbers shown on slide 6 (in the example above a = 101, b = 001). The basic rule is that if $a_i = 1$ then shift and add b (as seen above in the first and last row) and if $a_i = 0$ then add 0 (as seen above in the middle row).

The algorithm for signed numbers (slide 7) is similar but the sign bit replaces the carry-out bit for unsigned numbers.

The algorithm for unsigned division is shown on slide 8.

Slides 9 – 13 Floating Point Numbers (IEEE-754 Standard)

Floating point numbers are represented by the data structure

s	e	f

where

s represents the sign of the number

e is the exponent — biased power of 2 (8 bits for float or 11 bits for double)

f is the fraction — (23 bits for float or 52 bits for double)

The value N of the represented number is

$$N = (-1)^s \times (1.f)_2 \times 2^{e-127}, \qquad \begin{matrix} 1 \le e \le 254 \\ -126 \le e-127 \le 127 \end{matrix} \quad \text{for float}$$

$$N = (-1)^s \times (1.f)_2 \times 2^{e-1023}, \qquad \begin{matrix} 1 \le e \le 2046 \\ -1022 \le e-1023 \le 1023 \end{matrix} \quad \text{for double}$$

For example, the value of the float

1	10000001	10100000000000000000000

is $(-1)^1$ x 1.10100000000000000000000 x $2^{129-127}$ = -1.101 x 2^2 = -110.10 = -6.50_{10}

Special values for **e** and **f** are

e	f	N	
0	0	0	
0	Not zero	$N = (-1)^s \times 0.f \times 2^{-127}$	$N = (-1)^s \times 0.f \times 2^{-1023}$
		denormal numbers — permit representation of very small numbers	
255	0	$N = (-1)^s \times \infty$	
255	Not zero	NaN (not a number — overflow/underflow)	

Slides 10 – 13 give examples of operations on floats and an indication of the algorithms used in the FPU to perform the operations.

Slides 14 – 15 DLX FP Pipeline

Because of the complexity of FP operations it is convenient to divide them into multiple pipeline stages. In the example, the ADD/SUB unit consists of 4 pipelined stages and the MULT/DIV unit has 7 pipelined stages. In these units, a new FP operation can begin on every clock cycle but it takes 4 or 7 CC for each operation to complete. The execution table for programs with FP operations is shown on slide 15. Notice that RAW data dependencies are more serious for FPU operations.

Slides 16 More Hardware

ALU speed can be limited by propagation delays, such as the time required for results to flow between bits in a ripple adder made of full adders. The **look-ahead adder** uses additional hardware to calculate complex pieces of these calculations in parallel and speed up the final result.

Chapter 8

Memory and I/O Organization

מעבר למעבד הראשי, מחשב מורכב מרכיבי אחסון ומערכת קלט/פלט (I/O) שמחברת את המעבד לרשתות וממשקי משתמש. בפרק הזה נדון בארגון מערכת הזכרון והיבטים של גישה להתקנים נוספים.

Slides 2 – 4 Principle of Locality

The principle of locality states is another **rule of thumb** — for 90% of run time the next instruction or data access will come from a small volume of memory (about 10% of the program) closest to the current instruction or data. Given an access to address A in a program that occupies a volume B of memory, the next access will be to an address within $A \pm 0.05B$ for 90% of run time. Slide 2 shows statistics for programs from SPEC CINT showing the percentage of total memory accessed in 80% (dark gray) and 90% (light gray) of run time.

Amdahl's law suggests that these memory accesses should be made as fast as possible. The **memory hierarchy** shown on slide 3 divides memory into types from long-term storage to register. Long-term **storage** holds all programs and data in a complex system organized by the operating system with relatively long access times. **Main memory** holds running programs and data of a smaller size, closer to the CPU and with faster access. **Cache** copies a section of main memory into the CPU for faster access. Both main memory and cache are organized by byte address. **Registers** hold a small amount of data in the CPU with fastest access — organization is defined in the ISA for the CPU.

The table on slide 4 shows typical sizes and access times for various memory types. These numbers improve constantly and may already be out of date.

Slides 5 – 7 CPU and Memory Hierarchy

The diagram on slide 5 shows the relationship between memory elements for the DLX. The Instruction Memory in the IF stage is **instruction cache** and the Data Memory in stage MEM is the **data cache**. These caches are called **Level 1 cache (L1)** because they are the first level of access to memory by the CPU. Cache holds a **copy** of a limited section of main memory. The sections copied to cache are characterized by their RAM address locations.

The L1 instruction and data caches are connected to a **cache controller** that also connects to a larger **Level 2 cache (L2)** and to the **external bus interface** that connects the CPU to the rest of the world. The CPU is connected over the external bus to an **I/O controller** (often called a **chipset**) that connects to **main memory** (RAM), **long-term storage** and other devices such as the user interface and network.

Slide 6 shows the operation of the cache update mechanism in general terms. When the information required by the CPU has been previously copied to L1 cache, then the instruction fetch or data access in the CPU takes place in 1 clock cycle. If the required information is not in L1, we say that the **addresses are not present in L1** (referring to the data/instruction contents of those addresses, not the actual addresses). In that case, L1 requests the information from L2 cache and from main memory. If the addresses are present in L2 then it is copied to L1 and the request to main memory (which takes much more time) can be cancelled. If the information is not in L2 then it will be copied from main memory to both L1 and L2. Copying from L2 to L1 can take several clock cycles — copying from main memory to L1 takes many clock cycles.

Two special cases are shown on slide 7. Although cache and main memory are accessed by the byte address of the information, long-term storage and other devices are addressed by the OS at a specific **device address**. For example, to write k bytes to the hard drive, the OS performs k successive byte writes to the single address assigned to the hard disk. A similar process occurs for network access. Since the information at these device addresses changes constantly there is no reason to copy them to cache — the device addresses are marked **non-cacheable** by the CPU.

Another special case is virtual memory in which the total address space is larger than the physical memory present in the machine. For example, a 32-bit Pentium 4 CPU from 1999 was capable of addressing 64 GB of memory but even today few personal computers hold 64 GB of physical memory. With virtual memory the relationship of program address and physical address may change, and may affect the relationship of main memory to cache.

Slides 8 – 12 Memory Organization

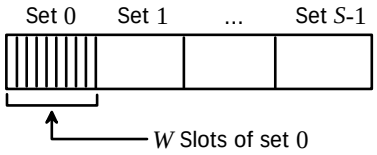
An n-bit address specifies the location of a specific byte in memory. The address space consists of 2^N byte locations from 0 to $2^n - 1$. The CPU must have an n-bit register to hold the address sent to the physical memory.

Information from main memory is copied to cache as needed (we refer to data to mean both data and instruction information). Working with cache presents three important questions:

How much data should be copied to cache?
Which data is located in cache at any given time?
How can the CPU find a particular address location in cache?

To simplify matters, bytes are copied to cache in **blocks** (called cache lines in Intel documents). A block is a collection of **B bytes** in memory (main or cache) and B is chosen so that the **page size** in virtual memory is an integer multiple of B (a page swap involves an integer number of blocks).

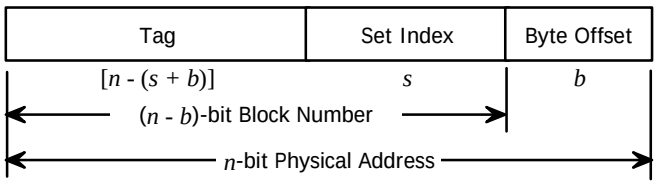
The cache is divided into **S sets** and each set is divided into **W slots**. Each slot holds exactly one block. The organizing rule is: **The block number determines the set number — any slot in the set is allowed to hold the block**.



S sets in cache

As shown on slide 10, the block size is chosen as a power of 2: **$B = 2^b$**. The **block number** for the n-bit **address A** is the **upper n – b bits** of the address = int[A / B]. The lower b bits are called the **block offset** (the number of a certain address within the block from 0 to B – 1).

Similarly (slide 11) the number of sets is a power of 2: **$S = 2^s$**. The set index is the lower s bits of the block number, and is found from **set number = block number % S**. A block copied to cache must be placed in the set numbered by the set index.



Parsing of n-bit address

The remaining upper n – (b + s) bits of the address are called the **tag**. The tag is written into a register associated with the cache to identify the blocks copied to the set. Writing the tag and set index provides the block number.

To **find a particular byte in cache**, the CPU calculates the block number for that address, then the set index for than block number, and then searches the tags of the blocks located in the set. If the tag is found, then the contents of the associated slot in the set is the desired block. The required byte is found by counting within the bytes stored in the slot.

The total cache size is

$$\text{total cache size} = S \times W \times B = \frac{\text{sets}}{\text{cache}} \times \frac{\text{blocks}}{\text{set}} \times \frac{\text{bytes}}{\text{block}} = \frac{\text{bytes}}{\text{cache}}$$

CPU documentation usually specifies the total size along with W and B.

Slides 13 – 15 **Example of Memory Organization**

A 32-bit address defines a 2^{32} = 4GB memory space.

Choose a block size B = 2^b with b = 4 and B = 16. The block number is 32 – 4 = 28 bits and defines 2^{28} = 256 Mblocks.

Choose S = 2^s with s = 8 and S = 256 sets. When block B is copied to cache it must be assigned to set B % 256. Set 0 may hold block 0, or 256, or 512, and so on. Set 1 may hold block 1, 257, 513, and so on.

The tag number is 32 – (4 + 8) = 20 bits.

Choose W = 2^2 = 4 slots per set. There are "4 ways" to choose the location of a block in an empty set and the number of "ways" is called **associativity**. This cache is **4-way associative**.

The total cache size = 256 sets/cache x 4 blocks/set x 16 bytes/block = 16 KB.

At some moment in run time set 1 may hold blocks 1025, 513, 1281, 257. The tag numbers that identify these blocks are int[block/256] = 4, 2, 5, 1.

If the program requires access to address 20509_{10} = 0x0000501D the cache controller parses this address as byte 0x0D = 13 in block 0x501 which — if it is in cache — must be located in set 01 with tag 5. The cache controller scans the tags for set 1 and finds 5 in slot 2 (of slots 0, 1, 2, 3).

The cache controller reads the 16 bytes of block 0x501 and sends byte 0x0D to IR or LMD.

Slide 16 **Cache Definitions and Policies**

Cache hit	The CPU requires a block for read or write and finds the block in cache.
Cache miss	The CPU requires a block not in cache. On a read miss the cache controller loads the block to cache.
Write allocate	A cache policy: on a write miss the cache controller loads the block to cache.
No write allocate	A cache policy: on a write miss the cache controller writes directly to RAM without loading the block.

Swapping out a cache block

The CPU accesses a block not in cache and must copy the block to cache, but all of the slots in the required set are full. The cache controller must remove (swap out) the block that is least recently used (LRU). There are two possible policies for cache / RAM consistency:

Write Through — the cache controller updates RAM on every write to cache

Write Back — the cache controller updates main memory when the block is swapped out

Slides 17 – 18 Cache Performance Issues

L1 Cache hit	The CPU reads/writes memory in 1 clock cycle.
L1 Cache Miss	The CPU stalls while L1 loads the missing block from L2 or main memory
Miss Rate	The number of cache misses per cache accesses (instruction fetch, data load or data store). This rate depends on cache size and organization.
Miss Penalty	The number of stall cycles while cache loads a missing block depends on cache hardware technology and organization

Cache misses contribute to CPI^{stall}.

Using the definitions of miss rate and miss penalty, we easily calculate CPI^{stall} for a **1 level unified cache** (not split into instruction and data):

$$\begin{aligned} CPI^{stall} &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}} \\ &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{memory accesses}}{\text{instruction}} \\ &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction accesses} + \text{load accesses} + \text{store accesses}}{\text{instruction}} \\ &= \text{miss penalty} \times \text{miss rate} \times \frac{IC + IC^{load} + IC^{store}}{IC} \end{aligned}$$

CPI^{stall} for a **1 level split cache** treats instruction cache misses and data cache misses separately:

$$\begin{aligned} CPI^{stall} &= \frac{\text{instruction stall cycles}}{\text{stalls}} \times \frac{\text{instruction stalls}}{\text{instruction access}} \times \frac{\text{instruction accesses}}{\text{instruction}} \\ &\quad + \frac{\text{data stall cycles}}{\text{stalls}} \times \frac{\text{data stalls}}{\text{data access}} \times \frac{\text{data accesses}}{\text{instruction}} \\ &= \text{instruction miss penalty} \times \text{instruction miss rate} \times \frac{1 \text{ instruction access}}{\text{instruction}} \\ &\quad + \text{data miss penalty} \times \text{data miss rate} \times \frac{IC^{load} + IC^{store}}{IC} \end{aligned}$$

For most programs the instruction cache miss rate is lower than the data cache miss rate because most instructions executed are located in loops — an entire loop may be loaded to cache on the first iteration and there will be no instruction cache misses while the loop runs. Access to a large data structure may be sequential with little return to previously accessed data.

For example, with

Instruction Miss Penalty = Data Miss Penalty = 50 cycles per stall
Instruction Miss Rate = 0.5%
Data Miss Rate = 5.0%
IC^{load} = 25%
IC^{store} = 15%

$$CPI^{stall} = 50 \times 0.005 \times 1 + 50 \times 0.05 \times 0.40 = 0.25$$

$CPI = 1.25 \Rightarrow 20\%$ degradation

This high performance degradation makes the addition of a 2nd level of cache worthwhile.

Slide 19 Two-Level Unified Cache — Definitions

We model a two-level cache as follows:

L1 cache has miss rate M_{L1} so that its hit rate is $1 - M_{L1}$. On a miss at L1, the controller accesses L2 and main memory.

L2 cache has miss rate M_{L2} so that its hit rate is $1 - M_{L2}$. On a miss at L2, the controller must wait for main memory.

On an L1 miss and an L2 hit the miss penalty is P_{L1}. On an L1 miss followed by an L2 miss an extra penalty P_{L2} is added — the total penalty is $P_{L1} + P_{L2}$.

$IC^A = IC^{load} + IC^{store}$ is the total number of data accesses. As always, the total number of instruction access is just the total number of instructions = IC.

Slides 20 – 23 Performance of Two-Level Unified Cache

Slide 20 shows an event tree that analyzes the probability of incurring each penalty. We derive the CPI^{stall} formally:

$$CPI^{stall} = \frac{\text{stall cycles}}{IC} = \sum_{\text{n = stall types}} \frac{\text{stall cycles}}{\text{stall type n}} \times \frac{\text{stalls of type n}}{IC}$$

$$\text{where stall types} = \{\text{hits, misses, data, instructions}\}$$

$$= \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{IC} \right]$$

$$= \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{IC} \right]$$

We make the (approximately valid) assumption that misses at L1 and L2 are statistically independent. Then

$$\frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 hit}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 hit}}{\text{L2 access}} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}}$$

$$\frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 miss}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 miss}}{\text{L2 access}} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}}$$

and so

$$CPI^{stall} = \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{IC} \right]$$

Using the definitions for miss rates and miss penalties, and isolating common factors this becomes

$$CPI^{stall} = P_{L1} \times M_{L1} \times (1 - M_{L2}) \times \frac{IC + IC^A}{IC} + (P_{L1} + P_{L2}) \times M_{L1} \times M_{L2} \times \frac{IC + IC^A}{IC}$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} \times (1 - M_{L2}) + (P_{L1} + P_{L2}) \times M_{L2}\right]$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} - P_{L1} \times M_{L2} + P_{L1} \times M_{L2} + P_{L2} \times M_{L2}\right]$$

$$CPI^{stall} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

Notice that this expression can be read directly off the **event tree**:

CPI^{stall} = L1 miss rate × L1 access rate^{I+D} × (L1 penalty + extra penalty × L2 miss rate)

Slides 24 – 25 Split L1 Cache with Unified L2 Cache

For the unified cache we have

$$\frac{\text{misses at L1}}{IC} = \frac{\text{misses at L1}}{\text{accesses at L1}} \times \frac{\text{accesses at L1}}{IC} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] = M_{L1} + M_{L1} \times \frac{IC^A}{IC}$$

To find the expression for a split cache (I + D in separate caches) we replace:

$$\frac{\text{misses at L1}}{IC} \rightarrow \frac{\text{data misses at L1}}{\text{data accesses at L1}} + \frac{\text{instruction misses at L1}}{\text{instruction accesses at L1}}$$

$$= M_{L1}{}^{\text{instructions}} \times 1 + M_{L1}{}^{\text{data}} \times \frac{IC^A}{IC} = M_{L1}{}^{I} + M_{L1}{}^{D} \times \frac{IC^A}{IC}$$

and so

$$CPI^{stall}{}_{\text{unified}} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

$$\rightarrow \quad CPI^{stall}{}_{\text{split}} = \left[M_{L1}{}^{I} + M_{L1}{}^{D} \times \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

Comparing the expressions for **1-level** (slide 18) and **2-level** cache, the miss rates are the same and the enhancement between is entirely in the miss penalty:

$$CPI^{stall}_{1-level} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^A}{IC}\right] \times P_{L1} \qquad CPI^{stall}_{2-level} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^A}{IC}\right] \times \left(P_{L1} + P_{L2} \times M_{L2}\right)$$

In 1-level cache, the penalty is L1 to RAM and so P_{L1} = 50 CC.

In 2-level cache, the penalty is P_{L1} = 5 CC for L1 to L2 and on an L2 miss an extra penalty of P_{L2} = 45 CC for L2 to RAM. With a 1% miss rate at L2, $P_{L1} + P_{L2} \times M_{L2}$ = 5 + 45 x 0.01 = 5.45 CC.

Slides 26 – 31 Issues Affecting Miss Rate

Cache misses can be divided into three basic types:

Compulsory miss

Since a block is not copied to cache until the first access to a byte in the block, there is always a first access to a block and the first access will always miss in cache. Compulsory misses are not affected by cache properties.

Capacity miss

The cache is smaller than main memory and some blocks must be removed from cache to make room for other required blocks. If the cache were larger it would be possible to hold more data with less swapping out. The capacity miss rate is lower for a larger cache size.

Conflict miss

A given block must be copied to specific set. There are times when some blocks must be removed from the set to make room for required blocks, even when there are empty slots in other sets. This is not caused by an overall capacity miss, but by the associativity of the cache organization. For larger W there is less likelihood that one set will be full while another set has space. The conflict miss rate is lower when block placement (associativity) is more flexible. For example, conflict misses can be caused by address aliasing — data structures from different sections of memory alias to the same set and conflict.

Slides 27 and 28 show statistics on cache misses as size and associativity vary. The compulsory miss rate stays constant as expected. For any given cache size the conflict miss rate drops as associativity goes up. The capacity miss rate drops as the cache size increases. These statistics are summarized in the graph on slide 29. Slide 30 shows the conflict miss rate by cache size and associativity. For standard cache sizes there is very little improvement above W = 4 and this is the usual choice. The reason for restricting associativity is explained on slide 31. For a fixed cache size, a larger W requires fewer sets $S = 2^s$ and so the tag length $n – (b + s)$ is longer. A longer tag requires a longer tag search, which slows down cache access.

Slides 32 – 39 Miss Examples

We begin with two examples at the extremes of cache behavior — locality and non-locality.

Extreme locality

The program

`int i,a;for (i = 0 ; i < 4096 ; i++){ a = i;}`with **`i`** stored in a register and **`a`** stored in memory performs 4096 memory writes. As in most cache systems, the write allocate policy means that the first access to **`a`** causes a compulsory miss and load of **`a`** to cache. There are no further misses and so the miss rate is just 1 miss / 4096 accesses = 2.44×10^{-4}.

Extreme non-locality

The program

```
int i,a[4096];
for (i = 0 ; i < 4096 ; i++){
      a[i] = i;
}
```

with **i** stored in a register and **a[]** stored in memory also performs 4096 memory writes, but now to 4096 different addresses **a[i]**. We take the block size = 16 bytes/block which means that each block holds 4 words = 4 array elements. This time there are 4096/4 = 1024 compulsory misses and loads to cache. There are no read misses and so the miss rate is 1024 misses / 4096 accesses = 0.25 = 25%.

More practical examples require more **details of cache parameters**: We take B = 16 bytes per block, S = 256 sets and W = 4 slots per set. Total cache size is 16 x 256 x 4 = 16 KB = 4 Kwords.

Good locality

The program

```
int a[4096],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 4096 ; j++){
      a[j] = i + j;
      }
}
```

with **i,j** stored in registers and **a[]** stored in memory performs 40960 memory writes. There are 4096/4 = 1024 compulsory misses and loads to cache for **i = 0**, but the 16 KB of the array fills the cache and there are no further misses on **i > 0**. The miss rate is now 1024 misses / 40960 accesses = 0.025 = 2.5%.

Extreme non-locality

Making the array **a[]** too large to fit into cache has a dramatic effect on the miss rate. The program

```
int a[5120],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 5120 ; j++){
      a[j] = i + j;
      }
}
```

with **i,j** stored in registers and **a[]** stored in memory performs 51200 memory writes, to 5120 different addresses **a[i]**. For **i = 0** there is a compulsory miss every 4 elements, a total of 5120 / 4 = 1280 misses. But unlike the smaller array, the first outer loop iteration does not finish with the full array in cache. Looking at set 0 for i = 0,

on j = 0, block 0 → slot 0
on j = 1024, block 256 → slot 1
on j = 2048, block 512 → slot 2
on j = 3072, block 768 → slot 3
on j = 4096, block 1024 → slot 0

Since block 1024 must enter slot 1024 % 256 = 0 it forces the **eviction** (swap out) of block 0 (the LRU). Set 0 now holds block 1024, 256, 512, 768.

On the second iteration of the outer loop, i = 1

on j = 0, block 0 → slot 1	cache **evicts** the LRU = block 256
on j = 1024, block 256 → slot 2	cache **evicts** the LRU = block 512
on j = 2048, block 512 → slot 3	cache **evicts** the LRU = block 768
on j = 3072, block 768 → slot 0	cache **evicts** the LRU = block 1024
on j = 4096, block 1024 → slot 1	cache **evicts** the LRU = block 256

This continues until the end of the program. Again there is a capacity miss every 4 accesses and so the miss rate is 25%.

Slide 36 shows that this high miss rate could be improved if it were possible to switch the cache controller to evict the most recently used (MRU) instead of LRU. In that case, slots 0, 1 and 2 would be stable after the first outer loop iteration, while slot 3 of set 0 alternates between the 768 and 1024. The miss rate falls from 25% to 11.5%.

The most effective way to prevent cache misses for this program is to split the loop into two pieces as

```
int a[5120],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 4096 ; j++){
            a[j] = i + j;
      }
for (i = 0 ; i < 10 ; i++){
      for (j = 4096 ; j < 5120 ; j++){
            a[j] = i + j;
      }
}
```

With this optimization both sections run with a miss rate of 2.5%. Unfortunately, this solution suffers the severe disadvantage of requiring that the programmer be concerned about performance.

Address aliasing

We consider the program (slide 38)

```
int a[512],b[512],c[512],i,j;
  for (i = 0 ; i < 20; i++){
    for (j = 0 ; j < 512 ; j++){
      c[j] = a[j] + b[j] + c[j];
    }
  }
```

where the arrays are placed at the addresses

```
a: 02000000 – 020007FF
b: 02001000 – 020017FF
c: 02002000 – 020027FF
```

Array **b[]** does not follow array **a[]** immediately in memory, but a compiler may do this for convenience.

We first consider the case that B = 16 and S = 256 with W =2. The block offset for each address is the last hex digit (in green), and the set number is the two next hex digits (in blue). We see that all 3 arrays must be placed in sets 00 – 7F. No blocks are assigned to sets 80 – FF which remain empty.

For the **`i = 0`** iteration of the outer loop, there is a compulsory miss for **`a[j]`** loaded to slot 0, then a compulsory miss for **`b[j]`** loaded to slot 1, then a compulsory miss for **`c[j]`** loaded to slot 0 (evicting **`a[j]`**) — there is no miss for **`c[j]`** on the write.

For the **`i > 0`** iteration of the outer loop, there is a capacity miss for **`a[j]`** loaded to slot 0 (evicting **`c[j]`**) and a capacity miss for **`c[j]`** (evicting **`a[j]`**) loaded to slot 0.

The miss rate is 12.8%.

If the cache we reorganized (slide 39) with fewer sets and more slots per set (S = 128 and W = 4) then all sets 00 – 7F will contain blocks. Each array will occupy one of the 128 sets and there will be no conflict misses — no misses for **`i > 0`**.

The miss rate will drop from 18,8% to 0.94%.

Slides 40 Workstation Layout with PCI

Slide 5 shows the CPU external bus connection to main memory and long-term storage through an I/O controller. Slide 40 shows a simplified layout of a workstation using the **PCI** standard I/O system. PCI (Peripheral Component Interconnect) is organized around an I/O controller called a **bridge** that determines communications protocols among the CPU, main memory, long-term storage, user interface, network interfaces and other I/O systems.

The CPU and RAM are directly connected to the bridge. A switching fabric (a data exchange protocol that may be an active switch as in a computer network, or a simple bus that permits only one transmission at a time) connects other device controllers to the bridge. Software drivers installed in the operating system provide an API (application programming interface) between the CPU and the device controllers.

Slides 41 – 46 PCI Services

In addition to permitting CPU access to main memory and long-term storage, the PCI system provides system services:

Boot services

The PCI enables access to BIOS — the basic input/output system. This is a ROM-based device holding software routines that enable system initiation and OS boot.

Timers

The PCI operates a collection of system timers, counters and real time clocks for managing, coordinating and synchronizing system activities.

Interrupt controllers

The PCI implements a programmable interrupt controller that enables hardware interrupts to be enabled and assigned to system activities via IRQ (interrupt requests). On modern systems an advanced programmable interrupt controller (APIC) also permits generation, transfer and interpretation of interrupt messages between processors.

Direct Memory Access (DMA)

The DMA system permits devices to define memory transfers that do not require CPU intervention for byte-by-byte read and write.

BIOS

BIOS is the BASIC INPUT/OUTPUT SYSTEM and provides code that implements the minimum functionality of the workstation at system start-up. When the hardware system is started or reset, the CPU performs a self-check (also known as a POST — power-on self-test — or a sanity check). If the CPU is OK, then it fetches an instruction from physical address **FFFF0h** (a defined constant address near the top of the physical memory space of the Intel 8086). Address FFFF0h contains a branch instruction — the branch target is a code section located in PCI BIOS ROM (Read Only Memory) called **firmware**. (The form of ROM is usually E^2PROM, Electrically Erasable Programmable ROM, that permits updating of the system code.)

The BIOS firmware permits location and operation of the keyboard, display, and a boot device (a long-term storage unit containing an operating system). Most workstations today contain an additional level of firmware above BIOS called **UEFI** (Unified Extensible Firmware Interface). UEFI is a small hardware-oriented operating system that performs system management including boot of the main OS. When UEFI is present, BIOS loads UEFI and UEFI loads the main OS from the boot device. In a non-UEFI system, the BIOS loads the OS from boot device directly.

Interrupt Handling

An **interrupt** is an asynchronous event generated outside the CPU that temporarily halts normal instruction flow in a controlled manner, directs the CPU to a handle task in need of attention and then restarts normal instruction flow in a controlled manner. A **trap** is a software instruction that causes behavior similar to an interrupt.

PCI-compatible CPUs contain a **local APIC** (Advanced Programmable Interrupt Controller) that enables software-defined interrupt handling. The APIC accepts **local interrupts** generated by an interrupt number placed on the processor external bus interface and triggered by an input signal on the bus called **INTR**. The APIC also sends and receives formatted **interrupt messages** on a CPU bus shared among processors in the system. The OS uses these messages to **assign threads to cores**, among other coordination tasks.

The **I/O APIC** behaves like a local APIC except that it belongs to the I/O system attached to the PCI bridge. This enables attached devices (user interface, long term storage, networks and so on) to exchange interrupt messages of the type exchanged by CPUs. In older workstations (20th century), a subset of the local interrupts was assigned to the IRQ (interrupt request) system. Each hardware device was assigned an **IRQ** number by the OS. In a PCI system with processor support, IRQs are emulated in the interrupt message system. All hardware devices share a single IRQ number — the actual system request is sent in an interrupt message.

The messages between processor cores are called **Interprocessor Interrupts** (IPI). The drawing on slide 44 shows the Processor System Bus as a common communication channel for the CPUs, also attached to the PCI bridge. The I/O APIC permits the I/O system to behave as another CPU on the system.

Direct Memory Access (DMA)

DMA permits a peripheral device (such as the user interface, storage and network) to access main memory directly without intervention by the CPU to execute transfer instructions. It is

used for large data transfers. DMA permits the CPU to continue working concurrently with the transfer and the CPU can preempt DMA to perform cache updates.

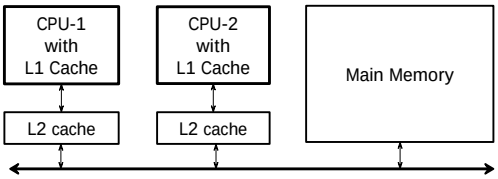
In **interrupt mode**, the CPU executes code that instructs the DMA controller to set up the transfer. The DMA request includes the start address in memory and the number of bytes to transfer. The DMA controller acts as master of the data path and actively transfers data between RAM and the peripheral device. The DMA controller executes an IRQ at end of the transfer and the CPU takes back control of the bus.

In **PCI arbitration**, the peripheral device requests control of the bus directly from the PCI bridge, including details of the required read or write memory access. The PCI bridge grants bus control of the system bus to the attached device and executes the data transfers.

Slides 47 – 48 Cache Coherency and Consistency

On slide 16 we saw two methods (**write back** and **write through**) to maintain consistency of the information stored in the CPU cache and the main memory. With multiple CPU cores in most computers, telephones and tablets, the consistency problem becomes more complex.

Suppose that CPU-1 and CPU-2 are running **concurrent threads** in the same process. Both CPUs read variable X to cache from main memory. If CPU-1 updates X in its cache copy, then this write to X invalidates the copies of X in CPU-2 cache and in main memory — the correct value of X is stored in CPU-1 cache and the old copies in CPU-2 cache and in main memory should not be used (another example of a RAW hazard).



Cache consistency is the case in which the copies of data in all locations are always identical, and is extremely difficult to obtain. **Cache coherence** is the case in which all reads and writes occur in the correct order — this is the goal of cache management in multicore systems.

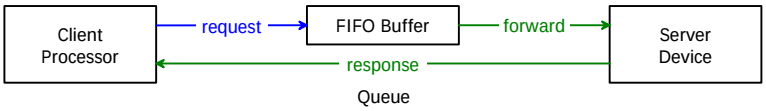
The most common method for maintaining cache coherence is **bus snooping**. All caches in the system use the write back policy. The CPU updates its private cache with no update to main memory. Main memory is updated only when a cache block is evicted from cache. But on every CPU write to a block in its cache the **CPU writes the destination addresses** on the shared processor bus (this is a short and fast bus cycle). All devices on the bus monitor these addresses, which allows each CPU and the PCI I/O device to see which CPUs are loading a particular cache block, and to see which CPUs are writing to a particular cache block.

The bus enforces **write synchronization** — bus arbitration prevents writes to multiple copies of a cache block. Only one CPU can place a target address on the memory bus per bus cycle, and only one CPU can write to any given cache block in a bus cycle.

When a CPU or the main memory sees a write to a certain address by another CPU, it marks its own copy of the block **invalid**. If two CPUs load X at the same time and try to write to X at the same time, only one CPU will be allowed to write by the write synchronization method. The other CPU will have to invalidate its copy of X and cancel all the work that led up to the attempted write of X. After invalidating X, the CPU that needs X will try to read the memory location from main memory. The read attempt will trigger the CPU that has the valid copy to evict the cache block and update main memory. The problem of cache coherence is one of the main sources of **complexity in parallel programming**.

Slides 49 – 52 Client/Server Model of I/O

It is useful to model the relationship between CPU and an I/O device (including storage) as a client/server system. The CPU operates at a much higher data transfer rate than I/O devices and data accesses must be stored in a FIFO (first-in, first out) buffer. The buffer stores read/write requests from the CPU (up to some maximum capacity) and forwards them to the I/O device at the lower rate.



Characteristics of a client/server system:

Latency The time between a client request and the buffer response

Throughput The number of services provided per unit time

Bandwidth The maximum data transfer rate of the I/O channel (including buffer)

Capacity The maximum throughput of the server through the buffer, which depends on the bandwidth and the service rate (server speed)

Utilization The request rate as a proportion of capacity

The FIFO buffer can handle the client/server rate disparity because client requests are bursty, appearing intermittently at high speed. The peak client request rate is much larger than the average client request rate. Generally,

Peak client rate > Server rate > Average client rate

The client sees a **minimum latency** determined by the maximum service response rate (which is related to the minimum time required to service one request). This creates a trade-off between utilization and latency. As more requests are stored in the buffer, the latency increases. If the client (CPU) makes more requests than can be held by the FIFO then there will be overflow and data will be lost.

Slides 53 – 55 Queuing Theory

Queuing theory provides a mathematical description of queues in client/server systems. The simplest results begin with the following assumptions:

Client requests arrive independently (by Poisson statistics) and have a random length (number of bytes to transfer). Over the long run, requests reach a steady state (small variations around a fixed value) characterized by the average request rate.

The **buffer** stores requests and forwards in order of arrival (FIFO) at service rate. In the steady state there is an average buffer level (number of stored requests).

The **server** provides services to each request independently (by Poisson statistics). In the steady state there is an average service rate.

The results of Markov chain analysis are:

$$\text{Utilization} = \frac{\text{Request Rate}}{\text{Service Rate}}$$

$$\text{Latency} = \frac{1}{\text{Service Rate} - \text{Request Rate}} = \frac{1}{\text{Service Rate}}\left(\frac{1}{1 - \text{Utilization}}\right)$$

$$\text{Buffer Level} = \text{Latency} \times \text{Request Rate} = \text{Utilization} \times \left(\frac{1}{1 - \text{Utilization}}\right)$$

Buffer level and latency rise sharply with increased utilization.

85% utilization is a reasonable trade-off to prevent long I/O waits.

As utilization rises, there is a greater chance of system noise, transmission errors and buffer overflows that must be treated by re-transmission of data. The re-transmission increases the volume of data transmitted and can make the buffer congestion worse. **Traffic shaping** is a method to prevent very long I/O latency. Clients are by allocated transmission quotas that prevent average utilization from rising above some safe value, usually about 85%.

Chapter 9

Advanced Architectures

הטכנולוגיות שהוצגו בפרקים 1 עד 8 מציירים תמונה של ה- state of the art עד 1995 בערך. כבר אז מעבד RISC בתוספת branch prediction היה מסוגל לבצע פקודות integer ב- CPI קרוב ל- 1. אין צורך לומר שפיתוח מעבדים לא נגמר לפני 20 שנה. בעיקר המאמץ מאז התרכז בשיפור ביצוע פקודות FP ושיפור ה- CPI לפקודות integer לערכים קטנים מ- 1 (סיום יותר מפקודה אחת בכל מחזור שעון).

Slides 2 – 5 General Outlook

The basic parameters of **performance** are $T = CPI \times IC \times \tau$ where $CPI = CPI^{ideal} + CPI^{stall}$. To improve run time requires reducing CPI, IC, or τ. Because IC grows with software complexity there is little likelihood of making programs smaller. This is especially true because software complexity grows as new methods are introduced to decrease software development time. Also the clock speed cannot be raised significantly because of the fundamental physical limitations seen on slide 34 in chapter 1.

This points to CPI, with 4 possible areas of improvement:

Instruction and thread level parallelism to achieve $CPI^{ideal} < 1$

Reducing instruction dependency stalls to lower $CPI^{stall}_{data\ dependency}$

Reducing cache latency to lower $CPI^{stall}_{cache\ miss}$

Reducing branch stalls to lower $CPI^{stall}_{branch\ penalty}$

There has been some success in all of these areas:

Instruction Level Parallelism (ILP) is improved by providing multiple copies of hardware units in each processor. Multiple instructions begin executing on the same clock cycle. Several instructions finish on every clock cycle so that $CPI^{ideal} < 1$

Reducing instruction dependency stalls is achieved through advanced **compiler rescheduling** and hardware **dynamic rescheduling** (out-of-order execution).

Improvement of floating point performance is achieved by processing FP instructions in parallel in special **vector ALUs**.

As seen in chapter 6 (slide 62), **branch prediction** reduces branch stalls.

Cache latency can be reduced by **prefetching** cache blocks to the CPU based on address prediction.

With **thread level parallelism** on multicore CPUs, code can be divided into independently executing threads further reducing CPI^{ideal} and data dependency.

In order to make sense of these developments, we distinguish four types of parallelism:

Pipelining

Instruction I_{n+1} begins before I_n completes as in DLX so that 1 instruction completes every clock cycle τ. Therefore $R = 1 / \tau$ instructions complete every second.

Superscalar

This type of CPU contains $M > 1$ copies of the pipeline operating in parallel. M instructions start on same clock cycle and ideally M instructions complete on every clock cycle.

Superpipelining

This type of CPU divides the pipeline into smaller stages and each stage performs less work. Less work means a shorter clock cycle $\tau' < \tau$, and therefore a higher clock rate $R' > R$. Now $R' > R$ instructions complete every second.

Multiprocessor

In this type of CPU, usually a multicore chip, $N > 1$ program sections run on N processors in parallel. Therefore the overall program runs in less time.

The DLX is the model of a scalar RISC **pipelined** processor. As we have seen, 1 instruction completes on every clock cycle in the idea case and so $CPI^{ideal} = 1$.

Slides 6 – 9 Vector Processors

Vector processors are special ALUs that implement the SIMD execution model — a single instruction is performed in parallel on multiple data operands. SIMD is very useful for graphics and audio processing where typically one particular operation must be performed on a large array or data structure. Slide 7 shows the execution diagram for an SIMD ALU added to the DLX. A parallel load copies 4 memory words (16 bytes) to register P1 in one CC. Another parallel load brings 4 more words to P2 in 1 CC and a parallel add performs addition in 1 CC (following the load-to-alu stall). A loop over a large data array will run 4 times faster on this SIMD vector processor than a standard sequential loop in the ALU.

Examples of commercial vector processors are listed on slide 8. Slide 9 shows an SIMD example for an Intel x86 CPU. The prefix ASM means that the compiler passes the line of assembly code through as machine code.

Slides 10 – 11 Cache Prefetch

A limitation of the SIMD code example is that the sequential data loads will cause many compulsory cache misses. The short loop shown on slide 10 performs 1024 x 1024 SSE SIMD operations on 16-byte operands — a total of 16 MB of data loads. Since the L1 cache block in a Pentium 4 CPU is 64 bytes which is four 16-byte SSE operands, there will be an L1 miss every 4 data accesses.

Cache prefetch is a method for bringing data blocks to cache before a cache miss. The prefetch instruction initiates copying of 2 blocks to cache on execution while the SSE is executing instructions on the previous cache blocks. The data blocks will be in cache by the time the SSE requires access to them, hiding the latency associated with the cache miss (the cache update latency is not removed — it is "hidden" by being run in parallel to useful work).

Slides 12 – 18 P6 Superscalar with Dynamic Rescheduling

In a superscalar processor, elements of the pipeline are duplicated. The Intel Pentium processor (1993) as built around 2 complete copies of the 5-stage Intel 486 pipeline (1989), but this method had significant limitations. The P6 architecture model instead duplicates only the execution units (ALUs, Load, Store, FPU) and uses dynamic rescheduling to allocate instructions to the ALUs. All Intel x86 processors have used a version of this architecture since the Pentium II (1997). The machine language for x86 processors conforms to the Intel IA-32 ISA, which is a CISC type language. Internally, instructions are converted to a RISC type language for execution. P6 can be described as a RISC ISA with a silicon-based CISC adaptation layer for accepting IA-32.

The stages of execution in the P6 architecture are:

Fetch/Decode

The first stage fetches multiple Intel IA-32 instructions from memory per clock cycle and converts these CISC type instruction to several (1 to 6) RISC-type instructions called **micro-ops**.

Instruction pool

The micro-ops are copied to an instruction pool (a large instruction register) until they are ready for execution. The instruction pool is also called a Reorder Buffer (ROB).

Scheduler

A scheduler performs **out-of-order dynamic rescheduling**. It scans the instruction pool for instructions in that are **ready** — their source operands are already known. The scheduler issues these micro-ops to parallel execution in the various execution units: ALU, FPU, Load, Store. After execution, the finished micro-ops are returned to instruction pool together with their execution results.

Retirement

The finished micro-ops are committed to state (written to registers and memory locations) in their original order in the program (in-order write back). From the programmer's point of view, the program executes and updates architectural state in the original program order.

Dynamic Scheduling

The dynamic scheduling is determined by a process called **scoreboarding**. Each instruction is labeled by a status field:

NR	Not Ready	At least one source operand is not available
R	Ready	All source operands are available
X	Executed	Instruction executed but the destination operand is not available
F	Finished	Instruction was executed and all destination operand(s) are available

Only instructions marked **ready** can be executed, according to a scheduling policy that depends on the hardware organization.

The scoreboard is updated after each clock cycle. Completed instructions are marked **finished** and instructions are marked **ready** as source operands become available.

Slide 14 shows a scoreboard for the standard DLX. Only the EX stage is shown because the pipeline order is well known. In CC1 the first LW is executed and marked X (not F because the loaded operand is not available until the next clock cycle — this is the Load-to-ALU stall in the DLX). The other LW instructions are ready, but cannot be executed because the DLX uses program-order scheduling. On CC2 the first LW is marked Finished and so the ADD is marked Ready. The program continues in this way, executing as expected.

Slide 15 shows a scoreboard for a **DLX that permits dynamics execution**. Instructions are executed as they become ready — the three LW instructions are executed on CC1 to CC3. On CC2 the first ADD becomes Ready and so it is executed in CC4. Since there is no ALU-to-ALU stall the ADD is marked F in CC4. This makes the first SW ready. The SUB R3 instruction became ready next and is marked F in CC5. The program continues in this way — in the optimized scheduling that would have been generated by an optimizing compiler.

Slide 16 shows a scoreboard for a **P6** that permits dynamics execution. Instructions are executed in program order if they are ready. Only 1 load and 1 store is permitted per CC, along

with 2 ALU and 2 FPU instructions. The first LW executes in CC1, followed by another LW and ADD in CC2. The program finishes in 5 CC instead of 12 CC on the standard DLX.

Slide 17 summarizes the P6 execution. Three IA-32 CISC type assembly instructions are interpreted to 9 RISC type micro ops. These micro ops are executed in 5 CC as the scheduler identifies ready instructions and issues them to execution units.

Slide 18 shows the utilization of executions units (EU) in each clock cycle. We see that most EUs are idle in most CCs. The program executes in the minimum number of sequential cycles but with low hardware utilization. In order to obtain higher ILP it is necessary to achieve higher utilization of the execution units, and this requires a larger pool of independent instructions.

One method to find more instructions without data dependencies is through **speculative execution**, which uses branch prediction to find instructions that are likely to run and to execute them before the actual program flow is determined. This is a useful approach when the predictions are good — otherwise the CPU performs useless work that must be cancelled. Another way to increase the pool of independent instructions is through multithreading. Instructions from different threads are inherently independent and so running multiple thread on a single processor can produce many independent instructions.

Slides 19 – 23 Deep Superpipeline

In superpipelining, each pipeline stage is divided into 2 smaller stages. Each new stage does half the work in half the time and so the new stages finish their work in half the time. Thus, it is possible to divide the clock period τ in half which doubles the clock speed. As long as CPI^{ideal} does not change, this means that the run time should divide in half, producing a speedup of 2.

The problem with superpipelining is that some activities cannot be effectively split in half, for example a simple ALU operation. Also, some operations **do not scale in time**. External events take a fixed time so that running a faster CPU clock means that more stall cycles are wasted while waiting for cache updates, branch penalties, page faults, and so on.

As an **example**, the Intel Pentium III was designed with a 10 stage pipeline that ran at a clock speed up to about 1.5 GHz. The successor Pentium 4 as designed with a 20 stage pipeline at a clock speed up to about 4.0 GHz. Based on run time considerations we expect that a 1.5 GHz processor will be 50% faster than the same processor at 1.0 GHz. Since the instruction set is nearly the same for the two processors, it was surprising to find from measurements on SPEC CINT2000 that the 1.5 GHz Pentium-4 was only 20% faster than the 1.0 GHz Pentium III. Analyzing published statistics on the Pentium III one can estimate its total $CPI^{stall} = 0.2$. It is then possible to conclude that the Pentium 4 has $CPI^{stall} = 0.5$, larger than the older CPU. These extra stalls are probably a consequence of doubling the pipeline length.

In order to handle the increased CPI^{stall} Intel introduced **Hyper-Threading**. The hyper-threaded CPU contains two copies of architectural state (registers, stack pointers and program counter) but only one execution core (complete set of EUs). The OS sees two sets of registers (in particular two program counters requesting instructions) and so it looks like two CPUs. The OS assigns threads to both CPU 0 and CPU 1. Since both CPUs issue instructions to a shared execution core, they alternate operations. As long as there is no data stall in either thread, CPU 0 and CPU 1 request instructions on alternate clock cycles. If there is a stall in one thread, the other CPU continues to issue instructions on each clock cycle until the stall ends. Both CPUs keep working on most clock cycles. A stall in a hyper-threaded CPU requires a stall in both threads. Based on estimates from performance statistics, it is possible to estimate that hyper-threading provides a 20% speedup, which is precisely the measured result published by Intel.

Slide 24 Intel Nehalem Micro-Architecture

As an example of the P6 slide 24 shows the Nehalem microarchitecture (2008) developed for the i7 processor.

L1 instruction cache

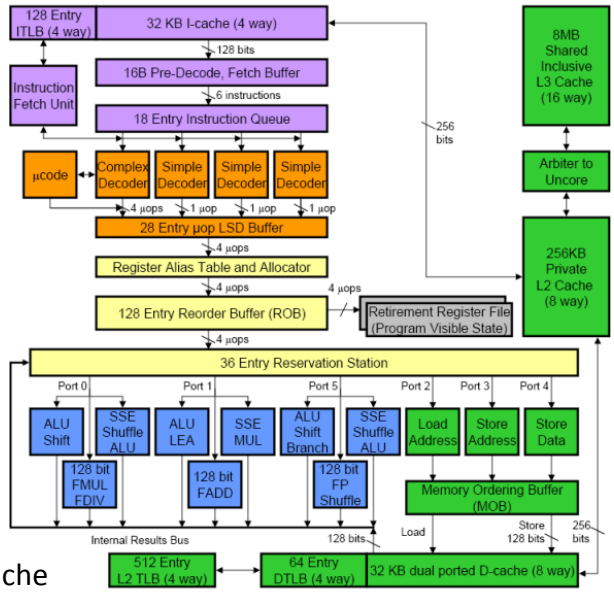
Fetch/decode Unit

LSD Buffer

Instruction Pool (ROB)

Execution Units

L1 data cache



L3 cache

L2 cache

The **LSD buffer** is a Loop Stream Detector. The IA-32 instructions within a loop are not re-fetched and re-decoded in each loop iteration. The decoded instructions within a loop are stored in the LSD in their order of dynamic execution, so that each loop simply recalls the stream of micro ops run in the previous iteration, in the same order.

Slides 24 – 27 Parallel Processing

In parallel processing, instructions are divided among processor cores and executed in parallel (at the same time). Running a program on N-processors divides the number of clock cycles by N for instructions that can be parallelized. **Amdahl's equation** in this case is derived on slide 24.

Slide 25 shows statistics on speedups for 2 and 4 cores. As stated in slide 22, hyper-threading for one CPU provides a speedup of S = 1.2, a 20% improvement. Without hyper-threading we see a speedup of 1.7 for 2 cores and 2.6 for 4 cores. Inverting Amdahl's equation for these values, we find that the proportion of the work that can be parallelized is F_P = 80%.

The difficulty in parallel processing led to a dramatic fall in research in the field after 1995. At that time it was generally believed that during the time required to write a parallel version of a large application with a speedup of 2, the clock speed of available CPUs would also rise by 2, making the investment in parallel processing not worthwhile. Slide 26 shows that the number of research papers in the field grew quickly from 1975 to 1995 and then fell off quickly after 1995. The return to parallel computing is now motivated by lack of other options. Chips with large numbers of cores are much easier to build than chips with faster clock speeds.

Parallel computers can be divided into two basic categories:

Shared memory

The coordination between threads (called **interprocess communication**) is performed through write/read operations to shared memory locations. The system has a single address space shared by multiple cores. Sequential memory coherence is enforced by cache snooping (the interprocessor bus imposes correct write/read order). This creates a cache coherency overhead in processing time.

Message passing system

Interprocess communication is performed by sending and receiving structured messages between cores. These messages send and request data or status information. Sequential coherence is enforced by the message content and message order synchronization. There is no snooping or snooping overhead but message management contributes overhead to processing time. Message passing is generally used for very large distributed systems using the MPI (message passing interface) API.

Slides 28 – 34 Shared Memory

Shared memory parallel computing does not scale well to large systems and is usually used for multicore systems. One or more physical microprocessors each contain private cache, architectural state (registers, including stack pointers and program counter) and execution core (integer ALUs, FPUs, vector processors, memory access). The OS assigns a thread to each processor that runs independently. On a long stall (page fault) a CPU can switch threads.

On multicore CPUs, there is usually a private L1 cache for each core and a shared L2 cache for all cores.

A convenient API for shared memory parallel programming is **Open MP** (supported by gcc 3 and newer versions). Open MP (OMP) supports shared memory applications in C/C++ and Fortran, providing directives for explicit thread-based parallelization. OMP uses the Fork-Join Model in which a master thread (consumer thread) initiates as a single thread and executes sequentially until a parallel construct is encountered. It then forks a team of parallel producer threads. When the parallel section completes, a join operation combines the results of the threads and the master thread continues. Parallel teams can be nested within parallel sections. Slide 31 shows a schematic parallel section within a C program and slide 32 shows a parallel "Hello, worlds" program in which each thread prints "Hello, world from thread number __". The master thread prints the total number of threads.

Slide 33 shows a **parallel for** loop in which 12 loop iterations are split among 3 threads that each perform 4 loop iterations in parallel. This form of parallelization is called **data decomposition**.

Slide 34 shows a **functional decomposition** using the **sections** structure that assigns different programming tasks to each thread.

Slides 35 – 38 Message Passing

Slide 35 shows an example of parallel programming on a message passing system. Each of 4 CPUs performs one part of a scalar product calculation in parallel. Then P0 sends a message to P1 with its result and P2 sends its result to P3. P1 adds the result from P0 to its own, and sends the sum to P3. P3 receives the result from P2 and adds it to its own result. Then P3 receives the sum from P1 and adds it to its sum, forming the final result. The coherence of results is maintained by information in the messages including the sender ID and a timestamp.

Message passing systems define **collective operations**: scatter, gather and reduce. Scatter distributes data among several CPUs and gather collects results from several CPUs. Reduce performs an operation on the results from several CPUs.

A standard API for message passing is the message passing interface (MPI). An example of an MPI Hello, World program is shown on slide 38.

Chapter 10

Real-Life RISC

עד כה הצגנו DLX כמודל למעבד RISC ומעבדי P6 של Intel כדוגמה לשימוש במושגי RISC במחשבים נפוצים בשוק. בפרק הזה נציג מידע כללי על שני משפחות מעבדי RISC גם נפוצים מאוד בשוק, משפחת MIPS שמיועדת בעיקר למערכת משובצות (וגם הבסיס של מודל DLX), ומשפחת ARM שתופסת נתח שוק ענק בטלפונים וטבלטים.

Slides 2 – 6 MIPS Architecture

The main features of the MIPS family are:

RISC Instruction Set Architecture (ISA)

The basic ISA defines registers and instructions common to MIPS processors. MIPS32 defines the 32-bit RISC ISA. This is the basis for the DLX model. MIPS64 defines the 64-bit RISC ISA. It is binary compatible with MIPS32 —binary code compiled for MIPS32 will run on MIPS64.

MIPS core descriptions

Product guides for individual MIPS cores define device-dependent implementation details, such as pipeline organization, I/O organization, control registers, interfaces, programming models and so on.

Applications

MIPS core specifications are typically licensed to OEMs (original equipment manufacturers) for use in specific products. The core design is then implemented in embedded systems. MIPS-based PCs are used in China.

The **MIPS32** registers and instruction formats are shown in slide 3. These definitions are nearly identical to the DLX. In addition, MIPS32 defines logical extensions called **coprocessors**. These include a **control coprocessor** for management of virtual addressing, an interface to an **FPU**, and an open interface for **device-specific** hardware implementations. The MIPS instruction set is familiar from DLX. Slide 5 shows some convenient instructions not defined in the DLX instruction set.

The **MIPS64** register set is identical to the MIPS32 register set except that registers hold 64-bit values. All MIPS32 instructions run on MIPS64 — some instructions act on the lower 32 bits in a register and some act on the full 64-bit values. Addresses in memory are formed from a 64-bit pointer stored in a register and a 16-bit immediate value from the instruction.

Slides 7 – 12 ARM ISA

ARM (Advanced RISC Machine) is a processor developed by ARM Limited, which does not manufacture chips but licenses ISA designs for implementation by developers. ARM is the most widely used 32-bit RISC ISA with over 50 billion ARM processors used in phones, tablets, games, computer peripherals and other equipment. Over 98 percent of mobile phones use ARM processors.

The main features of the ARM family are:

Data Types

Byte 8 bits, Halfword 16 bits (in ARMv4 and higher) and Word (32 bits)

Standard RISC features

Load/store architecture, large uniform register file, simple addressing modes, uniform and fixed-length instruction fields and a scalar in-order pipeline.

Additional ARM architectural features

The arm defines special Shift and ALU operations, auto-increment and auto-decrement addressing modes for loops, load and store multiple instructions, conditional execution (predication) of most instructions (these cancel instructions on certain condition flags) can replace control hazards in forward jumps

The ARM is produced in 8 different versions as shown on slide 9. In addition, several classes of extension features are defined:

Jazelle — permits the ARM to run Java bytecode with no further compliations

Thumb — a reduced 16-bit version of the ISA for small devices

Debugger, ICE — testing features for embedded systems

Multiplier (64-bit result) — permits high precision integer multiplication

DSP Enhancement — defines and instructions set for common operations in digital signal processing (DSP)

Embedded systems are devices — as small as a USB flash drive or as large as a medical MRI machine — that uses one or more microprocessors but is not a general-purpose programmable computer. CPUs intended for embedded processor applications are usually chosen to be the simplest and least expensive device that meets the application requirements.

The ARM ISA defines 16 architectural registers of 32-bit width (the ARMv8 also supports 64-bit registers). These are named r0 to r15, as shown on slide 10. Registers r11 to r15 are special purpose registers, defined with similar roles to the corresponding registers in the VAX architecture —Frame Pointer, intra-procedure-call scratch register, Stack Pointer, Link Register used to return from function calls, and the Program Counter.

ARM processors operate in 7 different operating modes shown on slide 11. Besides user and supervisor modes, there are dedicated modes for DMA, IRQ, virtual memory, coprocessor emulation, and OS mode. Depending on the operating mode in which an instruction runs, it will see one of several copies of registers r8 to r14. There is a 32-bit status register visible in all operating modes. Slide 12 shows the registers visible to the instructions depending on the operating mode. For example, when switching to supervisor mode, the program counter is swapped so that no jump instruction is required to start executing OS code.

Slides 13 – 21 ARM Instructions

The ARM has several instruction sets.

The **standard instruction set** is the ARM 32-bit instructions, which are aligned on 32-bit boundaries in memory (the lowest 2 bits of the PC are always 0).

The **Thumb** instruction set contains 16-bit instructions that are 1-to-1 mapped to the 32-bit instructions. These are shortened versions with restricted options and implicit operands. For

example, **add dest, src1, src2** is reduced to the form **add dest, src**. The instructions are aligned on 16-bit boundaries (the lowest bit of the PC is always 0).

The **Jazelle** instruction set executes Java bytecode directly. The ARM reads four 8-bit instructions per instruction fetch.

The ARM permits conditional execution of most instructions (predication). Slide 14 shows an example in which a conditional branch is replaced with conditional execution. This implementation of the **`if-then-else`** control block

```
r3-- ;
if ( r3 == 0 ) then r0 = r1 + r1 else r0 = r1 – r1 ;
r4 = r5 + r6 ;
```

removes the branch stalls caused by BEQ and B.

```
     SUB r3, r3, #1                          SUBS r3, r3, #1
     CMP r3, #0
     BEQ L1           ; branch on r3 == 0
     ADD r0, r1, r2   ; skipped if r3 == 0   ADDNE r0, r1, r2 ; ADD if r3 != 0
     B L2             ; jump to L2
L1:  SUB r0, r1, r2   ; skipped if r3 != 0   SUBEQ r0, r1, r2 ; SUB if r3 == 0
L2:  ADD r4, r5, r6                          ADD r4, r5, r6
```

The basic instruction set is similar to the DLX integer instructions and is shown on slide 15.

Slide 16 shows the shift and rotate instructions which permit more complex operations. For example, the expression a = b + 4 × c can be performed by left-shifting c to bits and then adding to b.

Slides 17 and 18 describes the FPU and vector extensions defined for ARM.

Slides 18 and 19 describe instructions designed for DSP (digital signal processing).

Slide 21 shows various performance improvements in the ARM with the DSP instruction set on appropriate applications.

Slide 22 Apple iPhone 5

The Apple iPhone 5 uses the ARMv7 core and the iPhone 6 uses the ARMv8. The Apple A6 is a SOC (system on a chip) designed for the iPhone, containing the ARM CPU and associated devices. Additional components of the iPhone 5 — networking and interface controllers — are shown on slide 22.

Slides 23 – 28 WinARM

WinARM is a cross-compiler for the ARM ISA that permits development of applications for ARM in C/C++ on Windows platforms. The compiler includes extensive documentation and tools such as GNU GCC compiler, GNU-Utils for compiler/linker, ARM header-files and sample applications with source-code. The compiler permits compilation to assembly code that enables study of ARM assembly language and the ARM programming process.

Slide 24 shows a simple C program and the compiler output.

This output is analyzed in detail on slides 25 – 26. The section labeled .main (slide 25) sets up the data frame and jumps to .L2, which sets up the while() control block. The ALU operations are in section .L3 (slide 26).

Another more complex example is given on slides 27 – 28.