TALLINN UNIVERSITY
of TECHNOLOGY
Department of Computer Engineering

MICROPROCESSOR SYSTEMS I

Microprocessor Systems Architecture

IAY 0021

Lecture Notes

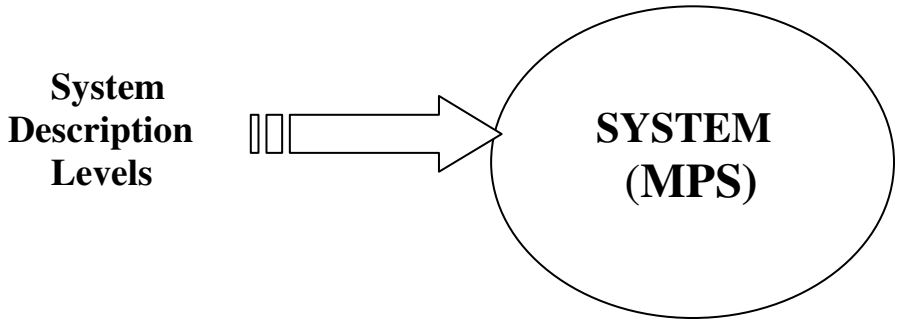
Arvo Toomsalu

Tallinn

Contents

MICROPROCESSOR SYSTEM ARCHITECTURE

System Description (AIR Principle)



I Architecture [A] Level

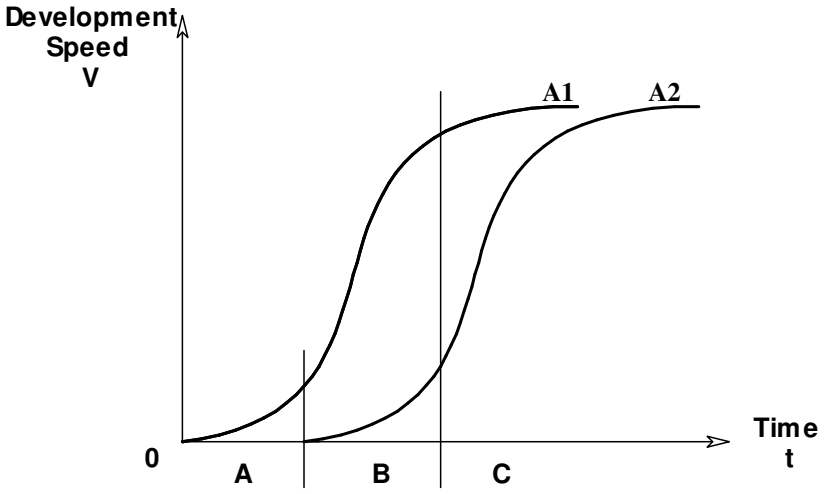
***WHAT*?**

System extraordinariness, singularity (some quality of a thing by which it is distinguished from all, or most, others).

Specific **software/hardware *interface***, which includes:

Instruction set;
Memory management and protection;
Interrupts;
Floating-point standard, etc.

Architecture development life-cycle



A1, **A2** – different architectures of a system

Forces on the microprocessor system (computer) architecture are:

1. Applications;
2. Technology;
3. Operating systems;
4. Programming languages;
5. History.

***Architecture* is the iterative searching of the *possible designs* at all levels of microprocessor system.**

II Implementation [I] Level

***HOW*?**

Logical structure, **organization** or ***microarchitecture***.

Number and location functions;
Pipeline configuration;
I/O system organization;
Location and configuration of caches, etc.

► **Microprocessor Families**

III Realization [R] Level

***WHICH* and *WHERE*?**

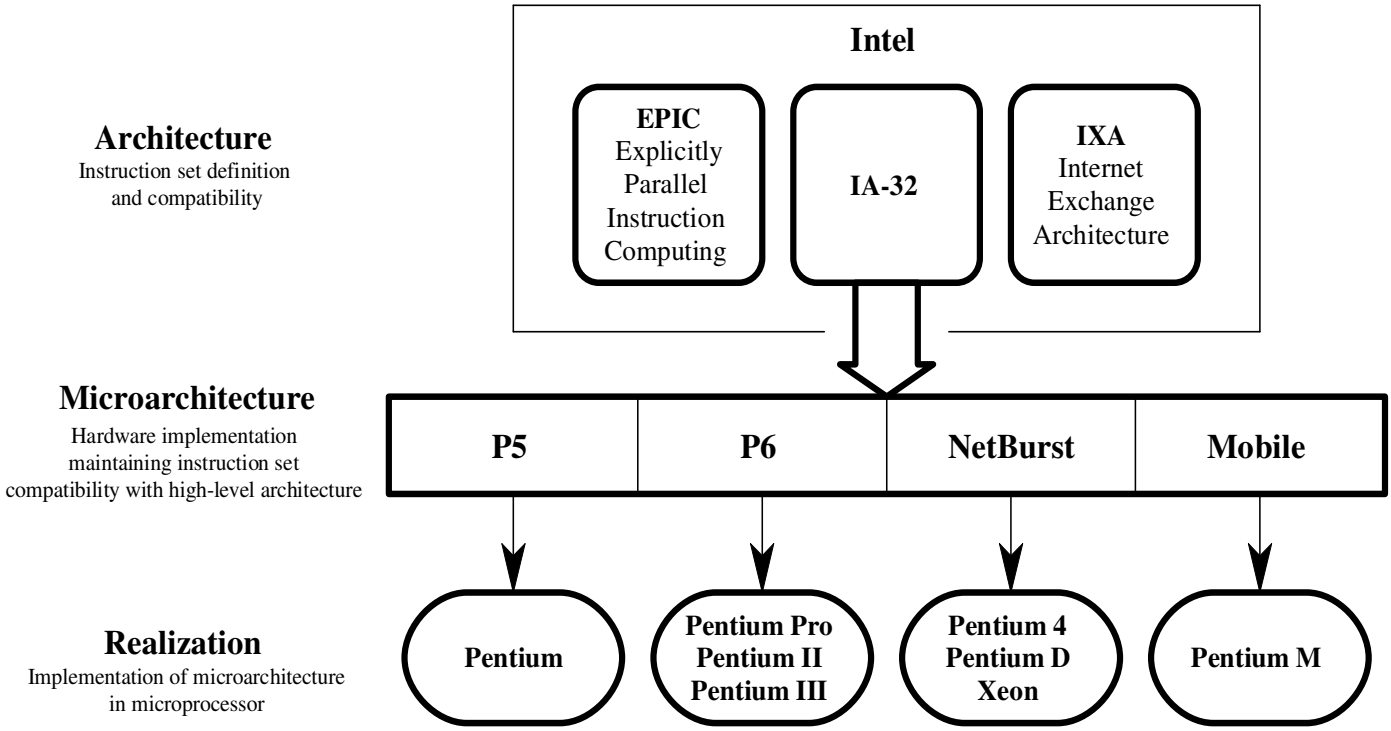
Architecture *versus* Microarchitecture

Architecture is used to describe an abstract requirement specification.

It refers to the instruction set, registers, and memory data-resident structures that are public to a programmer.

Microarchitecture is used to describe the specific composition of elements that realizes architecture in a specific design.

It refers to an implementation of (micro)processor architecture in silicon.



AIR principle uses only three system description levels, but for more precise description there is used more description levels, as for:

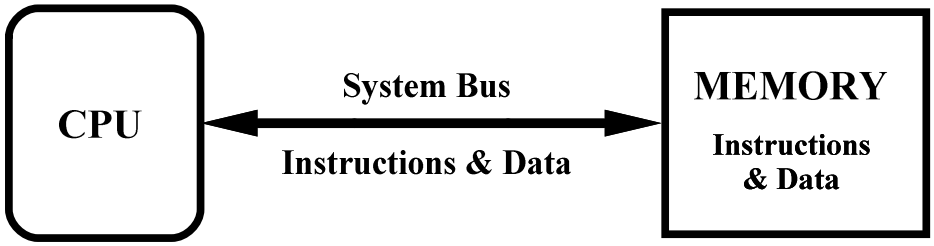
1. **Processor-memory level**, at which architecture of a microprocessor system is described.
2. **Instruction set level**, at which the function of each instruction is described.
3. **Register transfer level** (**RTL**), at which the system's hardware structure is described in details.
4. **Logic gate level**, at which the hardware components are described as standard logic elements and flip-flops.
5. **Circuit level,** at which the hardware components internal structure is opened and they described by active (transistors) and passive elements.
6. **Integrated circuit masks level,** at which the silicon structures and their layout, that used to implements the system components, are shown.

Moving from the first description level to the last, we can see how the* behavior *of the microprocessor system is transformed into the real hardware and software structure.

Architecture Taxonomies

Classical Architectures

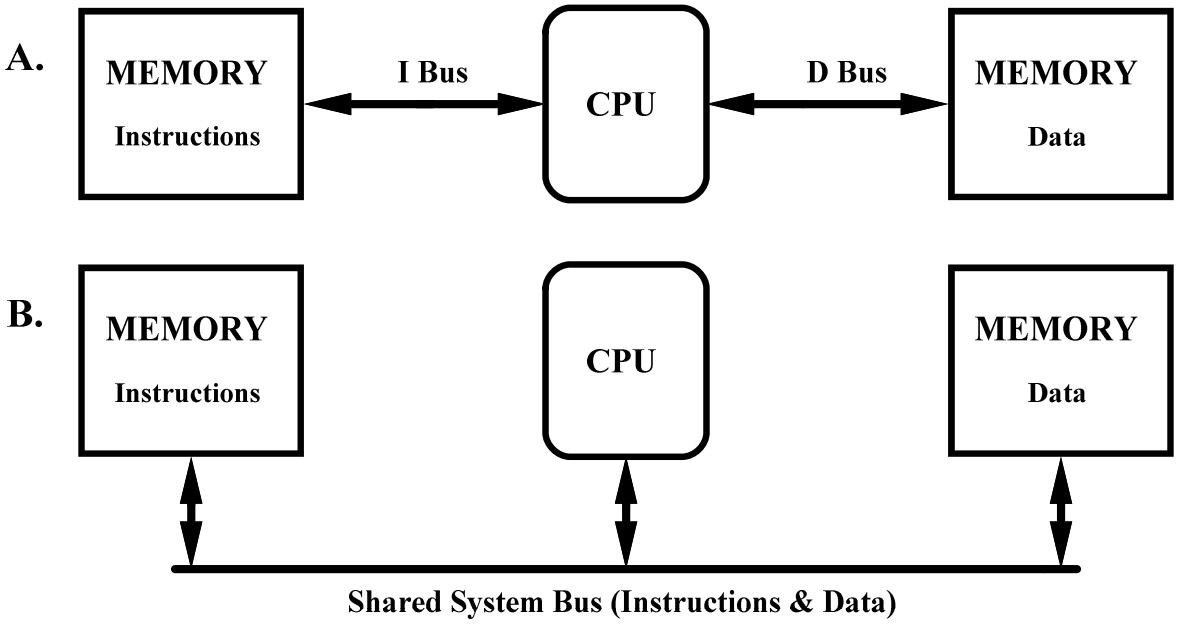
Princeton or **von Neumann Architecture**



Characteristic features:

1. Integral processing unit;
2. United data and instructions memory;
3. United bus between CPU and memory;
4. Centralized control;
5. Low-level programming languages;
6. Memory linear addressing.

Harvard Architecture



Characteristic features:

1. Instruction and data memories occupy different address spaces;
2. Instruction and data memories have separate buses to the CPU (**A**);
3. Instruction and data memories can be implemented in different ways.

Modified Harvard Architecture and Flynn`s Classification

A **pure Harvard architecture** has the **disadvantage** that mechanisms must be provided to separately load the program to be executed into instruction memory and data to data memory.
Modern Harvard architecture computers often use a **read-only technology** for the **instruction memory** and **read-write technology** for the **data memory**. This allows the computer to begin execution of a pre-loaded program as soon as power is applied.
The Modified Harvard Architecture is relegated to niche applications – microcontrollers and digital signal processors.

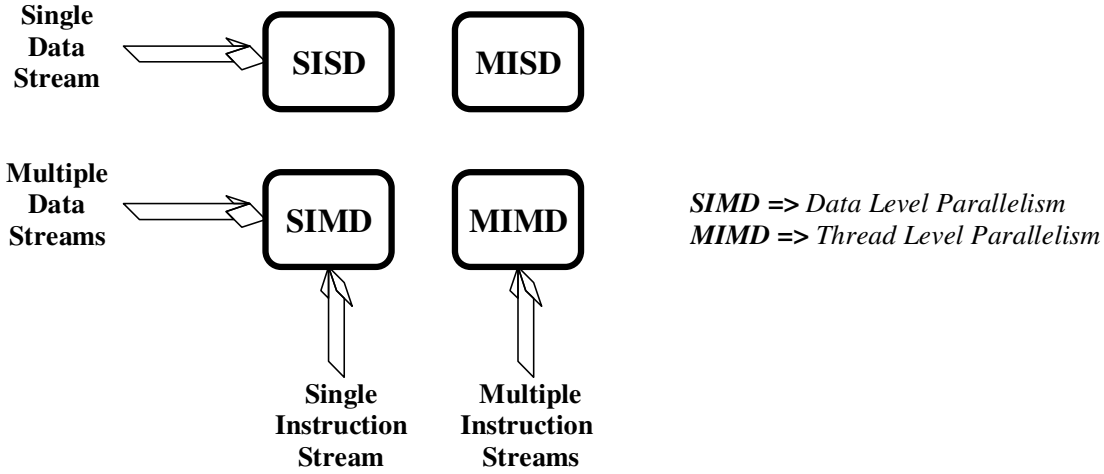
> **The principal advantage of the Harvard architecture** –
> simultaneous access to instruction and data memories
> **has been nullified by modern cache systems**.

Multi-microprocessor system - a system that use more than one microprocessor to perform a desired application.

Multiprocessor systems were designed for at least one of two reasons:

1. Speed-up program;
2. Fault tolerance.

The **classification of computer architectures** based on notions of instruction and data streams or **Michael Flynn`s** (1972)) **classification.** Flynn`s classification stresses the architectural relationship at the memory-processor level. Other architectural levels are overlooked.



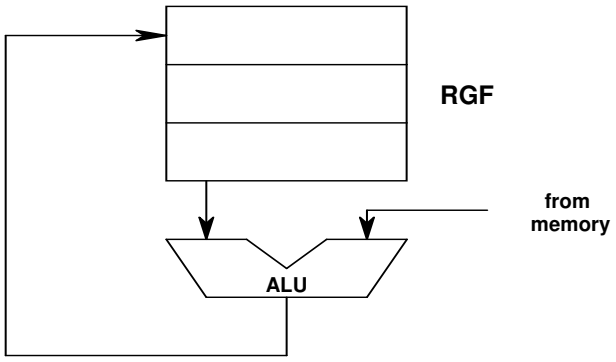
Flynn-Johnson Classification

Instruction Set Architecture

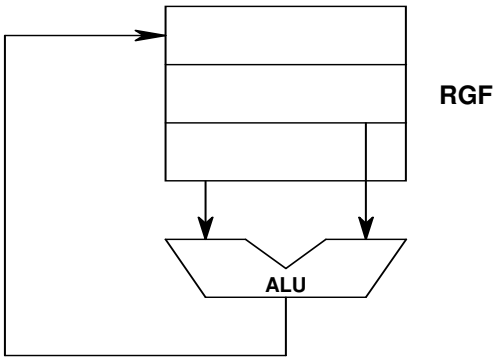
In the past, the term ***computer architecture*** referred only to **instruction set** design.
The term ***instruction set architecture*** (**ISA**) refers to the actual programmer-visible instruction set.
ISA enables different implementations of the same architecture but it may prevent using new innovations.

Classification instruction set architectures (by J. L. Hennessy & D. A. Patterson)

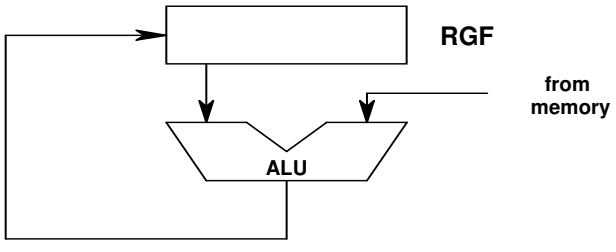
1. **Register-memory architecture**
2. **Register-register or load-store architecture**
3. **Accumulator architecture**
4. **Stack architecture**
5. **Memory-memory architecture**



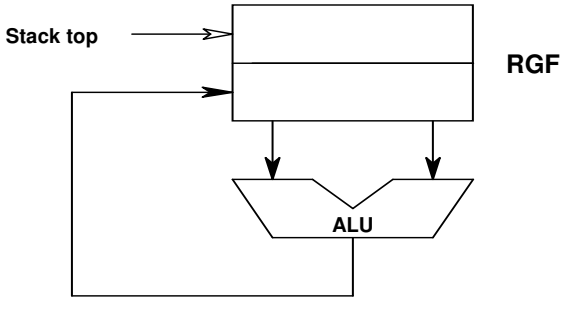
Register-memory architecture



Register-register/load architecture



Accumulator architecture



Stack architecture

Architectures Taxonomy by Milutinovic

(1989)

1. **Control -driven** (**control-flow**) **architectures**

   The instruction sequence guides the processing activity.
   - **a.** Reduced instruction set computers (**RISC**);
   - **b.** Complex instruction set computers (**CISC**);
   - **c.** High-level language architectures (**HLL**).

2. **Data-driven** (**data-flow**) **architectures**

   The processing activity is controlled by the readiness of data (experimental).

3. **Demand-driven** (**reduction**) **architectures**

   An instruction is enabled for execution when its results are required as operands for another instruction that has been already enabled for execution (experimental).

Architecture Standardization and Open-system Concept

An **open-system** is a **collection interfaces**, **protocols**, and **data formats** that are based on the commonly available, universally accepted standards providing software portability, system interaction, and scalability.

- **Portability** is a property of a source code program to be executed on different hardware platforms under different operating systems.
- **System interaction** is the ability of systems to exchange information automatically, recognizing the format and semantics of data.
- **Scalability** is a property of a program to be executed using different resources with the efficiency rate being proportional to the resources available.

Summary

Comparison Different ISA Structures

Number of memory accesses	Maximum number of operands allowed	Architecture	Examples
0	**3**	**register-register**	Alpha, Power PC
1	**2**	**register-memory**	IBM 360/370, Intel 80x86
2	**2**	**memory-memory**	VAX
3	**3**	**memory-memory**	VAX

Advantages and Disadvantages Different ISA Structures

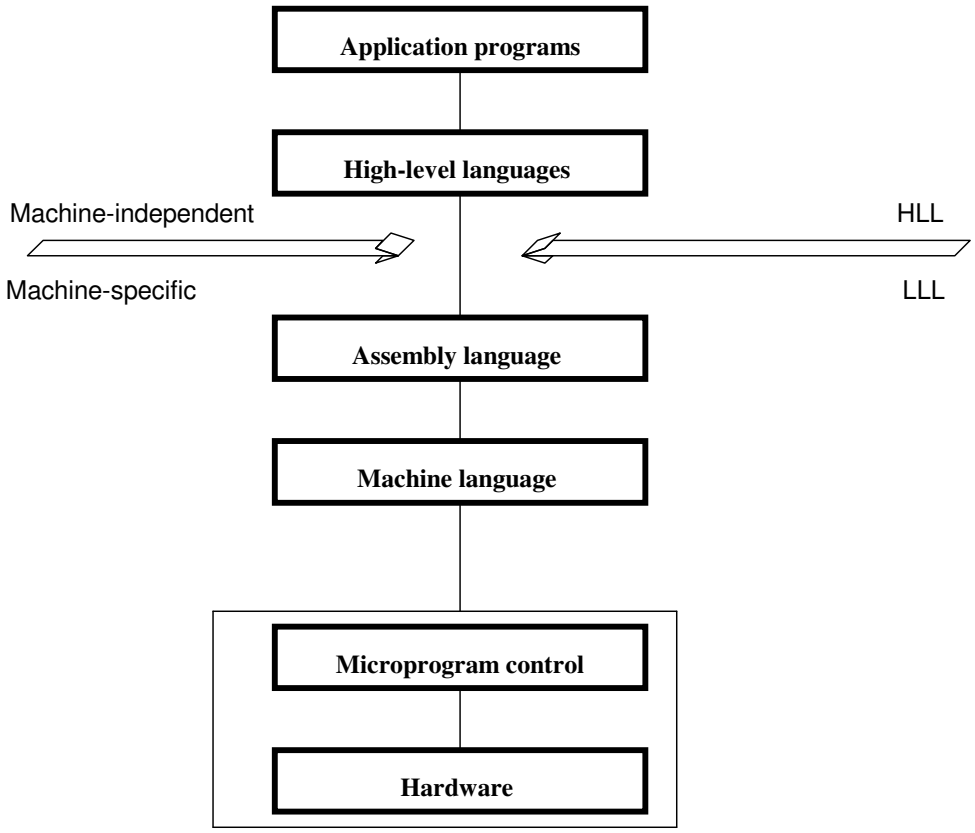
Architecture	Advantages	Disadvantages
register-register (0,3)*	**1**. Simple, fixed-length instruction decoding. **2**. Simple code generation model. **3**. Instructions take similar number of clocks to execute.	**1.** Higher instruction count than architectures with memory references in instructions. **2.** More instructions, lower instruction density - larger programs.
register-memory (1,2)	**1**. Data can be accessed without a separate load instruction first. **2**. Instruction format tends to be easy to encode and offers good density.	**1.** Operands are not equivalent since a source operand in a binary operation is destroyed. **2**. Encoding a register number and a memory address in each instruction may restrict the number of registers. **3.** Clock per instruction varies by operand location.
memory-memory (2,2) or (3,3)	Most compact, and doesn't waste registers for temporaries.	**1.** Large variation in instruction size. **2.** Large variation in work per instruction. **3.** Memory accesses create memory bottleneck. **4.** Not used.

- In the notation (**m,n**) means **m** memory operands and **n** total operands.

Operating System Management Functions

Operating system (OS) controls the execution of programs on a processor and manages the processor's resources.

Programmer's View of a Computer System



The main functions of OS are process (task) scheduling //*plaanimine*// and memory management //*mäluhaldus*//.
The OS determines which process should run at any given time.

- The main types of process scheduling are:

a. Long-term scheduling;
b. Medium-term scheduling;
c. Short-term scheduling;
d. I/O scheduling.

During the lifetime of a process, its state will change a number time:

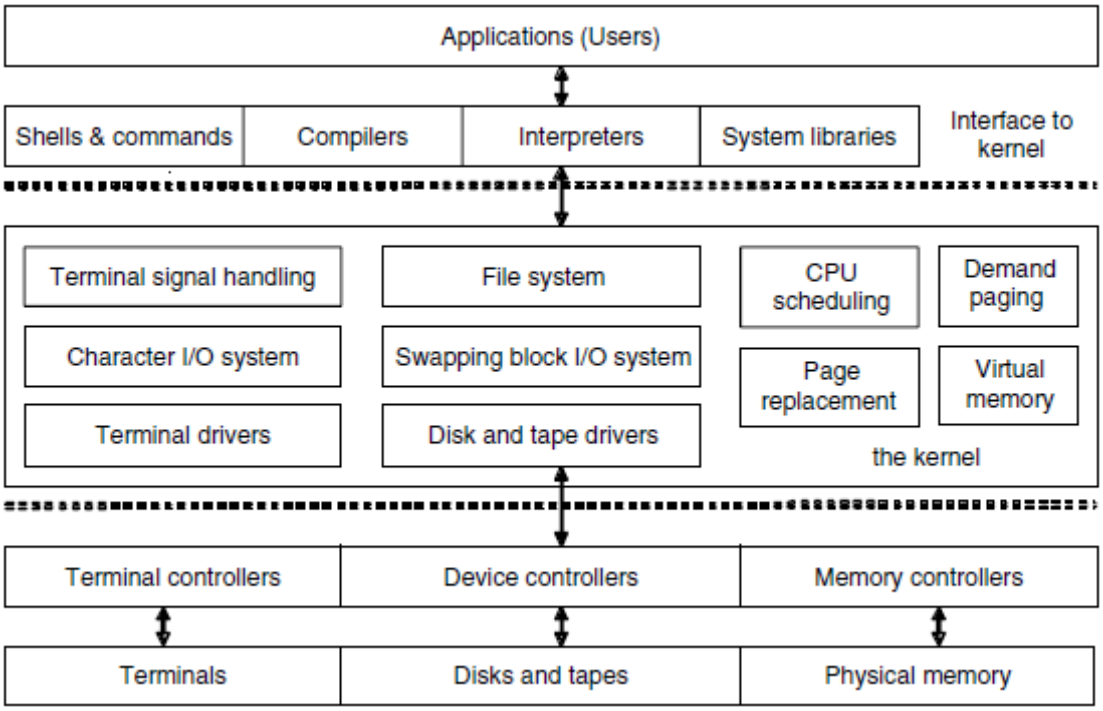
New > Ready > Running > Waiting > Halted.

- The main tasks in memory management are swapping //*saalimine*// and partitioning //*sektsioneerimine*// (fixed-size partitions or variable-size partitions).

Generic Architecture of Operating System

The kernel is the basic set of computing functions needed for an operating system. The kernel contains the interrupt handler, the task manager, and the inter-process communication manager.
It may also contain the virtual memory manager and the network subsystem manager.
The services provided by an operating system can be organized in categories:

1. Task control,
2. File manipulation,
3. Device control,
4. Information maintenance.



Architecture of UNIX

Process and **Thread**

The basic unit software that the OS deals with is either a process (heavyweight process) or a thread (lightweight process) *//lõim, haru//*, depending on the OS.
A process is software that performs some action and can be controlled (by user, by other application, by the OS).

A process may start threads or other processes, but thread cannot start a process.

The information needed to keep track of a process when switching is kept in a data package is called a process control block (PCB). The PCB contains information about:

1. An ID number that identifies the process;
2. The priority of the process;
3. Pointers to the locations in the program and its data where processing last occurred;
4. Pointers to the upper and lower bounds of memory required for the process;
5. States of flags and switches;
6. A list of files opened by the process;
7. Registers content;
8. The status of all I/O devices needed by the process.

In many OSs processes can subdivide themselves into more easily managed subunits called threads.

A thread is a portion of a process that can be scheduled and executed independently.

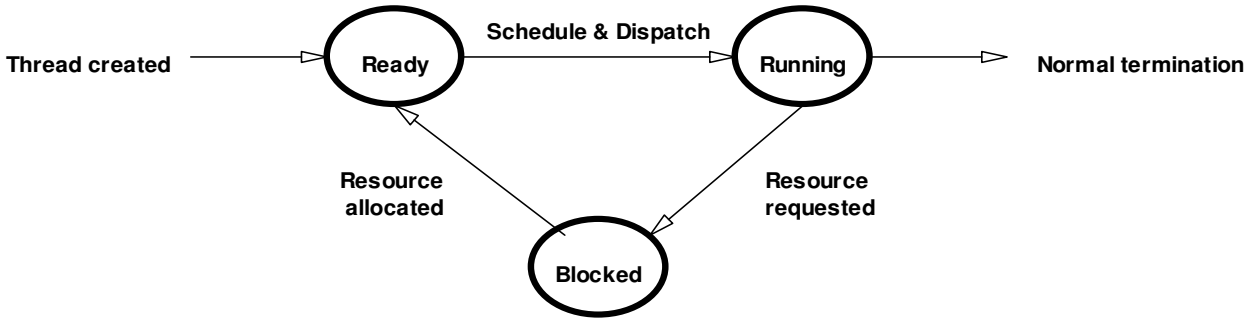
The thread can deal with all the CPU-intensive work of a normal process, but generally does not deal with the various types of I/O operations and does not establish requiring the extensive process control block of a regular process.

Thread levels

- **User-level threads** or **ULT** (thread libraries)
- **Kernel-level threads** or **KLT** (system calls)

The OS keeps track of thread-specific information in a thread control block (TCB).
An active thread can be only one of the following states:

1. Ready,
2. Running,
3. Blocked.



a. Thread in ready state is waiting for access to a CPU.
Usually there are many threads in ready state.
The act of giving control of a CPU to a ready state process is called dispatching.
The decision making process used by the OS to dertermine which ready thread moves to the running state is called scheduling.
The OS vary in their scheduling methods (preemptive scheduling, priority-based scheduling,

real-time scheduling).

b. Once dispatched, a thread has entered the running state and retains control of the CPU until one of the following events:

1. The thread or its parent process terminates normally;
2. An interrupt occurs.

c. The most common method is for the process or thread to execute an exit service call. The exit service call triggers a software interrupt.

d. When any interrupt is received, the CPU automatically suspends the currently executing thread, pushes current register values onto the stack, and transfers control to the OS master interrupt handler. The suspended thread remains on the stack until interrupt processing is completed.

e. During this period of time the thread is blocked. Once the interrupt has been processed, the OS can do:

1. Leave the suspended thread in the blocked state,
2. Move it to the ready state,
3. Return it to the running state.

Thread advantages over process

1. Less time to create a new thread (newly created thread uses the current address space);
2. Less time to terminate a thread;
3. Less time toswitch between two threads within the same process;
4. Less communication overheads (the threads share everything, particularly the address space).

A process or program that divides itself into multiple threads is called a **multithreaded.**

OSs capbilities
MS-DOS supports a **single user process** and a **single thread**;
UNIX supports **multiple user processes** but only **one thread per process**;
Solaris supports **multiple threads (multithreading)**.

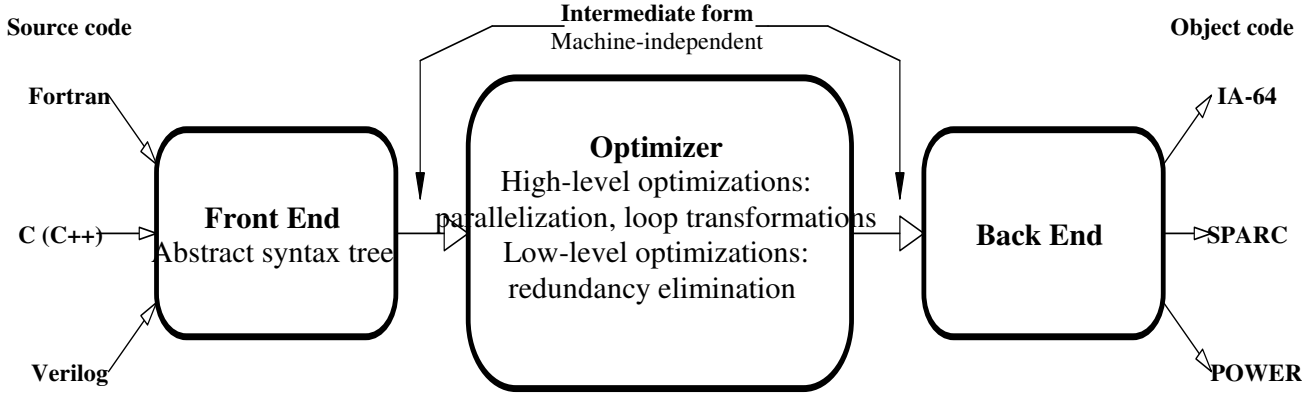
Compiler

Compiler is a program that translaters a higher programming language into machine language.

Compiler

1. Minimizeses the number of operations;
2. Replaces expensive operations with simpler ones;
3. Minimizeses cache misses;
4. Minimizeses object code size.

Compilers usually decompose programs into their ***basic blocks*** (the first step in the analysis process).



The general translation and interpretation steps that have to be taken in order to execute a program written in a HLL are:

- Frontend compilation;
- Determine dependences and produce data and control dependence graphs;
- Partitioning the graph;
- Bind partitions to nodes;
- Bind operands to locations;
- Bind operands to time slots;
- Bind operations to FUs;
- Bind transports to buses;
- Exacurte the operations and perform the transports.

The compiler affects significantly the performance of a microprocessor system. The modern optimizing compiler consists of a number of passes. Three classes of optimizations are performed:

1. Local optimizations within a single basic block
   Global optimizations work across multiple basic blocks.
2. Global register allocation allocates variables to registers for regions of the code.

PERFORMANCE MEASURING

Performance is a manner or **quality of functioning**, the **speed of operation**. It is a **key to the effectiveness of the MPS**.

- The **computer user** is interested in reducing:
  response time or execution time– the time between the start and the completion
  ($\mathbf{t_e}$) of an event (more jobs in an hour).

- The **manager of a large data processing center** is interested in increasing:
  throughput – the total amount of work done in a given time.

In comparing design alternatives often is related to the **performance** (**P**). In comparing systems **X** and **Y** we can state, that: "Y *is* ***n*** *times faster than X*" if:

$$\mathbf{n = t_{eX} / t_{eY}}$$

The **execution time** is reciprocal to the **performance**, i.e.

$$\mathbf{t_e = 1 / P,}$$
$$\mathbf{P = 1 / t_e,}$$
$$\mathbf{n = (1 / P_X) / (1 / P_Y) = P_Y / P_X}\text{ , or}$$
$$\mathbf{P_Y = n \times P_X}$$

The number of tasks completed per unit time on system ***Y*** *is* ***n*** *times the number completed on* ***X****.*

The key measurement of performance is *TIME*

Execution time can be measured in several ways:

1. **Elapsed time** (**wall-clock time**) - the total time for a complete task.
2. **CPU time -** the time the processor is computing and not waiting for I/O.

Performance Measurement and Evaluation

When we measure, then we determine the value of some parameter. So obtained data is meaningless until compared to some designed quantities, i.e. until evaluated. Evaluation of MPS is of vital importance in the selection of system, the design applications and analysis of existing systems. Performance evaluation purposes are:

1. Selection evaluation in which the evaluator plans to include performance as a major criterion in the decision to obtain a particular system from a vendor is the most frequent case. The procedure includes both hardware and software evaluation.
2. Performance projection is oriented toward designing a new system, either a hardware component or a software package. The goal here is to estimate the performance of a system that

does not yet exist. A secondary application is the projection of the performance of a given system on a new workload //*koormus*//.
For performance projection is used different performance modeling methods – simulation and analytical modeling.
3. Performance monitoring provides data on the actual performance of an existing system.

No single measure of performance can give a truly accurate measure of MPS's performance!
In real terms it can be defined only together with the *user program*.

The most commonly quoted measure of performance is peak performance – the maximum rate at which some operation can be executed.
Peak performance is based on the clock rate of the processor and on the minimum number of cycles (clocks) per instruction (CPI) attainable.
The peak performance ignores the fact that there is a mix of CPI values that depend on the instruction set, the cache behavior and the proportions in which instructions are executed.
The clock rate is the second most often quoted performance figure.
It is only a bit more meaningful than peak performance.

The CPU Performance Equations

$$\mathbf{t_{CPU} = t_e}$$

$$\mathbf{t_{CPU}} = \frac{Seconds}{\mathrm{Pr}\,ogram}$$

$$\mathbf{P_{CPU} = t_{CPU}^{-1} = t_e^{-1}}$$

$$\mathbf{t_{CPU}} = \underset{Architecture}{\mathbf{(instruction\ count)}} \times \underset{Implementation}{\mathbf{(clock\ cycles\ per\ instruction)}} \times \underset{Realization}{\mathbf{(cycle\ time)}}$$

$$\mathbf{P_{CPU} = (CPI \times IC)^{-1} \times clock\ rate}$$

$$\mathbf{\{P_{CPU} = (} \sum_i CPI_i \times IC_i \mathbf{)^{-1} \times clock\ rate\}} \text{ or}$$

$$\mathbf{P_{CPU} = (IPC \times clock\ rate) \times IC^{-1}},$$

$$\mathbf{CPI_{total}} = \sum_{i=1}^{n} CPI_i \times \left(\frac{IC_i}{Instruction - count}\right) \text{ , where}$$

$\frac{IC_i}{Instruction - count}$ is a **frequency** of **i-**type instructions.

CPI - clock cycles per instruction; **IPC -** instructions per clock cycle (per clock); **IPC = CPI^{-1};**
IC – the instruction count (number of instructions executed).
t_{CLK} - cycle time ; **f –** clock rate (**f = t_{CLK}^{-1});**

$$P_{CPU} = IPC \times IC^{-1} \times f$$
$$P_{CPU} = CPI^{-1} \times IC^{-1} \times f$$

- The **clock rate** (**f**) depends on the technology used to implement the processor.
- The **average IPC** depends on processor microarchitecture.
- The **instruction count** (**IC**) depends on instruction set architecture, compiler, and the operating system.

Average IPC reflects the average instructions throughput achieved by the processor and is a key measure of microarchitecture effectiveness.

The use of CPI was popular during the days of scalar pipelined processors.
For superscalar processors, it becomes more convenient to use IPC.

An ideal scalar processor pipeline would execute instructions at rate of one per cycle, resulting in a core CPI = 1. This CPI assumes that all memory references are hits in the cache cycles.

The **actual CPI** of the **processor is** equal to:

$$CPI = CPI^{core} + MCPI, \text{ where}$$

MCPI is the memory-cycles-per instruction.

Computer Performance Metrics

- **MIPS** – **M**illion **I**nstructions **p**er **S**econd;
- **BIPS, BOPS, GIPS, GOPS** – **B**illions (**G**iga) (10^9) of **I**nstructions **/O**perations **p**er **S**econd;
- **TOPS** – **T**rillions or Tera (10^{12}) **O**perations **p**er **S**econd;
- **FLOPS** – **Fl**oating **P**oint **O**perations **p**er **S**econd;
- **CPS** or **COPS** – **C**onnections **p**er **S**econd.

Remark

1. **MIPS = Clock rate (f) / (CPI × 10^6)**
2. ***MIPS are misleading*** because the work done by an instruction varies.
3. ***FLOPS, LIPS, COPS are more meaningful*** because they specify a particular operation, but they are often calculated.

Rating Technology

The application ratings //*reiting, tootlus*// are based on a comparison of workload run times between the system being tested and a fixed calibration platform.
A rating of 100 indicates that the system has performance equal to that of the calibration platform, 200 indicates twice the performance of the calibration platform.

Performance-analysis Technologies

- **Bottleneck** //*kitsaskoht*// - the bottleneck is the resource with the lowest maximum performance.
- **Latency** //*latentsusaeg*// - delay between start and end of operation
- **Response time //***reaktsiooniaeg***// -** delay between request and response.
- **Throughput** //*läbilaskevõime, jõudlus*// – the amount of work done per unit time, **(Capacity)** or the rate at which new results arrive.
- **Bandwidth** //*ribalaius*// - used to describe the theoretical maximum throughput of a data link or other device.
- **Availability** //*käideldavus*// - fraction of time that a resource is available for use.
- **Utilization** //*kasutatavus*// - fraction of time a resource is busy. Metric is used only inside a system.
- **Accuracy** //*täpsus*// - indicates how closely performance measurement results obtained when using the technique correspond to the results that would have been obtained on a real system.

Factors which Determine the MPS Performance

1. Algorithm and data set (size, length of execution, pattern of access);
2. Compiler (efficiency of code generated , ability to optimize access patterns);
3. Instruction set (how many instructions does it take to encode the program);
4. Available operations (floating point support, multimedia instructions);
5. Operating system (timeout period and cost, other processes);
6. Clock rate;
7. CPI (IPC);
8. Memory system organization (memory hierarchy, cache system, hit rate, miss penalty);
9. Technology, etc.

Benchmarks

Benchmark – program or program fragment and related data inputs that are executed to test

the performance of hardware component, a group of hardware or software components, or an entire computer system or network.

The goal of benchmarking is to predict the performance of a workload on a particular platform.

Workload – the amount of work which a computer has to do.
There are two ways in which workloads can be constructed:

1. Model workloads
2. Synthetic workloads

Benchmark Types

Benchmarks can be designed to focus on performance of individual components (CPU, cache, memory, I/O, graphics, and network resources), or on application (or system) performance:

1. **Component benchmarks** (**microbenchmarks**)

They measure the performance of a single aspect of a system.

2. **Application benchmarks (macrobenchmarks)**

They evaluate the performance of many system components working together to produce the throughput that a user perceives.

Real programs (representative or real workload);
Kernels (representative program fragments);
Mixes (instruction frequency of occurrence);
Synthetic benchmarks (programs intended to provide a *specific mix*).

Benchmark sources

1. Vendor specific benchmarks (**iCOMP**)
2. User organizations who share their benchmarks (**Linpack**)
3. Industry organizations (**SPEC, EEMBC**)
   (**EEMBC** – Embedded Microprocessor Benchmark Consortium)

Benchmark Examples

1. **MIPS** (**M**illions of **I**nstructions **P**er **S**econd).

The MIPS is related closely to the clock rate of the processor. Using MIPS as a tool to determine relative performance of different computers is very dangerous, so:

MIPS are only useful for comparison between two processors from the same vendor that support the same instruction set with the same compilers.

2. Linpack

The Linpack benchmark uses a set of high performance library routines for linear algebra.
These routines are used to measure the time it takes to solve a dense system of linear equations.
The benchmark reports average **megaflops rates** by dividing the total number of floating-point operations by time. There are several versions of the Linpack benchmark, differing in size, numerical precision and ground rules.

3. Whetstone

The first **synthetic benchmark** program that was designed for performance testing. The program was designed primarily **to measure floating-point performance**. Performance is quoted in **MWIPS** (millions of Whetstone instructions per second).

4. Dhrystone

It is a **synthetic integer performance benchmark** that was developed in 1984. Benchmark contains less than 100 HLL statements, compiling to *1-1,5 kB of code*.

5. HINT

HINT (**Hierarchical INTegration**) is a computer benchmark that **ranks a computer system as a whole**.
Most benchmarks measure either the number of operations in a given time period, or the time required to perform a given fixed calculations. **HINT** does neither; rather it **performs a particular calculation with ever-increasing accuracy**.
The **accuracy** of the result at any given time is called the "**Quality**". **HINT measures** the **improvement of quality at any given time** as "**Quality Improvements Per Second**" or **QUIPS**.
HINT rating is a function of raw CPU processing power, L1 and L2" cache size and speed, and main-memory bandwidth. HINT allows comparisons over extreme variations in computer architecture, absolute performance, storage capacity and precision.

6. SPEC Benchmarks

A group called the **Systems Performance Evaluation Co-operative** (**SPEC**) or **Standard Performance Evaluation Corporation** formed to establish and maintain a standard **set of relevant benchmarks that can be applied to the newest generation of high performance computers**. **The first SPEC benchmark suite** was called **SPEC89**.
It consisted of the geometric mean of the runtimes for 10 programs in the suite.
The benchmark did not differentiate between integer and floating-point performance**.**

The **SPEC92** benchmark has 20 programs in the **CPU benchmark suite**.

SPEC92 is reported in two summary figures – floating-point (**SPECfp92**) and integer (**SPECint92**).
The **SPEC95** is an improvement of the SPEC92 benchmark.
In addition to the **SPECfp95** and **SPECint95** numbers, there are several related benchmark suits (**SPECint_base95**, **SPECint_rate95**).

SPEC CPU2000 benchmark suite

The SPEC CPU 2000 suite consists of integer (**SPECint2000**) and floating-point (**SPECfp2000**) benchmark suites. The first one consists of 12 tests in C and C++. The second one has 14 tests (Fortran-90, Fortran-77, and C). The CPU tests are designed to test three parameters: the CPU, the memory hierarchy and the compilers.
The performance is stated relative to a **reference machine**, a **300-MHz Sun Ultra5-10**, which gets a **score of 100**.

SPEC CPU 2006

SPEC CPU2006 includes two benchmark suites: **CINT2006** for measuring compute-intensive integer performance and **CFP2006** for compute-intensive floating point performance.
The CINT2006 suite includes 12 application-based benchmarks written in C and C++ languages. CFP2006 includes 17 CPU-intensive benchmarks written in C, C++, FORTRAN and a mixture of C and FORTRAN.
Performance metrics within **SPEC CPU2006** measure **system speed** and **throughput**. The **speed metrics**, **SPECint2006** and **SPECfp2006**, compare the ability of a computer to complete single tasks.
The **throughput metrics**, **SPECint_rate2006** and **SPECfp_rate2006**, measure the rate at which a computer can complete a number of tasks in a given amount of time.
The **reference machine** for SPEC CPU2006 benchmark is a ***Sun Ultra Enterprise 2*** workstation with a **296-MHz *UltraSPARC II* processor**.

7. Transaction Processing Benchmarks //*tehingtöötluse jõudlustestid*//

The **Transaction Processing Council** (**TPC**) is an industry-based organization representing computer systems and database vendors. The TPC benchmarks measure **transaction processing** (**TP**) and **database** (**DB**) **performance** in terms of **how many transactions** a given system and database can perform **per unit of time** (transactions per second (or minute)) within a **given response time limit**.
TPC benchmarks are classified into two categories – **Online Transaction Processing** (**OLTP**) //*sidustehingtöötlus*// and **Decision Support Systems** (**DSSs**) //*nõustussüsteem*//.
DSSs are used for business analysis purposes, to understand business trends.

TPC-App

TPC-App is a **synthetic benchmark** for measuring application **server performance** and web services performance. There are two performance metrics. The first is the Web Service Interactions //*teeninduslikku andmevahetust*// per second (**SIPS**) per Application Server SYSTEM. The second is the **Total SIPS**, which is the total number of SIPS for the entire tested configuration (SUT).

TPC-C

TPC-C is a benchmark for **OLTP.** It models simple order-entry applications. **TPC-C performance** is measured in **new-order transactions per minute (tpmC)**.

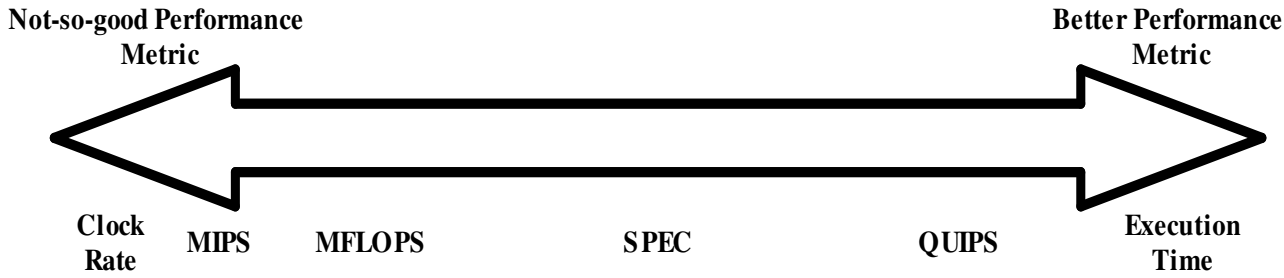
TPC-H

The **TPC-H** is a **DSSs benchmark**. It consists of a suite of business-oriented queries and concurrent data modifications. The **performance metric** is the **TPC-H** Composite Query-per-Hour Performance Metric **(QphH@Size),** which reflects multiple aspects of the capability of the system to process queries. The TPC-H Price/Performance metric is expressed as **$/QphH@Size.**

Benchmarks for Different Categories of Workloads

Workload Category	Benchmark Suite
CPU benchmarks	
Uniprocessor	SPEC CPU2000
	Java Grande Forum Benchmark
	SciMark
Parallel processor	SPLASH
	NASPAR
Multimedia	Media Bench
Embedded	EEMBC benchmarks
DSP	BDTI benchmarks
Transaction processing	
On-line transaction processing (**OLTP**)	TPC-C
Decision support systems (**DSS**)	TPC-H
Web server	SPEC web99
	TPC-W
	Volano Mark
Mail-server	SPECmail 2001
Network file system	SPEC SFS 2.0
Personal computer	SYSMARK
	Ziff Davis WinBench

Performance Metrics



Popular benchmarks typically reflect yesterday's programs, but microprocessor systems need to be designed for tomorrow's programs.

Example

We are going to run three benchmark programs **A**, **B**, **C** on brands **X**, **Y** and **Z**. The results are given in time units.

Benchmark Raw Time Results

Brand	Program / Contribution Rate		
	A 65%	**B 25%**	**C 10%**
X	302	**674**	**420**
Y	349	781	464
Z	**290**	694	421

Benchmark Normalized Time Results

Brand	Program / Contribution Rate		
	A 65%	**B 25%**	**C 10%**
X	1,04	**1,00**	**1.00**
Y	1,20	1,16	1,10
Z	**1,00**	1,03	**1,00**

Accumulated Benchmark Time

Brand X

1,04 x 0,65 + 1,00 x 0,25 + 1,00 x 0,1 = 1,03

Brand Y

1,20 x 0,65 + 1,16 x 0,25 + 1,10 x 0,1 = 1,18

Brand Z

1,00 x 0,65 + 1,03 x 0,25 + 1,00 x 0,1 = **1,01**

Beware pitfall!

An inviting method of presenting computer performance is to normalize execution times (t_e) to a reference computer, and then take the average of the normalized execution times.
If we average the normalized execution time values with arithmetic mean ($\mathbf{T_{am}}$), where

$$\mathbf{T_{am}} = \frac{1}{n}\sum_{i=1}^{n} tei \,.$$

The result will depend on the choice of the computer we use as the reference.

For example in the next case the program execution times are normalized to both A and B computers, and the arithmetic mean is computed.
When we normalize to A, the arithmetic mean indicates that A is faster than B by 5.05 times, which is the inverse ratio of the execution times.
When we normalize to B, we can conclude, that B is faster by exactly the same ratio.

Both results cannot be correct!
If we use the arithmetic mean, we can get confusing results.

	Time on A	**Time on B**	**Normalized to A**		**Normalized to B**	
			A	**B**	**A**	**B**
Program 1	1	10	1	10	0,1	1
Program 2	1000	100	1	0,1	10	1
Arithmetic mean	**500,5**	**55**	**1**	**5,05**	**5,05**	**1**
Geometric mean	***31,6***	***31,6***	***1***	***1***	***1***	***1***

Because we want to determine the computers relative performance, we should use an average that is based on multiplicative operations. The geometric mean is (T_{gm}) such an average:

$$\mathbf{T_{gm}} = \sqrt[n]{\prod_{i=1}^{n} tei} \;;$$

$$\mathbf{T_{am}} \geq \mathbf{T_{gm}}.$$

Conclusion: The two computers A and B are actually equal in performance.

Performance Measurement

Performance measurement can be done via the following means:

1. On-chip hardware monitoring
(*Pentium 3*, *Pentium 4*, *POWER 4*, *POWER 5*, *Athlon*, *Alpha*, *UltraSPARC*)
2. Off-chip hardware monitoring
    a. SpeedTracer (AMD)
    b. Logic analyzers
3. Software monitoring

Many systems are built with configurable features. Measurement on such processors can reveal critical information on effectiveness of microarchitectural structures, under real-world workloads.

Performance Estimation (Projection)

If the absolute drop-dead project performance goal is 1,3× some existing target, then the early product concept microarchitecture ought to be capable of some much higher number like 1,8×.

Dave Sager, one of the principal architects of the *Pentium 4*.

Some Benchmarking Mistakes

1. Only average behavior represented in test workload;
2. Caching effects ignored;
3. Buffer sizes not appropriate;
4. Ignoring monitoring overhead;
5. Not ensuring same initial conditions;
6. Not measuring transient (cold start) performance;
7. Using device utilizations for performance comparisons;
8. Collecting too much data but doing too little analysis, etc.

Summary

Performance Evaluation Techniques

Evaluation technique	Purpose of evaluation: Selection evaluation		Performance projection		Performance monitoring	
	New HW	New SW	Design new HW	Design new SW	Reconfigure HW	Change SW
Instruction mixes	**C**	**—**	**C**	**—**	**—**	**—**
Kernel programs	**B**	**C**	**B**	**C**	**—**	**C**
Analytic models	**B**	**C**	**B**	**C**	**B**	**—**
Benchmarks	**A**	**A**	**—**	**B**	**B**	**B**
Synthetic programs	**A**	**A**	**B**	**B**	**B**	**B**
Simulation	**A**	**A**	**A**	**A**	**A**	**A**
HW &SW monitoring	**B**	**B**	**B**	**B**	**A**	**A**

A – satisfactory;
B – provides some assistance but is insufficient. It should be used in conjunction with other techniques;
C – has been used but is inadequate;
"**—**" the technique is not applicable.

AMDAHL'S LAW

THE PERFORMANCE IMPROVEMENT TO BE GAINED FROM USING SOME FASTER MODE OF EXECUTION IS LIMITED BY THE FRACTION OF THE TIME THE FASTER MODE CAN BE USED.

Speedup (can be gained by using a particular feature) **– S**

S => (Performance for entire task using the enhancement) / (Performance for entire task without using the enhancement)

S => (Execution time for entire task without using the enhancement) / (Execution time for entire task using the enhancement)

Speedup Factors

1. Fraction enhanced – F_E or **f_x**

F_E (f_x) is the **fraction of the computation time** in the original system that can be converted to take advantage of the enhancement[1].

$$1 \geq f_x$$

2. Speedup enhanced – S_E or **S_x**

The **improvement** gained by the execution enhanced mode.

$$S_x > 1$$

Speedup $_{overall}$ – S_U

Execution Time

Execution time $_{old}$
Execution time $_{new}$

The new execution time is equal to the time spent using the unenhanced portion of task plus the time spent using the enhancement.

[1] **Enhancement** – a substantial increase in the capabilities of hardware or software.

$$\textbf{Execution time}_{\textbf{new}} = \text{Execution time}_{\textbf{old}} \times \{(1\text{-Fraction enhanced}) + \frac{Fraction - enhanced}{Speedup - enhanced}\} =$$

$$= (\text{Execution time}_{\textbf{old}} \times \text{Fraction}_{\textbf{old}}) + (\text{Execution time}_{\textbf{old}} \times \frac{Fraction - enhanced}{Speedup - enhanced})$$

$$\textbf{Execution time}_{\textbf{new}} = \textbf{Execution time}_{\textbf{old}} \times [(1-f_x)+(f_x/S_x)]$$
$$\textbf{Speedup}_{\textbf{overall}} = \textbf{Execution time}_{\textbf{old}} / \textbf{Execution time}_{\textbf{new}}$$

$$S_U = [(1-f_x)+f_x/S_x]^{-1}$$

$$S_x = (f_x \times S_U)/(S_U \times (f_x-1)+1)$$
$$f_x = (S_x \times (1-S_U))/(S_U \times (1-S_x))$$

An overall assumption of Amdahl`s Law is that the performance improvements are performed to be isolated from one another.

Conclusions

1. If the fraction of execution time reduced very small, then even a very large speedup for that piece will have very little impact on overall performance.

2. Resources are often better spent on small improvements for the most common operation instead of large speedups for rare events.

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve

***Cost / Performance* ratio**

To compare the performance of a new (modified) microprocessor **A** and old microprocessor **B** we can use the **speedup** (**speedup A**) as a function of the appropriate **CPI**s or **IPC**s:

$$\textbf{Speedup (A)} = \textbf{CPI}_B / \textbf{CPI}_A = \textbf{IPC}_A / \textbf{IPC}_B$$

The comparison is valid only if:

a. Both microprocessors (**A** and **B**) have the **same clock cycle time,** and
b. Both microprocessors run the **same number of instructions** (the same instructions in test program).

This assumption generally holds for the complete execution of single-threaded, uniprocessor, user-level programs. Multithreaded programs present a more complicated problem.

Amdahl's law applies only to improvements in IPC.

- More functional units allow higher instructions issue width.
- Better reordering algorithms reduce processor pipeline stalls.
- More specialized logic reduces the latency of computations.
- Large caches can reduce the average latency of memory accesses.

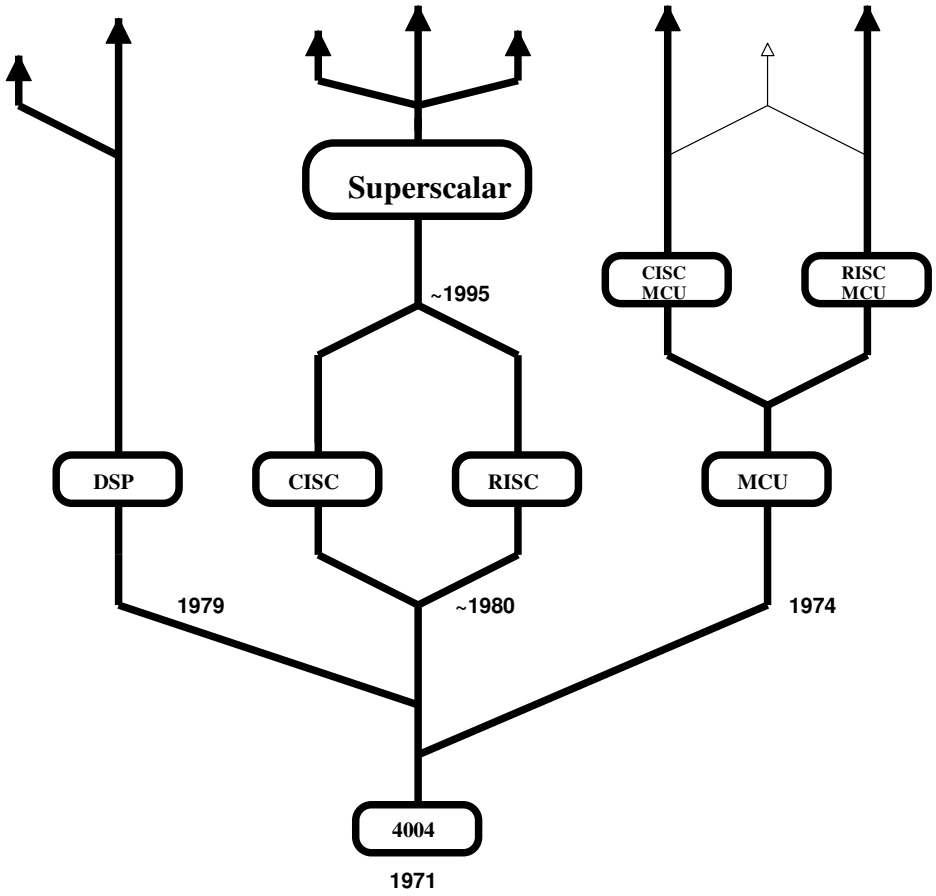
All these microarchitectural changes improve IPC but at the cost of design complexity and die area.

Extending Amdahl's Law to Multiple Enhancements

Suppose that **i enhancements** are used in the new design and **each enhancement affects a different portion of the system** (code, hardware). **Only one enhancement can be used at a time**, then the resulting **overall speedup is equal to (**where **all fractions f_i** refer to original execution time before enhancements were applied):

$$\textbf{Speedup} = \frac{1}{((1 - \sum_i f_i) + \sum_i f_i / S_i)}$$

MICROPROCESSORS TAXONOMY



Microprocessor Architecture Development Main Trends

The **first microprocessor**, the **Intel 4004**, was introduced in **1971**.

I During the first decade (1971-1980), the advent of the 4-bit microprocessors led to the introduction of the 8-bit microprocessors. The 8-bit microprocessors became the heart of the simple personal computers.

II The second decade (1981-1990) witnessed major advantages in the architecture and microarchitecture of 16-bit and 32-bit microprocessors. Instruction set design issues became the focus of researches. Instruction pipelining and cache memories became standard microarchitecture techniques.

III The third decade (1991-2000) was characterized by extremely aggressive microarchitecture techniques to achieve very high levels of performance. Deeply pipelined machines capable of achieving extremely high clock frequencies and sustaining multiple instructions executed per cycle became popular. Out-of-order execution of instructions and aggressive branch prediction techniques were introduced to avoid or reduce the number of pipeline stalls.

IV Now the fourth decade (2001-2010?) is focuses on instruction-level parallelism (ILP) will expand to include thread-level parallelism (TLP) as well as memory-level parallelism (MLP).

Architectural features that historically belong to large systems (multiprocessors and memory hierarchies), will be implemented on a single chip. Many traditional macroarchitecture issues will now become microarchitecture issues.
Power consumption will become a dominant performance impediment.

Evolution of Microprocessors
(by J. P. Shen and M. H. Lipasti)

	1970-1980	**1980-1990**	**1990-2000**	**2000-2010**
Transistor count	2K - 100K	100K - 1M	1M - 100M	100M - 2B
Clock frequency MHz	0,1 - 3	3 - 30	30 - 1000	1000 - 15000
Instructions per Cycle (IPC)	0,1	0,1 - 0,9	0,9 - 1,9	1,9 - 2,9

There are different ways how describe different microprocessor structures. Each classification holds an abundant variety of processor organization.
The simplest way to **classify microprocessor structures** may be next:

By **input signals**

1. Digital MP;
2. Analog MP:
   DSP;
   Media processor;

By **timing**

1. Synchronous MP;
2. Asynchronous MP (internally).

By **purpose**

1. Universal MP;
2. Special-purpose MP:
   [Microcontrollers].

By **resources**

1. One level;
2. Multilevel.

By **the number of tasks**

1. One task MP;
2. Multiple tasks MP.

By **chip implementation** (figure 1)

1. One chip MP;
2. Multiple chip MP;
3. Bit-slice MP;
4. Multi-chip module MP;
5. Multi-core MP.

By **internal bus organization** (figure 2)

1. One bus MP;
2. Multiple buses MP.

By **control unit organization**

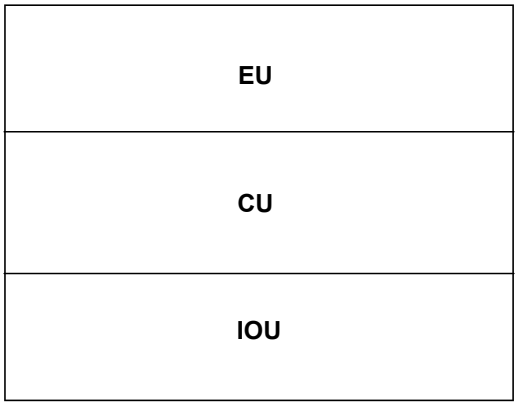
Microprocessor's **control unit** (**CU**) can be implemented in two schemes:

1. **Hardwired control unit** (**HCU**);
2. **Microprogrammed control unit (MCU).**

Comment

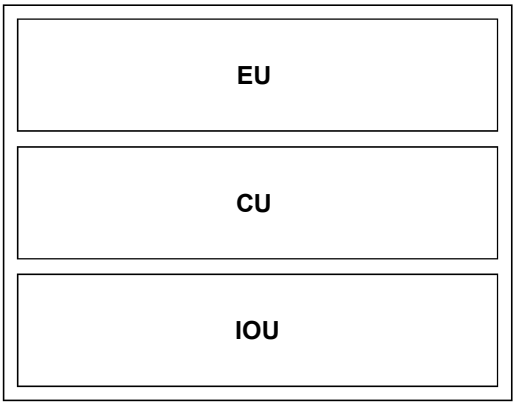
The **MCU** scheme is more flexible than the **HCU**. The meaning of an instruction can be hanged by changing the microinstruction sequence corresponding to the instruction. The instruction set can be extended simply by including a new ROM (a new ROM content) containing the corresponding microoperation sequences. Hardware changes to the CU are minimal.

In an HCU any change to the instruction set requires substantial changes to the hardwired logic but HCUs are generally faster than MCUs and used where the CU must be fast.



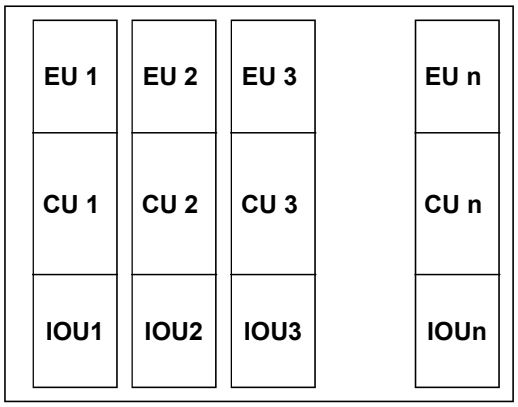
One-chip microprocessor

Intel P5



Multi-chip microprocessor

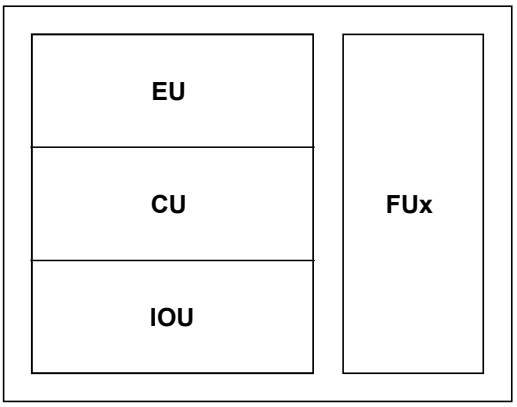
IMP 4 (National)
K 588 (SU)



Bit-slice microprocessor

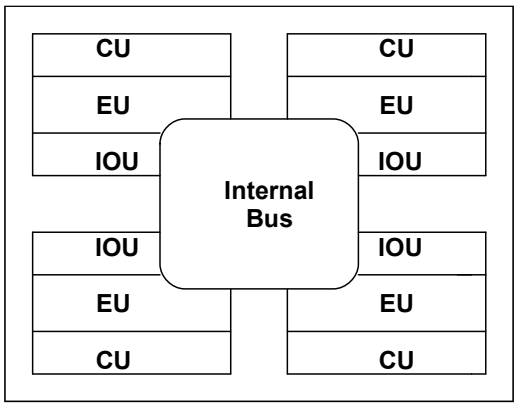
Intel 3002

The data word length in one slice is (2 to 4) or (8 to 16) bits



Multi-module microprocessor

Intel P6



Multi-core microprocessor

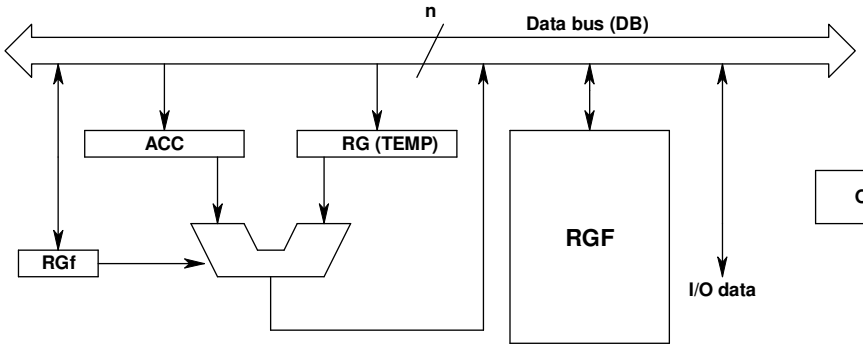
Quad-Core Intel Xeon
Eight-Core Sun Open SPARC T2

Microprocessor Implementation Models

CPU microarchitecture

Internal bus organization

A.

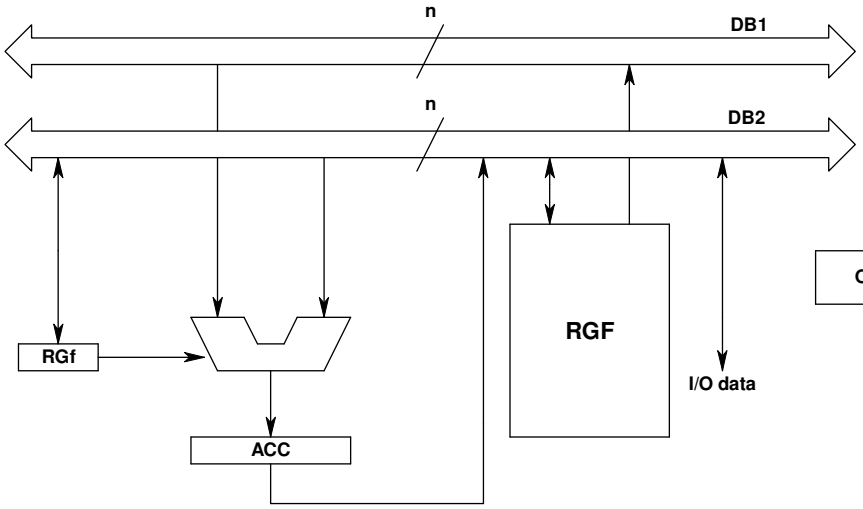


CISC microprocessors
(3 phases)

Intel 8080
Zilog Z80
Motorola M6800

OC	ADDR /mem/

B.

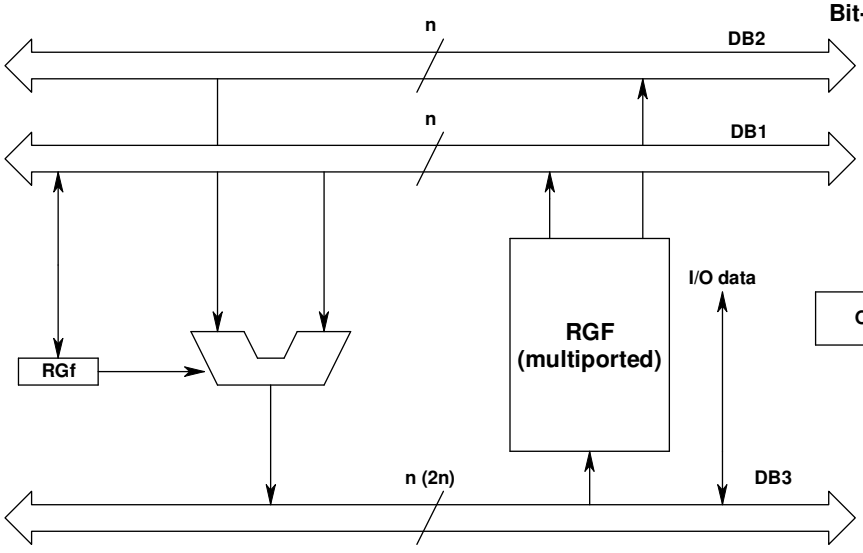


CISC microprocessors
(2 phases)

Intel 8086
Motorola 68000

OC	ADDR1 /RG/	ADDR2 /RG or mem/

C.

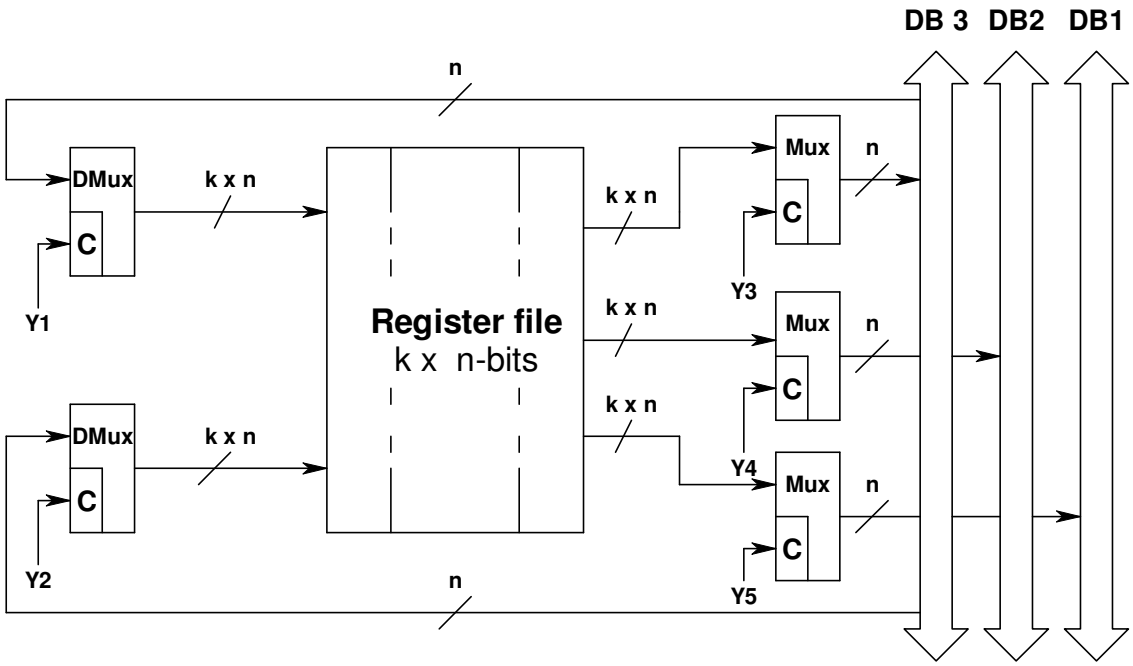


RISC and Post-RISC microprocessors
Bit-slice microprocessors
(1 phase)

MIPS R4000
Intel Pentium 4

OC	ADDR1 RD	ADDR2 SR1	ADDR3 SR2

Multiport register file organization



k - the number of registers
n - the register's length (in bits)
Yi - the control word i

Example

The **speed-up** (**$\%S_u$**) to be expected from the decrease in **CPI** period due to the different data path's bus-system organization is:

$$\%\mathbf{Su} = \frac{Tbi - Tb(i+1)}{Tb(i+1)} \times 100, \quad \text{where}$$

Tb – the operation execution time, using **i** buses ($\mathbf{i} \geq 1$) is:

$$\mathbf{Tb} = \mathbf{IC} \times \mathbf{CPI} \times \mathbf{t}, \text{ where}$$

IC – instruction count;
t – clock cycle time;

Assume, that IC will be not change (**IC=const.**) when going from the **i**–bus to the (**i+1**)-bus system in the microprocessor's microarchitecture.

CPI Count

3-phase structure	**2-phase structure**	**1-phase structure**
One bus	**Two buses**	**Three buses**
CPI = **3**	CPI = **2**	CPI = **1**

$$\%\mathbf{Su}_{\mathbf{i>(i+1)}} = = \frac{IC \times CPIi \times ti - IC \times CPI(i+1) \times ti}{IC \times CPI(i+1) \times ti} \times 100$$

$$\%\mathbf{Su}_{\mathbf{1>2}} = \frac{3 \times t1 - 2 \times t1}{2 \times t1} \times 100 = \frac{1}{2} \times 100 = \mathbf{50\%}$$

$$\%\mathbf{Su}_{\mathbf{2>3}} = \frac{2 \times t1 - 1 \times t1}{1 \times t1} \times 100 = 1 \times 100 = \mathbf{100\%}$$

$$\%\mathbf{Su}_{\mathbf{1>3}} = \frac{3 \times t1 - 1 \times t1}{1 \times t1} \times 100 = 2 \times 100 = \mathbf{200\%}$$

As the 2-bus or 3-bus structure increases the clock cycle (approximately by 10%) period, as a result of increased signal propagation time over the additional buses, then

$$\mathbf{t2 = 1{,}1 \times t1,}$$
$$\mathbf{t3 = 1{,}1 \times t2 = 1{,}21 \times t1.}$$

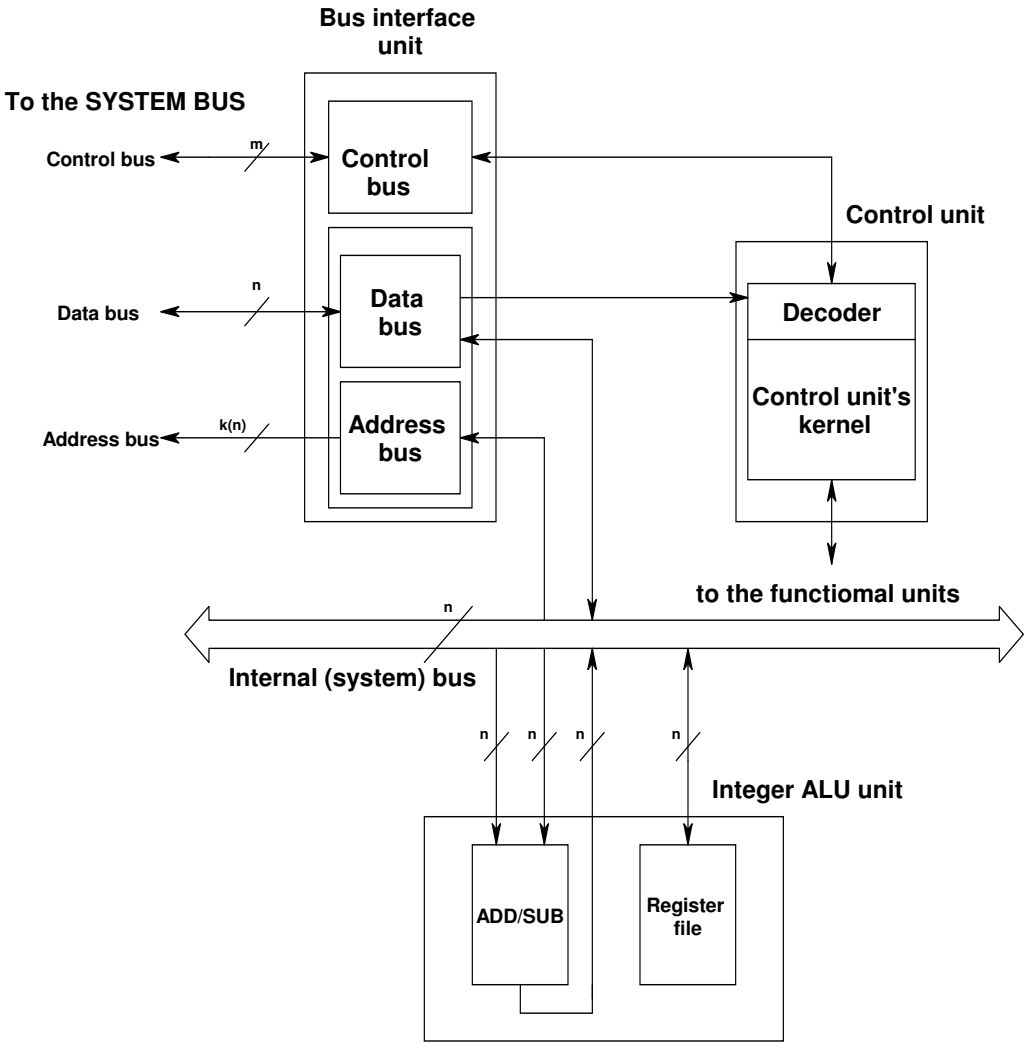
and

$$\%\mathbf{Su}_{\mathbf{1>2}} = \frac{3 \times t1 - 2 \times t2}{2 \times t2} \times 100 = \frac{3 \times t1 - 2 \times 1{,}1 \times t1}{2 \times 1{,}1 \times t1} \times 100 = 36{,}3\%$$

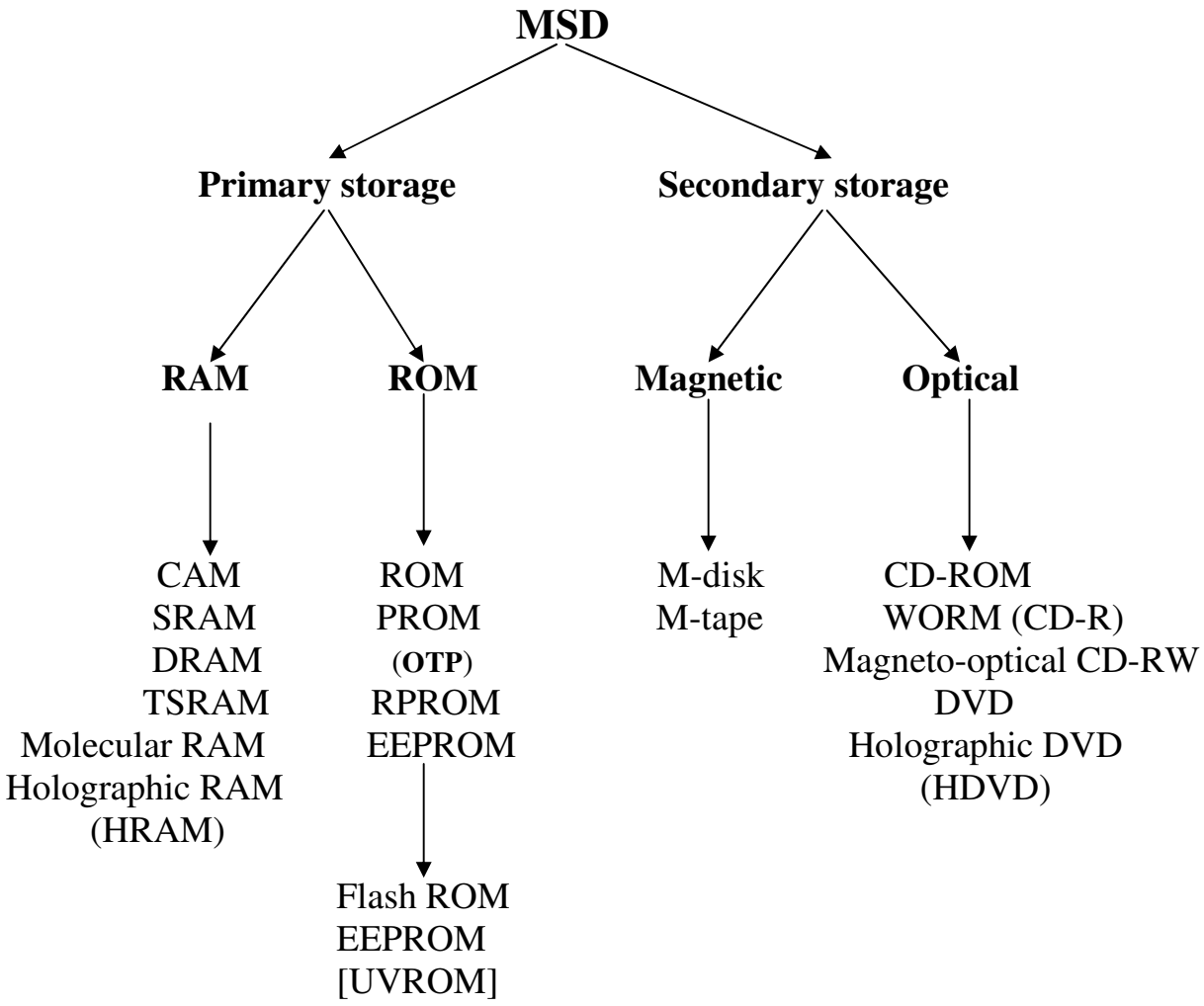
$$\%\mathbf{Su}_{\mathbf{2>3}} = \frac{2 \times t2 - 1 \times t3}{1 \times t3} \times 100 = \frac{2 \times t2 - 1 \times 1{,}1 \times t2}{1 \times 1{,}1 \times t2} \times 100 = 81{,}8\%$$

$$\mathbf{Su}_{\mathbf{1>3}} = \frac{3 \times t1 - 1 \times t3}{1 \times t3} \times 100 = \frac{3 \times t1 - 1.21 \times t1}{1{,}21 \times t1} \times 100 = 147{,}9\%$$

Earliest microprocessor's architectural model



MEMORY-STORAGE DEVICES



Word is a natural unit of organization of memory.

Unit of transfer for main memory is generally is word, but not always the unit of transfer is not equal to a word or to an addressable unit.

As for external memory, data are often transferred is much larger units than a word, and these are referred to as blocks.

Latency of the memory (L) – the time delay from when the processor first requests data from memory until the processor receives the data.

Memory bandwidth (BW) – the rate in which information can be transferred from the memory system.

An Ideal Memory

1. ***Infinite capacity***;
2. ***Infinite bandwidth*** (for rapidly streaming large data sets and programs);
3. ***Zero latency*** (to prevent the processor from stalling while waiting for data or program code);
4. ***Nonvolatility*** (to allow data and programs to survive even when the power supply is cut off);
5. ***Zero*** **or very low** ***implementation cost***.

Examples of Memories Internal Organization

SDRAM - synchronous dynamic RAM

SDRAM presents an architectural advance ⇒ multiple memory banks.
Each memory bank has its own independent page buffer, so that two separate memory pages can be simultaneously active. A DRAM memory address is internally split into a row address and a column address.
The row address selects a page from the storage and the column address selects an offset within the selected page. Synchronous DRAMs access modes include:

1. Burst read/write mode (fast successive accesses to data in the same page);
2. Interleaved row read/write mode (alternating burst accesses between banks);
3. Interleaved column access mode (alternating burst accesses between two chosen rows in different banks).

SDRAM incorporates features that allow it to keep pace with system bus speeds. SDRAM uses a clock input for synchronization whereas the DRAM is an asynchronous memory. DRAM uses two clocks (RAS and CAS). Each operation of DRAM is determined by the timing phase differences between these two clocks.

Z–RAM - zero capacitive RAM

Z-RAM is a capacitor-less, single-transistor DRAM technology that exploits the intrinsic floating-body effect of silicon-on-insulator devices. Z-RAM`s cell size can be half the size of an embedded DRAM transistor plus capacitor cell. It is less than a fifth the size of a six-transistor SRAM equivalent.
Z-RAM s small cell size results in higher memory density, low cost and the small cell size reduces the probability of memory cell alpha particle hits, which improves soft error rate performance up to 10 times over SRAM.

DDR-SDRAM - double-data-rate SDRAM

Internally DDR memory is similar to standard SDRAM, but employs four internal memory banks, which feed into an output data buffer.

SLDRAM - synchronous link DRAM

Similar to SDRAM, but it packs eight internal memory banks. Employs a programmable data burst transfer protocol. Like DDR-SDRAM, SL-DRAM uses both rising and falling clock edges to transfer data. Its bus interface is designed to run at clock speeds of 200-600 MHz and has a two-byte-wide datapath.

MDRAM – multi-bank DRAM

Internally MDRAM uses 32 banks per megabyte, where each bank has its own I/O port that feed into a common internal bus. Data can be read or written to multiple banks simultaneously. This permits to handle many overlapping data transactions.

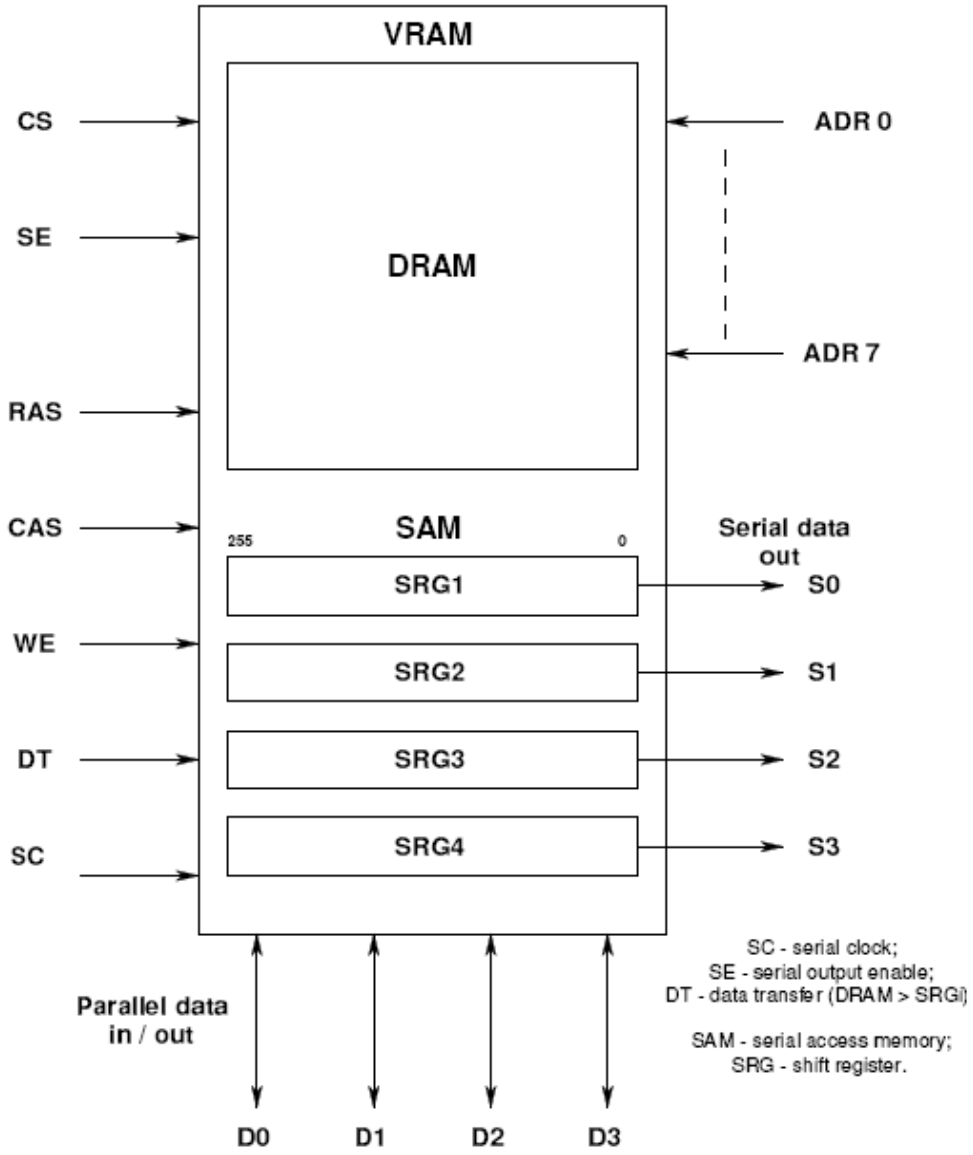
ESDRAM - cache-enhanced DRAM

ESDRAM combines a DRAM and 4-kbit cache on the same chip. DRAM is a synchronous DRAM, which has four internal databanks.

VRAM - video RAM

Video RAM bases on DRAM architecture. VRAMs have one or two special serial ports. VRAM is frequently referred to as dual-port or triple-port memory. The serial ports contain registers (SAM)

which may hold the contents of whole row. SAM cells are used as a shift register and are usually made with either standard SRAM cells or multiple transistor DRAM cells.
It is possible to transfer data from the whole row memory array to the register in single access cycle. Because the register is based on fast, static cells, the access to it is very fast, usually several times faster than to the memory array. In most typical applications VRAM is used as screen buffer memory.
The parallel port is used by host processor, and the serial port is used for sending pixel data to the display. The DRAM and SAM ports can be independently accessed at any time except during on internal transfers between the two memories.



Data transfer mode in the VRAM's DRAM:
READ => MODIFY => WRITE BACK \block transfers\

WRAM - window RAM

WRAM is a variation on dual-ported memory. WRAM is optimized for acceleration and can transfer blocks and supports text and pattern fills. Its two ports allow input of graphic drawing and output of screen refresh data to be processed simultaneously, resulting in much higher bandwidth than conventional single-ported memory types.

Fast block copies and fills are called window operations. WRAM is accessed in blocks or windows, which makes it slightly faster than VRAM. Now WRAMs are replaced by SGRAMs.

SGRAM - synchronous graphic RAM

SGRAM contains the speed-enhancing features of SDRAM. Like SDRAM, SGRAM can work in synchronously with system bus speed. It has the graphic capabilities that enhance 3D graphics performance. SGRAM is single-ported, but it can open two memory pages at once, simulates the dual-port nature of other video-RAM technologies.

3D-RAM

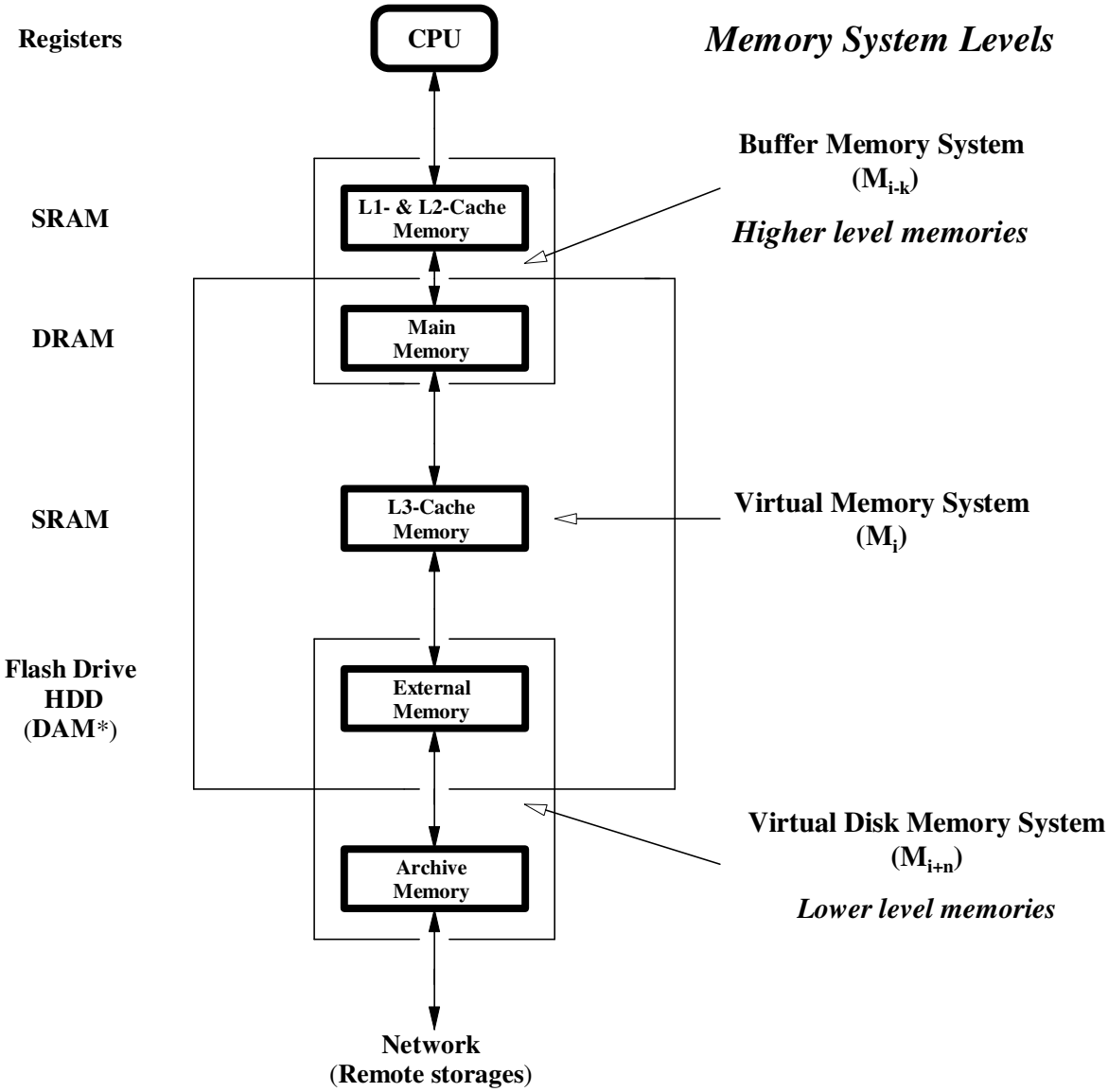
3D-RAMs are used in video cards in tandem with a 3D graphics accelerator. The memory contains a **CDRAM** (cached DRAM) array and an ALU block, which allows some image manipulation operations to be performed.

Literature

1. Bruce Jacob, Spencer W. Ng, David T. Wang. Memory Systems: Cache, DRAM, Disk. Elsevier, 2008.

2. Betty Prince. Emerging Memories. Technologies and Trends. Kluwer Academic Publishers, 2002.

MEMORY SYSTEM HIERACHY



Each memory level of the hierarchy is usually a subset of the level below (**lower level**) - data found in a level is also found in the level below. The storage devices get slower, large and cheaper as we move from higher to lower levels. At any given time, data is copied between only two adjacent memory levels.

Access time – time to get the first word. Access time is composed of **address setup time** and **latching time** (the time it takes to initiate a request for data and prepare access).

An average (effective) access time (t_{avg}) or AMAT (average memory access time) is equal to:

$$\mathbf{AMAT = t_{avg} = t_{hit} + m_R \times t_{miss}}$$

$\mathbf{t_{miss} = t_{penalty}}$; $\mathbf{m_R}$ – miss rate

Transfer time – time for transferring remaining words.

Memory System Management Goals

1. **Protection** (protect each process from each other and protect of the OS and of the users);
2. **Utilization** (ensuring full use of memory)**;**
3. **Allocation** (each process should get enough memory space);
4. **Address mapping**.

Terminology

1. **Block** (**page**) - maximum unit that may be present (usually has fixed length).
2. **Hit** – block is found in upper level.
3. **Hit rate** ($\mathbf{p_h}$) - number of hits / total number of memory accesses.
4. **Miss** – block not found in upper level.
5. **Miss rate** ($\mathbf{m_R}$) – fraction of references that miss.
6. **Hit time** ($\mathbf{t_{hit}}$) – time to access the upper level.
7. **Miss penalty** ($\mathbf{t_{miss}}$, $\mathbf{t_{penalty}}$) - time to replace block in upper level, (plus the time to deliver the block to the CPU).

Information in Memory System

Information that stored **in memory system hierarchy levels**

$$\mathbf{M_1,\ M_2,\ldots,\ M_{i-1},\ M_i,\ M_{i+1},\ldots,\ M_{n-1},\ M_n}$$

where $\mathbf{M_1}$ is the **highest level** and $\mathbf{M_n}$ is the **lowest level**,

must satisfy three main properties:

1. **Inclusion** //*sisalduvus*//;
2. **Coherence** //*koherentsus*//,
3. **Locality** //*lokaalsus*//.

Inclusion

The **inclusion** property implies that all information items are originally stored in the outermost (in the lowest) memory level.

$$\mathbf{M_{i-1} \subset M_i \subset M_{i+1}}$$

Coherence

A memory is coherent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same memory address.
The coherence property requires that all copies of the same information item at successive memory levels be consistent. The concept is to balance system to adjusting sizes of memory hierarchy components.

Locality

Memory hierarchy exploits locality to create illusion of large and fast memory.
The memory hierarchy was developed based on a program behavior known as locality references.

Dimensions of locality property:

1. **Temporal locality** – recently referred items (instructions or data) are likely to be referred again in the near future.
2. **Spatial locality** – the tendency for a process to across items whose addresses are near one another. Spatial locality implies temporal locality, but not vice versa.
3. **Sequential locality** – in typical programs the execution of instructions follows in a sequential order (unless branch instructions).
   Sequential locality is a subset of spatial locality.

Buffers and Caches

A. Buffer

A **buffer** is a small storage area (usually DRAM or SRAM) used to hold data in transit from one device to another. Buffers resolve differences in data transfer rate or data transfer unit size.

B. Cache

A **cache** is a storage area (usually RAM), which differs from a buffer in several ways:

1. Data content is not automatically removed as it is used,
2. Cache is used for bidirectional data transfer (*versus* input or output buffer),
3. Cache is used only for storage devices access (not for I/O devices),
4. Caches are usually much large than buffers,
5. Cache content must be managed intelligently.

Cache Controller

A **cache controller** is a **processor that manages cache content**. A cache controller can be implemented in:

a. a **storage device controller** or **communication channel**, as a special–purpose processor controlling RAM.
b. the **operating system**, as a program that uses part of primary storage to implement the cache.

Primary Storage Cache

One way to limit wait states is to use an SRAM **cache between the CPU and primary storage** (main memory). Usually is used **multilevel primary storage cache system** (**L1-, L2-,** and **L3-level caches**).

Secondary Storage Cache

Disk caching is common in modern systems, particularly in file and database servers.
The OS is the best source of file access information because it updates the information dynamically as it services file access requests. Because the OS executes on the CPU, it is difficult to implement access-based control if the cache controller is a special–purpose processor within a disk controller.

Cache Pipelining

Cache pipelining is a technique in which memory loads the requested memory contents into a small cache composed of SRAM, then immediately begins fetching the next memory contents.
This creates a two-stage pipeline, where data is read from or written to cache in one stage, and data is read or written to memory in the other stage.

Memory System Hierarchical Levels

Level	**1** (highest level)	**2**	**3**	**4** (lowest level)
Name	**Registers**	**Cache**	**Main Memory**	**Disk Storage**
Typical size	< 1 KB	<16 MB	<16 GB	>100 GB
Technology	Memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic Disk
Access time (ns)	0,25-0,5	0,5-25	80-250	5 000 000
Bandwidth (MB/s)	$2*10^4$ - 10^5	$5*10^3$ - 10^4	10^3 – $5*10^3$	20 – 150
Managed by	Compiler	Hardware	Operating system	Operating system / operator
Backed by	Cache	Main Memory	Disk	CD or M-tape

The efficiency with which memory space is being used at any time is defined as the ratio of the memory space occupied by active user programs and data to the total amount of memory space available.
The "wasted" memory space can be attributed to several sources:

1. **Empty regions** – the blocks of instructions and data occupying memory at any time are generally of different length (**memory fragmentation**);
2. **Inactive regions** – data may be transferred to main memory and may subsequently transferred back to external memory, without ever being referenced by a processor;
3. **System regions** – these regions are occupied by the memory-management software.

Main Memory System Organization

The memory consists of multiple **memory modules**. The **memory module** is the main **building block** of the main memory system.

Each **memory module** is capable of performing **one memory access at a time**.
Each **memory module** contains a **subset of the total physical address space**.
Each **memory module** has two important **parameters**:

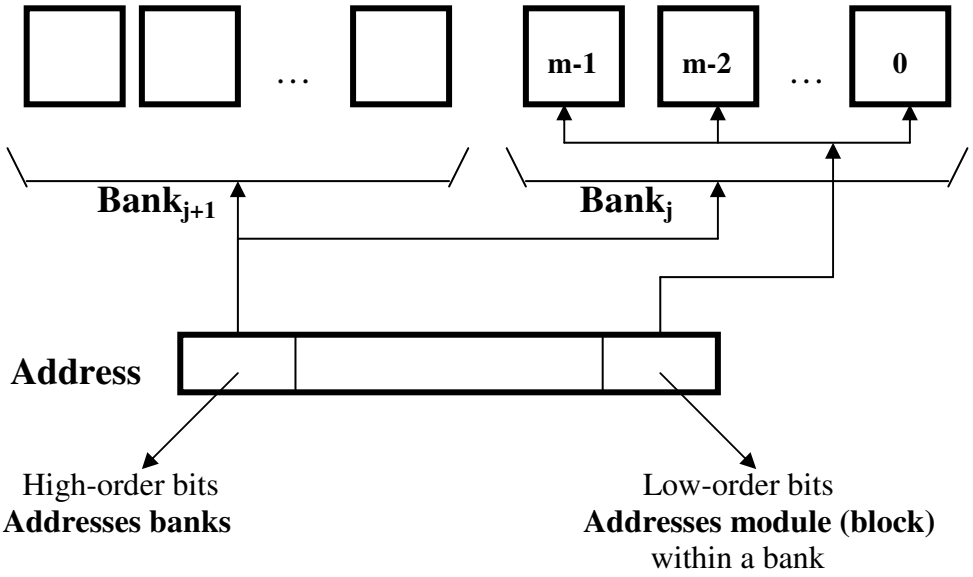
a. **Module access time** t_{accm} - the time required to retrieve a word into the memory module output buffer register.
b. **Module cycle time** t_{cycm} - the minimum time between requests directed at the same module.

The memory modules (blocks) are organized into memory banks.
Each memory bank consists of multiple modules that share the same input and output buses. Each memory bank consists of memory address register and a memory data register.
Memory interleaving is a speed enhancement technique which distributes memory addresses such that concurrent accesses to memory modules are possible.

A. In **low-order interleaving** the modules is organized such that the consecutive memory addresses lie in consecutive physical modules.

B. In **high-order interleaving** (**banking**) the consecutive addresses lie in the same physical module. The high-order address bits select a physical block and the remaining bits select a word within that block.

Memory Modules



Within a memory bank only one memory module is able to begin or complete a memory operation during any given bus cycle. The memory bank organization is meaningful only **if the memory cycle time is greater than the bus cycle time.**

Address Space

Address space – the set of identifiers, that may be used by a program to reference information.

$$q_{adr}$$

Memory address space – the set of physical main memory locations in which information items may be stored.

$$q_{madr}$$

System address space – The address space that contains memory (q_{madr}) and I/O-spaces (q_{IOadr}).

$$q_{sadr}$$

Physical (real or **direct) addressing space**

$$q_{padr}$$

$$q_{sadr} > q_{padr}$$

Virtual address is an address that corresponds to a location in virtual space and translated by address mapping to a physical address. The virtual address space is divided into pages.
The maximum physical address size can be larger or smaller than the virtual address size.

If a physical address is large than the virtual address, then it means that a microprocessor system could have more physical memory than any one program could access. This could be useful for running multiple programs simultaneously.
If a physical address is smaller than the virtual address, then a program cannot have all of its virtual pages in main memory at the same time.

Basic Addressing Modes

Addressing modes have the ability to significantly reduce the instruction count (t_3 = var).

Mode	Algorithm	Principal advantage	Principal disadvantage
Immediate	Operand	No memory reference	Limited operand's value
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = (R)	No memory reference	Limited address space
Register Indirect	EA = ((R))	Large address space	Extra memory reference
Displacement	EA = A+(R)	Flexibility	Complexity
Stack	EA = [top of stack]	No memory references	Limited applicability

In **load-store architecture** is dominated **immediate** and **displacement** addressing modes.

Memory Wall

Memory Bus Speed *versus* RAM Speed

The memory speed is a function of a memory bus (sometimes of a system bus speed) speed.
The increase in microprocessor performance places significant demands on the memory system.
Typical on-chip L2 cache performance degradation is now more than 2× the ideal cache.
For multiprocessors, inter-cache latencies increase this degradation to 3× or more for 4 processors and up. A perfect memory system is one that can supply immediately any datum that the CPU request.
The ideal memory is not practically implementable as the three general factors of memory

capacity,
speed, and
cost

are in direct opposition.

Example

Intel's Microprocessors Memory-bus Speed *versus* Internal Clock Speed

Type / Generation (Gi)	Year	Data / address bus width	Memory bus speed (MHz)	Internal clock speed (MHz)	Comments
8088 / G1	1979	8/20	4,77-8	4,77-8	
8086 / G1	1978	16/20	4,77-8	4,77-8	
80286 / G2	1982	16/24	6-20	6-20	
80386DX / G3	1985	32/32	16-33	16-33	
80386SX / G3	1988	16/32	16-33	16-33	L1 cache
80486DX / G4	1989	32/32	25-50	25-50	
80486SX / G4	1989	32/32	25-50	25-50	
80486DX2 / G4	1992	32/32	**25-40**	**50-80**	Clock doubling
80486DX4 / G4	1994	32/32	**25-40**	**75-120**	Cock tripling
Pentium / G5	1993	64/32	**60-66**	**60-200**	
Pentium MMX/G5	1997	64/32	**66**	**166-233**	Multimedia functions
Pentium Pro / G6	1995	64/36	**66**	**150-200**	RISC-like instructions L2 cache, 10-stage pipeline
Pentium II / G6	1997	64/36	**66**	**233-300**	
Pentium II / G6	1998	64/36	**66/100**	**300-450**	
Pentium III / G6	1999	64/36	**100**	**450-1200**	SMD extension
AMD Athlon / G7	1999	64/36	**266**	**500-1670**	
Pentium 4 / G7	2000	64/36	**400**	**1400-**	20-stage hyperpipeline

The memory wall problem arises because the difference between two competing trends, each of which is an exponential function but with a different base grows at an exponential rate.

Resulting in the gap doubling every 2,1 years.

Processor–Memory Performance Gap *is the primary obstacle to improved computer system performance.*

Overcoming Memory Wall

An economical solution is a memory hierarchy organized into several levels, each smaller, faster and more expansive per byte than the next.
The concept of memory hierarchy takes advantage of the principle of locality.

Traditional techniques

- Larger caches, deeper cache structures. ***We have to improve*:**

    1. **Hit time,**
    2. **Miss penalty,**
    3. **Miss rate (to increase the memory bandwidth).**

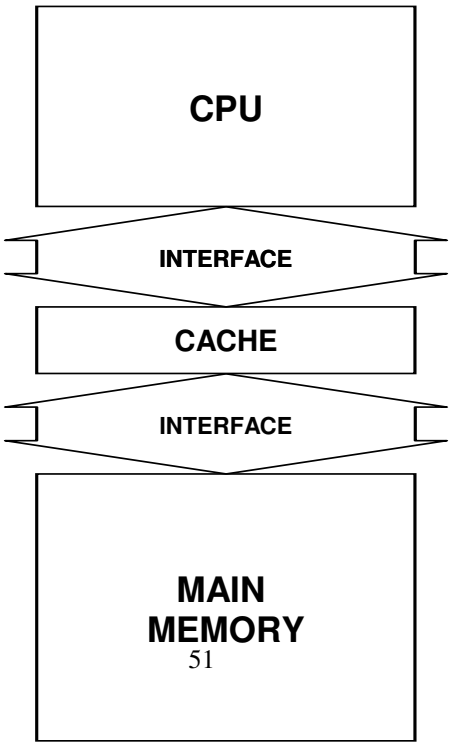
- Latency hiding via prefetching;
- Hardware multithreading.

Future research opportunities

- Reduced intercache scaling effects via affinity scheduling of tasks;
- Machine learning applied to code prefetching and code pre-positioning;
- Self-optimizing cooperation between the hardware and software directives.

Memory-level parallelism (**MLP**) is the ability to perform multiple memory transactions at once. As for read and write operations at once, or multiple read operations. Risky is to perform multiple write operations at once (address conflicts).

Classical (one-word wide) Main Memory Organization



Suppose that 1 clock cycle takes to transfer address or data in memory system. Each access to main memory takes 20 clock cycles. One cache memory line contains four main memory words. The total transfer time (T_{tot}) of one cache line content is equal to:

$$\mathbf{T_{tot} = T_{address\ transfer} + T_{memory\ access} + T_{data\ transfer}}$$

$$\mathbf{T_{tot}} = 1 + 4\times20 + 4\times1 = \mathbf{85\ clock\ cycles}$$

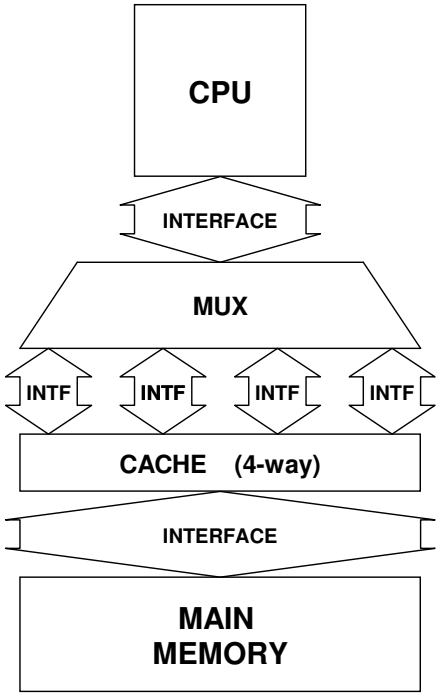
The Techniques to Improve Effective Memory Bandwidth

a. **Wider and faster connection to memory**;
b. **Large on-chip caches**;
c. **More efficient on-chip caches**;
d. **Dynamic access ordering**;
e. **Logic and DRAM integration** (**IRAM**).

Though many techniques have been suggested to circumvent the Memory Wall problem, most of them provide one time boosts of either bandwidth or latency.

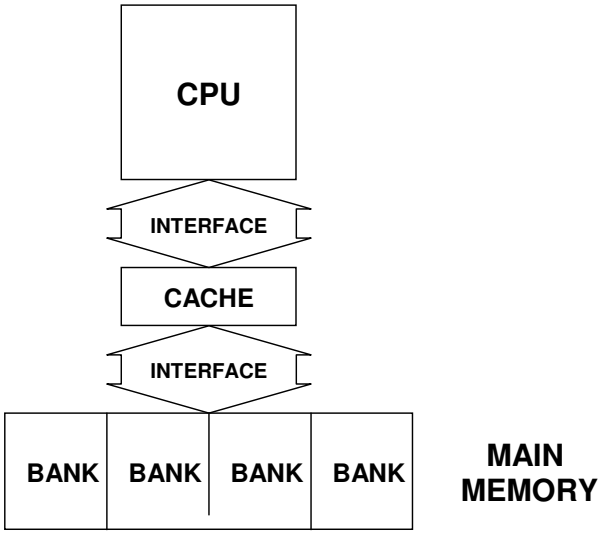
Examples

Wider Memory Organization (Alpha 21064)



$T_{tot} = 1 + 2\times20 + 2\times1 =$ **43 clock cycles**

Interleaved Memory Organization (DEC300 model 800)



$T_{tot} = 1 + 1\times20 + 4\times1 =$ **25 clock cycles**

Memory Capacity Planning

The optimization problem for a storage hierarchy concerns costs, and a selection among design trade-offs. It is important to achieve a high a hit ratio as possible at memory level M1.

In optimizing the cost-effectiveness of a storage hierarchy design, two approaches have been used:

1. In the simpler approach, only the storage hierarchy itself is optimized and the CPU is ignored.
   The *figure-of-merit //kvaliteedinäitaja//* employed is the product of the mean access time and the total cost of the storage system.
2. In the more complex approach, the storage hierarchy is treated as a component of computer system, and the cost and performance of the CPU are included in the study.

1. Access frequency (f_i) to memory system level Mi is:

$$\mathbf{f_i = (1-h_1) \times (1-h_2) \times \ldots \times (1-h_{i-1}) \times h_i}$$

$$\sum_{1}^{n} fi = 1$$

$$\mathbf{f_1 >> f_2 >> f_3 >> \ldots >> f_n,} \quad \text{where}$$

$\mathbf{h_i}$ – **hit rate** on memory system level $\mathbf{M_i}$ ($\mathbf{0 < h_i < 1}$);
$\mathbf{1-h_i}$ – **miss rate.**

2. Effective access time (T_{eff}) in memory system:

$$\mathbf{T_{eff} = h_1 \times t_1 + (1-h_1) \times h_2 \times t_2 + (1-h_1) \times (1-h_2) \times h_3 \times t_3 + \ldots +}$$
$$\mathbf{+ (1-h_1) \times (1-h_2) \times \ldots \times (1-h_{i-1}) \times h_i \times t_i,} \quad \text{where}$$

$\mathbf{t_i}$ – access time in memory system level **i.**

3. The total cost of a memory hierarchy (C_{tot}):

$$\mathbf{C_{tot} = \sum c_i \times s_i,} \quad \text{where}$$

$\mathbf{c_i}$ **-** i-level (relative) **cost**,
$\mathbf{s_i}$ – i-level **size** (capacity).

Generally:

$$\mathbf{s_1 < s_2 < s_3 < \ldots < s_n,}$$
$$\mathbf{c_1 > c_2 > c_3 > \ldots > c_n.}$$

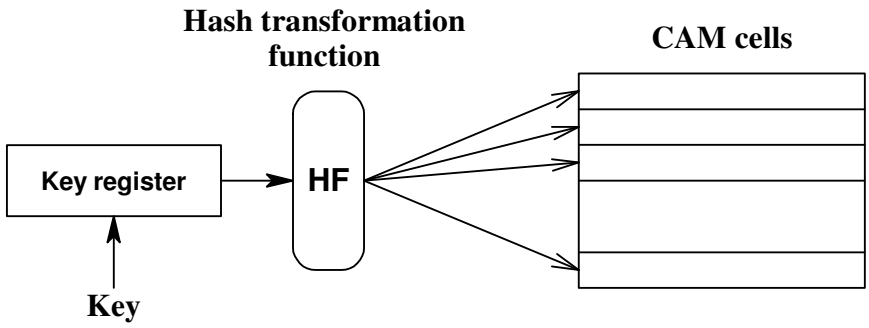
ASSOCIATIVE MEMORY
Content Addressable Memory

Associative memories are referred also as:

1. **Catalog memory;**
2. **Parallel search memory;**
3. **Data-addressed memory;**
4. **Content-addressable memory or CAM**.

- Traditional memories have been organized conventionally as sets of words, each word comprising a fixed number of binary digits.
- Associated with each word is an address, which corresponds in some manner with the physical location in memory at which the word value is stored. Traditional memories store data spatially.
- Access to a word in memory requires designation of an address in the form of binary number.
- In the associative memory (AM) a data word is not obtained by supplying an address that specifies the location of that word in memory.
- An identifying descriptor (A) or key is provided to memory.
- In the memory is then searched until an exact match is found between the submitted descriptor (A_S) and search key and a descriptor associated (A_A) or key with a data word (D_A).
- The A_A descriptor may be part of each data word or the descriptor may be stored separately (sometimes $A_A = D_A$).
- A true associative memory, in the sense of completely simultaneous interrogation of all bits in all words of a block of memory, requires that each elementary memory cell be furnished with a logic operation capability.
- Many implementations of content addressability can be viewed as hash addressing.
- Another approach is TRIE memory, which is a kind of hash coding but does not suffer from collisions.

Hash coding (key transformation) is a process in which a search key is transformed, through the use of a hash function, to an actual address for the associated data.
A hash function is the mathematical function for turning some kind of data into a small integer, that may serve as an index into an array. The values returned by a hash function are called hash values. The data is usually converted into the index by taking a modulo.

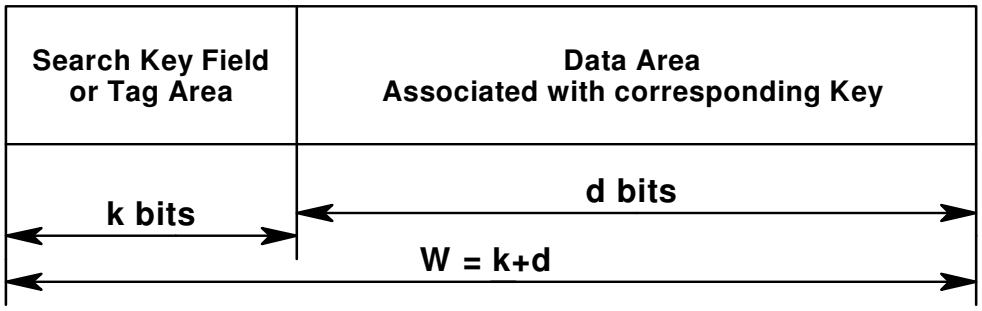


Age-addressable memory is a special-purpose associative memory which uses the date and time when the data was written as a retrieval key.

CAM Operation

The entire word line in the CAM is w elements wide where the first k bits are used for the tag and d bits used for the data.
The entire row or word can be viewed as a single entity, the key is just compared the first k bits within a word, essentially masking off the d data bits.

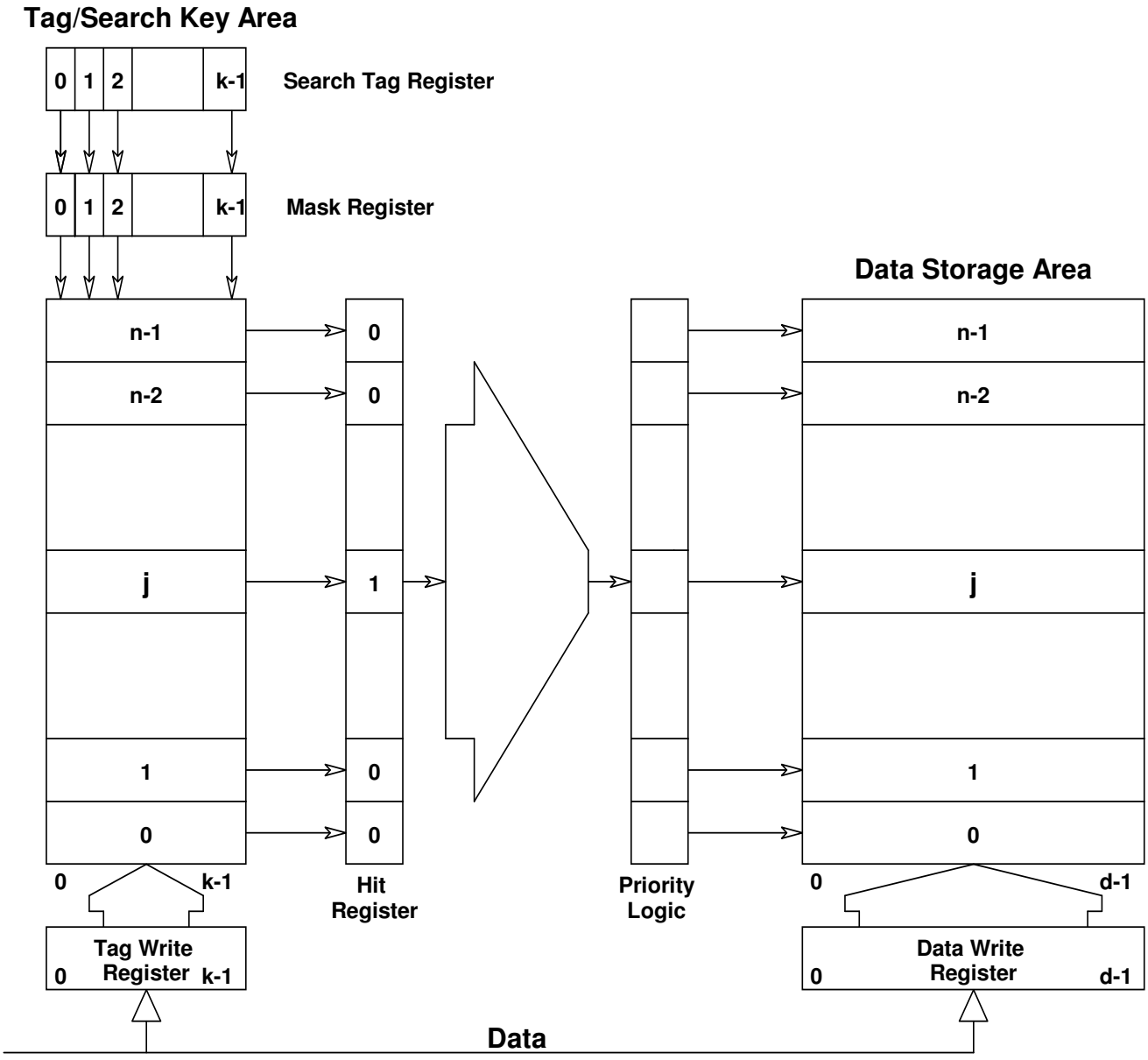
Layout of single cell in associative memory



The operation of a CAM is like that of the tag portion of a fully associative cache.
But unlike CAMs, caches do not use priority encoders since only a single match occurs.
CAM is not a RAM where a particular address is permanently tied to a specific physical location.

CAM Architecture Model

[Data storage capacity is $n \times d$ bits]



There are **four basic operations** which CAM performs:

1. Reads,
2. Writes,
3. Selection of a line,
4. Multiple response resolution.

Reads

During a read operation a key is first load into the search tag register and then, if needed, the mask register is loaded with whatever is desired. Typically mask register allows all bits of the search key to be used to scan the directory, but can be set to only compare certain bit position in the key.

Data searching methods

Associative search can determine not only an exact match, i.e., all bits are identical in the compared words, but also the degree of similarity in a partial match. For similarity measure CAMs use mostly the Hamming distance.

1. **Classical** or **trivial searching**
   $A_S = A_A$
2. **Complicated searching**
   $A_S > A_A$,
   $A_S < A_A$,
   $A_{Smin} < A_A < A_{Smax}$,
   $A_{Smin} = A_A$,
   $A_{Smax} = A_A$, etc.

Selection

The key is compared to all the tags of the words in the tag area, and when a match occurs, it sets a bit in the hit register. This indicates the unmasked bits in the key were the same as their corresponding bits for particular tag.
Once all the comparing is completed and the hit register is stable, the cell in the hit register containing a "**1**" indicates, which line in the data storage area is to be output (line **j**).

Multiple responses

This possibility exists, the hit register may designate to this. Since several hits may occur, but only one line of data can be output at a time, so some method of resolving this conflict must be incorporated into the system.
This is the job of the priority logic (response processor), to choose a single line of data to output based on the state of the hit register.

Writes

Writing to a CAM is not simply the converse of reading as it in a traditional RAM.
Once the decision has been make to write to the CAM, both the tag write register is loaded with the intended tag and the data write register is loaded with the data to be linked to the tag.
There is a question, how to choose the line in the CAM to replace with a new values for tag and data, by purging the old values from the CAM. This is known as a replacement strategy and includes the policies such as FIFO, LIFO and others.

Data writing methods:

1. **By address**,
2. **By attribute**,
3. **By step**,
4. **By sorting**.

CAMs Taxonomy

CAMs are divided into classes, which define how the search is done within the CAM. Since the tags are words of a certain bit length, the classifications are:

1. **Bit-serial-word-parallel,**
2. **Bit-parallel-word-serial,**
3. **Fully parallel (all-parallel).**

Bit-serial-word parallel

A search is accomplished by comparing the MSB of each $\mathbf{A_S}$ **tag** (**key**) to the MSB of the $\mathbf{A_A}$ **tag**, in all words simultaneously. Each bit of every tag is compared sequentially to the key.

Bit-parallel-word-serial

It is just opposite of the previous method. The entire key is compared to each tag in succession until a match occurs. This is the easiest to envision since it is just a linear search.

Fully parallel

In this case, every tag of every word is compared to the entire key simultaneously. This is the hardest to implement and offers the highest performance, but is the most costly method.

CAM Drawbacks

1. Functional and design complexity of the associative subsystems;
2. High cost for reasonable storage capacity
3. Relatively poor storage density compared to conventional memory;
4. Slow access time;
5. A lack of software to properly use the associative power of the memory system.

Using CAMs

1. In networking (routers, search engines);
2. For pattern recognition;
3. For picture processing;
4. For control functions within a multiprocessor systems;
5. Paging operations in TLBs;
6. In **ZISC** (**Z**ero **I**nstruction **S**et **C**omputer) devices;
7. In **Dataflow** machines, etc.

VIRTUAL MEMORY SYSTEM

Virtual memory (VM) deals with the main memory size limitations. The size of virtual storage is limited by

- the addressing scheme of the computer system,
- the amount of auxiliary storage available (not by the *actual number* of main storage locations).

Original motivation was to make a small physical memory look large and permit common software on a wide product line. Before virtual memory an overlaying *//ülekattuvus//* was used. Virtual memory automates the overlay process. Virtual memory *organization principles* are similar to the cache memory system.

Virtual memory is automatic address translation that provides decoupling *program's name space* from physical location and protection from interference with *name space* by other tasks.

Virtual Memory Architecture

Memory space in virtual memory is treated as a set of pages (blocks) that may be arbitrarily distributed among main memory and swap space.

1. Per process address space (*)
Each process is given an address space when created and disappears when process dies.

2. System-wide virtual address (not widespread)
One shared system-wide address space can persist over system lifetime.

VM Advantages

1. The **memory seen** by program can be greater than physical memory size:
   - **a.** * All programs can start at address "0" without being remapped by loader or by base register;
   - **b.** Machine need not have large contiguous address space for program;
   - **c.** Can leave huge holes in address space with no penalty;
   - **d.** Memory allocation is simple.
2. Programmer does not have to worry about managing physical address space (overlay**s**) and the automatic data transfer process maximize average speed**.**

Virtual memory main components are:

1. **Physical memory** divided up into **pages**;
2. **A swap device** that holds pages not resident in physical memory;
3. **Address translation mechanism**:
   - **3.1. Page tables** to hold virtual-to-physical address mappings;
   - **3.2. Translation Lookaside Buffer** (**TLB**).
4. **Management software** in the operating system.

Virtual Memory Main Principles

The working set *//töökogum, -komplekt//* is the group of physical memory pages currently dedicated to a specific process.

Page placement technique in VM

Page placement is **managed by system software**. Page placement consists of:

1. **Page identification;**
2. **Address translation (mapping).**

Let V be the set of virtual addresses (logical addresses) generated by a program running on a processor and let M be the set of physical addresses allocated to run this program.
A virtual memory system demands an automatic mechanism to implement the mapping φ(V)

$$\varphi: V \rightarrow M \cup \emptyset, \quad \text{where } \varphi(v) = m, \text{ if } m \in M \text{ or } \varphi(v) = \emptyset, \text{ if data is missing in } M.$$

The mapping uniquely translates the logical address L_A into a physical address P_A $[L_A \rightarrow P_A]$, if there is a memory hit in M. When there is a memory miss in M, the value returned, signals that the referenced item has not been brought into main memory yet.
A virtual address miss is called page fault.

Page fault handling:

Page fault is the sequence of events occurring when a program attempts to access data that is in its logical address space, but is not currently located in the main memory.

The efficiency of the address translation process affects the performance of the virtual memory.

Virtual memory has an automatic address translation that provides:

1. **Decoupling** of program's **name space** from **physical location**;
2. Provides **access to name space** potentially greater in size than physical memory;
3. **Expandability** of used name space without **reallocation** of existing memory;
4. **Protection** from interference with name space by other tasks.

Memory Management Unit (MMU)

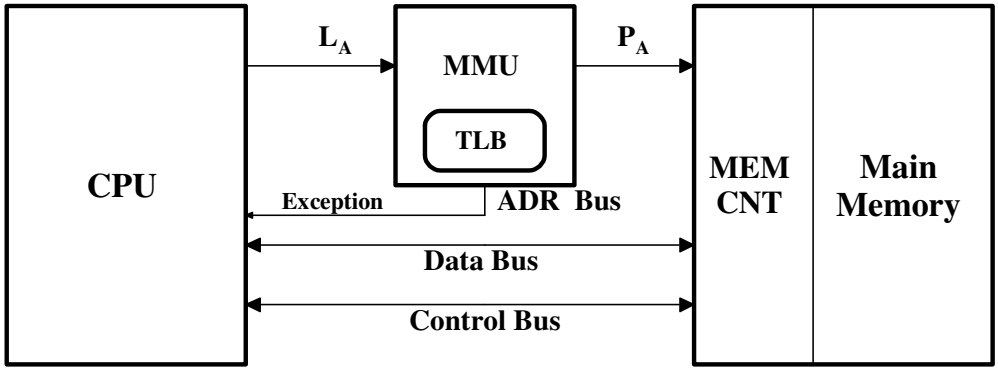
MMU is a hardware component, which is responsible for handling memory accesses requested by the CPU. MMU also solves the problem of fragmentation of memory. The main functions of MMU are:

1. The translation of virtual addresses to physical addresses;
2. Memory protection;
3. Cache system control;
4. Bus (usually memory bus) arbitration;

5. In simpler computer architectures (8-bit systems) bank switching.

There are two control registers in MMU – base register and bound register.

The **base register** //*baasiregister*// specifies memory location starting address.
The **bound register** //*piiriregister*// specifies size of address space.
The content of base and bound registers are changed by OS.



Memory Management Unit

- A logical address (L_A) is the location of a word relative to the beginning of the program.
- A logical address consists of a page number and a relative address within page (offset).
- A physical address (P_A) consists of a frame (physical memory page) number and a relative address within frame (offset).

When a problem arises, as for page fault, the MMU signals an exception so the OS can intervene and solve the problem.

The common approaches to the **memory mapping** are:

1. **Pagination (paging);**
2. **Segmentation;**
3. **Paged segmentation**.

The MMU contains a page table (PT). Each page table entry (PTE) gives the physical page number corresponding to the virtual one. This is combined with the page offset to give the complete physical address. Usually there is one page table per process. Page tables are usually so long that they are stored in main memory, and sometimes are paged themselves.
A PTE may also include additional information about whether the page has been written to, when it was last used (for a least recently used replacement algorithm), what kind of processes may read and write it, and whether it should be cached.

Page Table Entry Format:

1. **Address**;
2. **Control information,** which consists of:

1. **Valid/Present (V/P) bit**
   Set if the page being pointed to is resident in memory;
2. **Modified/Dirty (M/D) bit**
   Set if at least one word in page/segment has been modified;
3. **Reference (A/R) bit**
   Set if page or segment has been referenced;
4. **Protection (PR) bits**
   They are used to restrict accesses.

Usually is used **2-level page table format** (**Page Directory** (**Catalog**) + **Page Table(s)**).

↓ **In MMU** ↓ **In Main Memory**

Memory Management Techniques

Technique	Description	Comment
Fixed Partitioning	Main memory is divided into a number of partitions at system generation time. **A process may be loaded into a partition equal or greater size.**	Simple to implement. Little OS overhead. Inefficient use of memory.
Dynamic Partitioning	Partitions are created dynamically. **Each process is loaded into a partition of exactly the same size as that process.**	Inefficient use of processor. (additional overhead)
Simple Paging*	Main memory is divided into a number of **equal-size frames** (blocks or pages). **Each process is divided into a number of equal-size pages of the same length**.	
Simple Segmentation*	**Each process is divided into a number of segments**. A process is loaded by loading **all of its segments into dynamic partitions that need not be contiguous.**	Improved memory utilization.
Virtual-memory Paging	As with a simple paging, except that that **it is not necessary to load all of the pages of a process**.	Large virtual address space. Overhead of complex memory management.
Virtual-memory Segmentation	As with a simple segmentation, except that **it is not necessary to load all of segments of a process.**	Large virtual address space. Overhead of complex memory management.

* - Simple paging and simple segmentation are not used by themselves.

When segmentation is used with paging, a virtual address has three components:
A segment index (SI), a page index (PI) and a displacement (D).
In this case every memory address is generated by a program goes through a two stage translation process:

Virtual address → Linear address → Physical address

Paging *versus* Segmentation

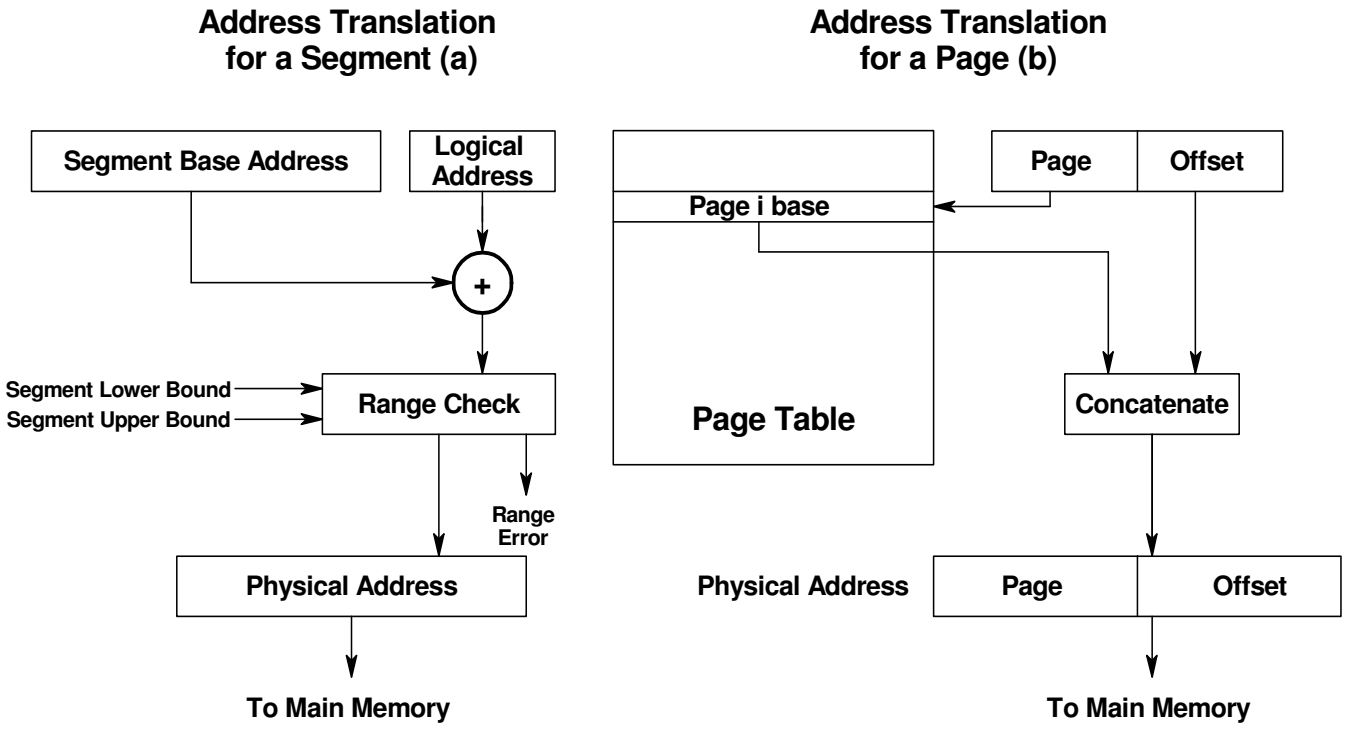
	Paging	**Segmentation**
Words per address	One (fig. **b***)	Two (fig, **a***) (segment & logical address)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	*Internal fragmentation* (unused portion of page)	*External fragmentation* (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

* *see* "Address Translation Styles in Virtual Memory"

Small Page Size Features:

- The large amount of pages in main memory;
- Less amount of internal fragmentation;
- More pages required per process;
- More pages per process ⇒ larger page tables;
- Large page tables' ⇒ large portion of page table is in secondary (slower) memory.

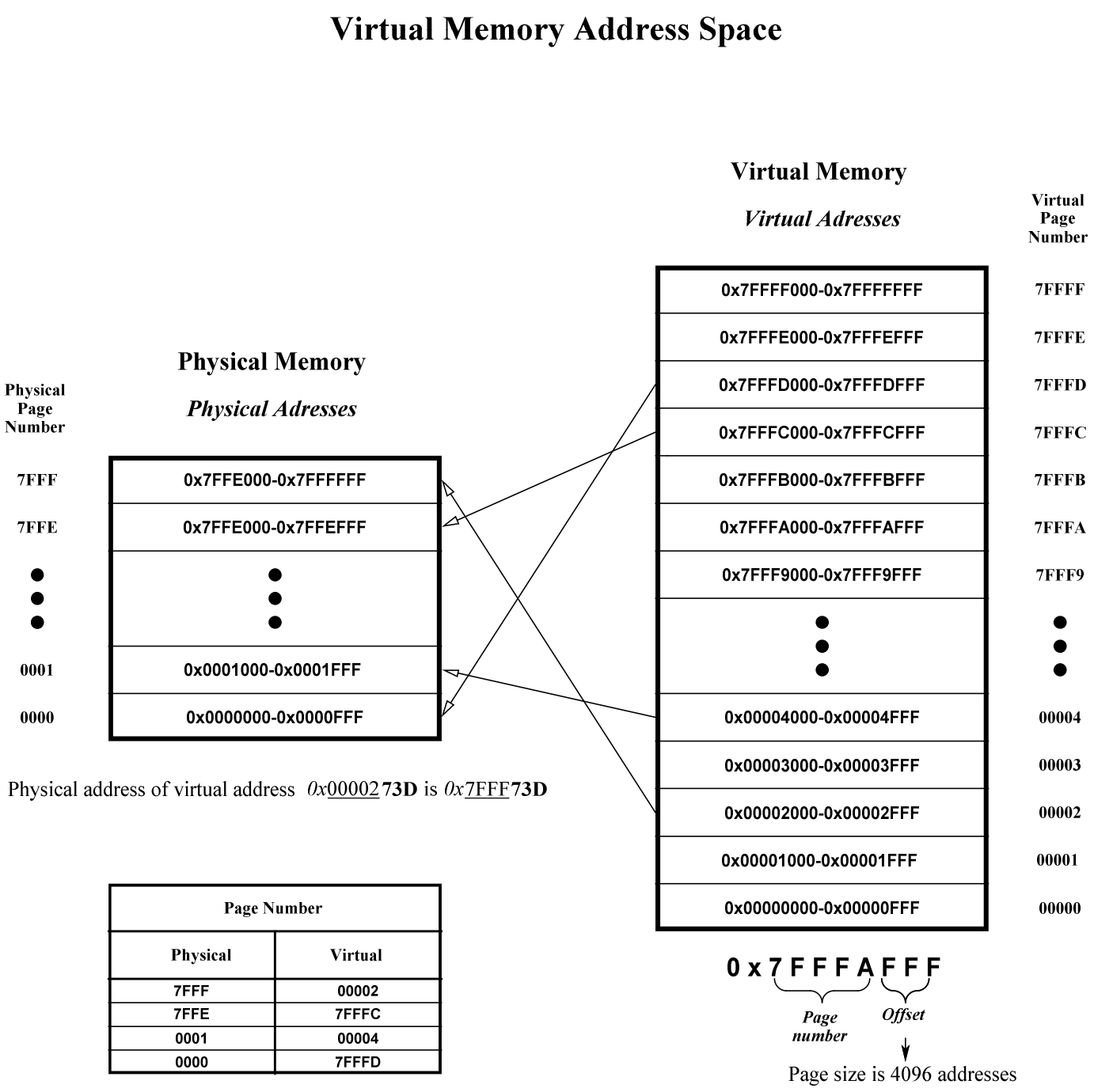
Address Translation Styles in Virtual Memory



Program Execution in Virtual Memory

1. OS brings into main memory some pieces of the program (resident set).
Resident set – a portion of process that is in main memory.
2. An interrupt is generated when a data (address) is needed that is not in main memory.
3. OS places the process in a *Blocking State*.
4. New data is brought into main memory from external (secondary) memory (disk).
Another process is dispatched to run while the disk I/O takes place.
5. An interrupt is issued when disk I/O complete, it causes the OS to place the affected process in the *Ready State*.
5. Each process has its own page table. The entire page table may take up too much main memory.
6. Page tables are also stored in virtual memory. When process is running, part of its page table is maintained in main memory.

Example

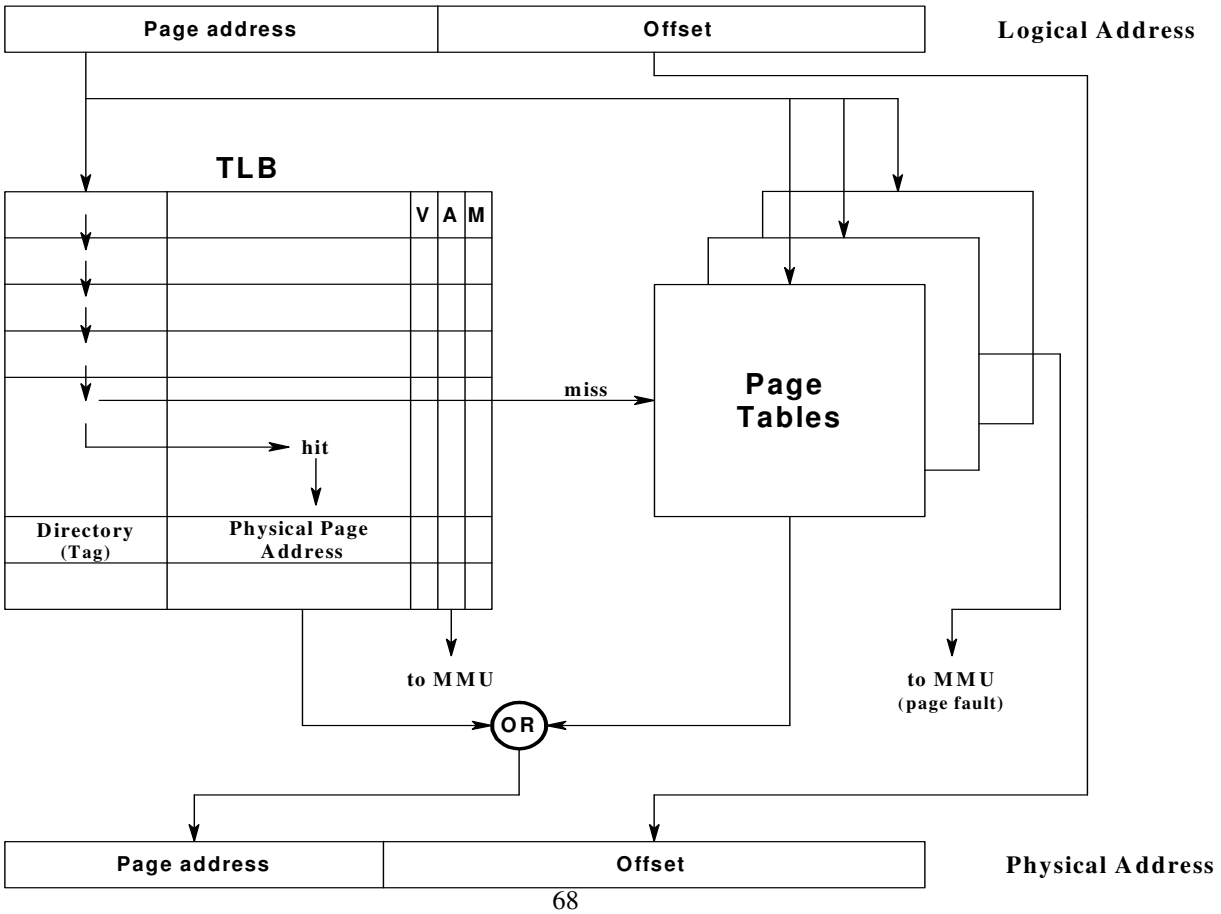

Virtual Memory Address Space
Virtual Memory
Virtual Adresses
Virtual Page Number
0x7FFFF000-0x7FFFFFFF
7FFFF
0x7FFFE000-0x7FFFEFFF
7FFFE
0x7FFFD000-0x7FFFDFFF
7FFFD
0x7FFFC000-0x7FFFCFFF
7FFFC
0x7FFFB000-0x7FFFBFFF
7FFFB
0x7FFFA000-0x7FFFAFFF
7FFFA
0x7FFF9000-0x7FFF9FFF
7FFF9
0x00004000-0x00004FFF
00004
0x00003000-0x00003FFF
00003
0x00002000-0x00002FFF
00002
0x00001000-0x00001FFF
00001
0x00000000-0x00000FFF
00000
Physical Memory
Physical Adresses
Physical Page Number
7FFF
0x7FFE000-0x7FFFFFF
7FFE
0x7FFE000-0x7FFEFFF
0001
0x0001000-0x0001FFF
0000
0x0000000-0x0000FFF
Physical address of virtual address 0x00002 73D is 0x7FFF 73D
Page Number
Physical
Virtual
7FFF
00002
7FFE
7FFFC
0001
00004
0000
7FFFD
0 x 7 F F F A F F F
Page number
Offset
Page size is 4096 addresses

Accelerating Address Translation

The first hardware cache used in a computer system did not cache the contents of main memory but rather translations between virtual memory addresses and physical addresses.
This cache is known as **Translation Lookaside Buffer** (**TLB**) or **Translation Buffer**.

TLB: **T**ransfer **L**ookaside **B**uffer or
Transfer **L**ookahead **B**uffer or
Address **C**ache **M**emory (**ACM**) or
Address **T**ranslation **C**ache (**ATC**}.

The TLB is used in the memory control unit for fast physical address generation, to hold recently used page or segment table entries. The TLB is used in the memory control unit for fast physical address generation.
The TLB is a high-speed look-up table which stores the most recently referenced page entries.
The first step of the translation is to use the virtual page number as a key to search through the TLB for a match. In case of a match (a hit) in the TLB, the page frame number is retrieved from the matched page entry.
In case the match cannot be found (a miss) in the TLB, a pointer is used to identify one of the page tables where the desired page frame number can be retrieved.
Classical TLB is fully associative.
In a TLB entry the tag portion holds portions of the virtual memory address and the data portion holds a physical page frame number (address), control bits (valid (V), use (A), dirty (M)) and protection information.



To change the physical page frame number or protection of an entry in the page table, the OS must make sure the old entry is not in the TLB. The dirty bit (M) means that the corresponding page is dirty, not that the address translation in the TLB is dirty.
The OS resets these bits by changing the value in the page table and then invalidating the corresponding TLB entry.

Newer processor architectures are supporting TLBs that allow each entry to be independently configured to map variable-size superpages.
The most serious problems associated with general utilization of superpages are – the requirement that superpages be used only to map regions of physical memory that are appropriately sized, aligned and contiguous.
Using superpages is restricted they are used for mapping a small number of large non-paged segments, such as frame buffer and the non-pageable portions of the OS kernel.

Virtual Memory is quite similar to Cache Memory

	CACHE MEMORY	VIRTUAL MEMORY
Implemented with	Cache SRAM	Main Memory DRAM
Baseline access via	Demand fetching	Demand paging
Misses to go	Main Memory	Swap device (disk)
Size of transferred data set	Block (4-64 bytes)	Page (4kB-64KB)
Mapping stored in	Tags	Page tables

The high order bits of the virtual address are used to access the TLB and the low order bits are used as index into cache memory.

Page Replacement Policies in Virtual Memory

A replacement policy is needed to determine which page is swapped out when memory space is needed. Replacement policy is supported by OS software. Because the swapping cost is great, it is desirable to use more complex algorithms to make selection.
There are six policies that are frequently used:

1. **LRU** - least recently used;
2. **FIFO** - first in first out;

3. **LIFO** - last in first out;
4. **Circular FIFO;**
5. **Random**;
6. **Belady`s optimal replacement policy**

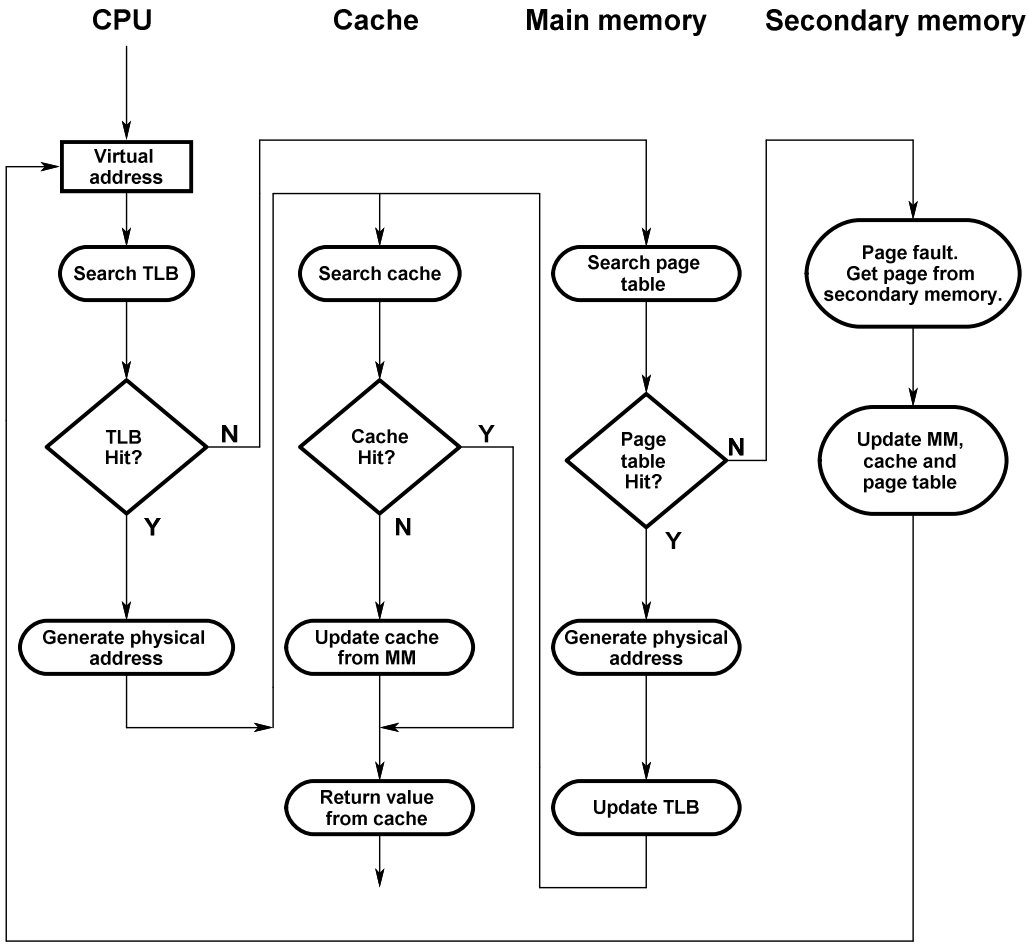
The **theoretically optimal replacement algorithm**) is an algorithm that works as follows: when a page needs to be swapped in, the OS swaps out the page whose next use will occur farthest in the future.

Write Strategy in Virtual Memory

Write-back (with page dirty bit)

Policy assumes that the compiler is unaware of the memory system. If the compiler has a large enough context to work with, it may be able to do a better job of managing memory.
It may be able to indicate that a phase of execution has just completed, a segment (page) of memory will no longer needed, even though it would not be replaced by LRU until much later.
But the compiler may also be able to initiate prefetch of a page or segment far enough in advance to avoid much of the paging penalty.

Summary



CACHE MEMORY SYSTEM

Thesis

A: Modern CPUs have one concern – keeping the processor pipelines filled with instructions and data.

B: This is becoming an increasingly difficult task because:
The speed of CPU performance doubles every **18 – 24** month,
The speed of the DRAM chips, that constitutes main memory, increases by only a few (**5 - 7**) percent each year.

C: The high-speed **cache memory** that acts as a **buffer** between main memory and CPU is an increasingly significant factor in overall performance.

D: Cache consistency //*vahemälu konsistentsus, ~kooskõlalisus***//**

Cache consistency – in presence of a cache, reads and writes behave no differently than if the cache were not there.
The cache consistency reflects the fact that a datum must have one value and only one value.
If two requests to the same datum of the same time return different values, then the correctness of the memory system is in question. Cache consistency aspects are:

1. **Consistency with backing store**
2. **Consistency with itself**
3. **Consistency with other clients**

The principle of memory coherence indicates that the memory system behaves rationally.
Memory consistency defines rational behavior, the consistency model indicates how long and in what way the system is allowed to behave irrationally with respect to a given set of reference.

Memory consistency behaves like a superset of memory coherence.

E: The **property of locality** which **justifies the success of cache memory** has two aspects:

1. **Locality in time** (**temporal locality**) – a hit that occurs on a word already in cache that has been previously accessed.

Temporal locality for code
Low $\Rightarrow$ no loops and no reuse of instructions;
High $\Rightarrow$ tight loops with lots of reuse.

2. **Locality in space** (**spatial locality**) – a hit that occurs on a word already in cache that has not been previously accessed.

Spatial locality for code

Low $\Rightarrow$ lots of jumps to far away places;
High $\Rightarrow$ no branches or jumps at all.

F: The **performance speedup of a cache** ($\mathbf{S_{cache}}$):

$$S_{cache} = \frac{T_m}{(1-h)\times T_m + h \times T_c} = \frac{1}{1 - h \times (1 - \frac{T_c}{T_m})},$$ where

$\mathbf{T_m}$ – main memory access time;
Tc – cache memory access time;
h – hit ratio,

a. $\mathbf{T_c = T_m \Rightarrow S_{cache} = 1.}$

b. $\mathbf{Tc << Tm \Rightarrow S_{cache}} = \frac{1}{1-h}$,

$h\uparrow \Rightarrow S_{cache}\uparrow$
$h\downarrow \Rightarrow S_{cache}\downarrow$

Cache Memory Fundamentals

Cache types

1. **Data cache**
2. **Instruction cache & trace cache //***jäljevahemälu***//**
3. **Address cache**

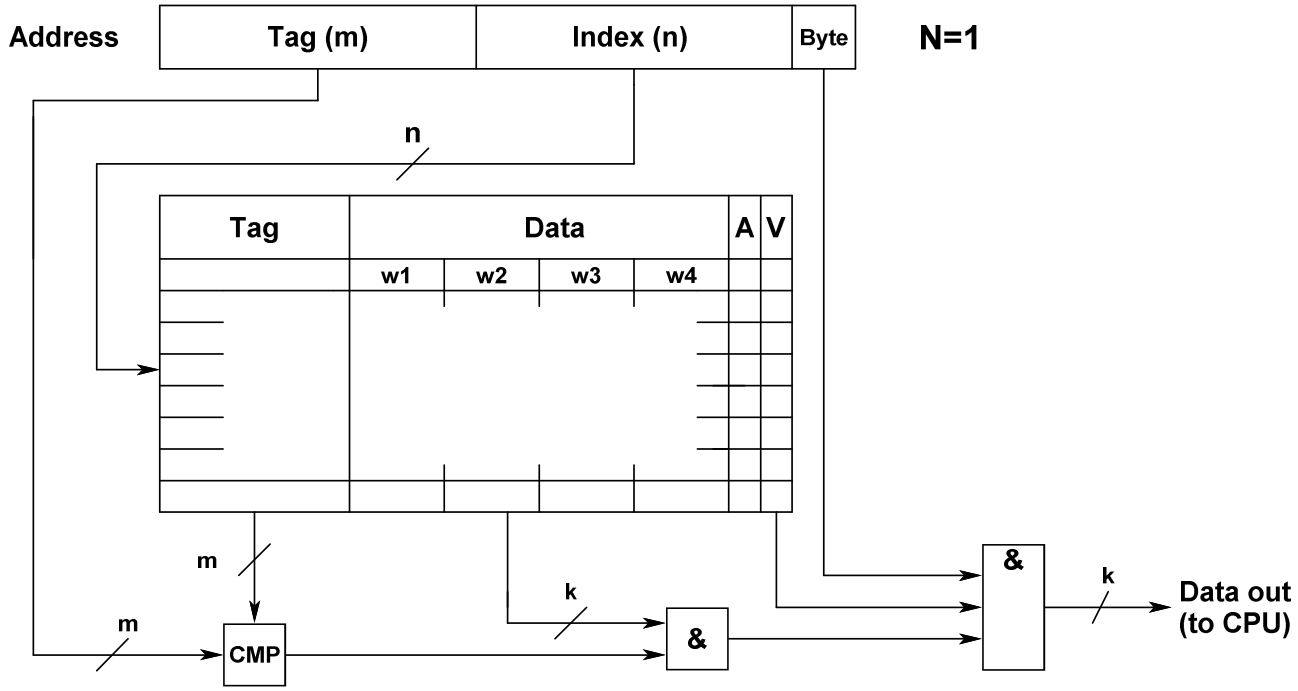
Cache memory o**rganization**

1. **Direct mapped cache** (SRAM)
2. **Set-associative cache** (SRAM)
3. **Fully associative cache** (CAM)

Main functional units

1. **Data memory (fast memory)**
2. **Tag memory (very fast memory)**
3. **Status memory (very fast memory)**
4. **Block (line) fetch circuits**
5. **Control circuits**

Direct Mapped Cache



Since there are more memory frames than lines in cache, an individual line cannot be uniquely and permanently dedicated to a particular frame. Therefore, each line includes a tag that identifies which particular frame of main memory is currently occupying that line of cache.

- **Address length (a):** **a = {(m+n)+w}** bits;
- **Number of addressable units in main memory:** 2^a words or bytes;
- **Block size (line size):** 2^w words or bytes;
- **Number of blocks in main memory:** $2^a / 2^w = 2^{n+m}$;
- **Number of blocks (lines, rows) in cache (r):** $r = 2^n$;

The mapping function gives the correspondence between main memory frames and cache memory lines. Each line is shared between several main memory frames.
Direct mapping cache maps each block of main memory into one possible cache line.
The mapping can be expressed as:

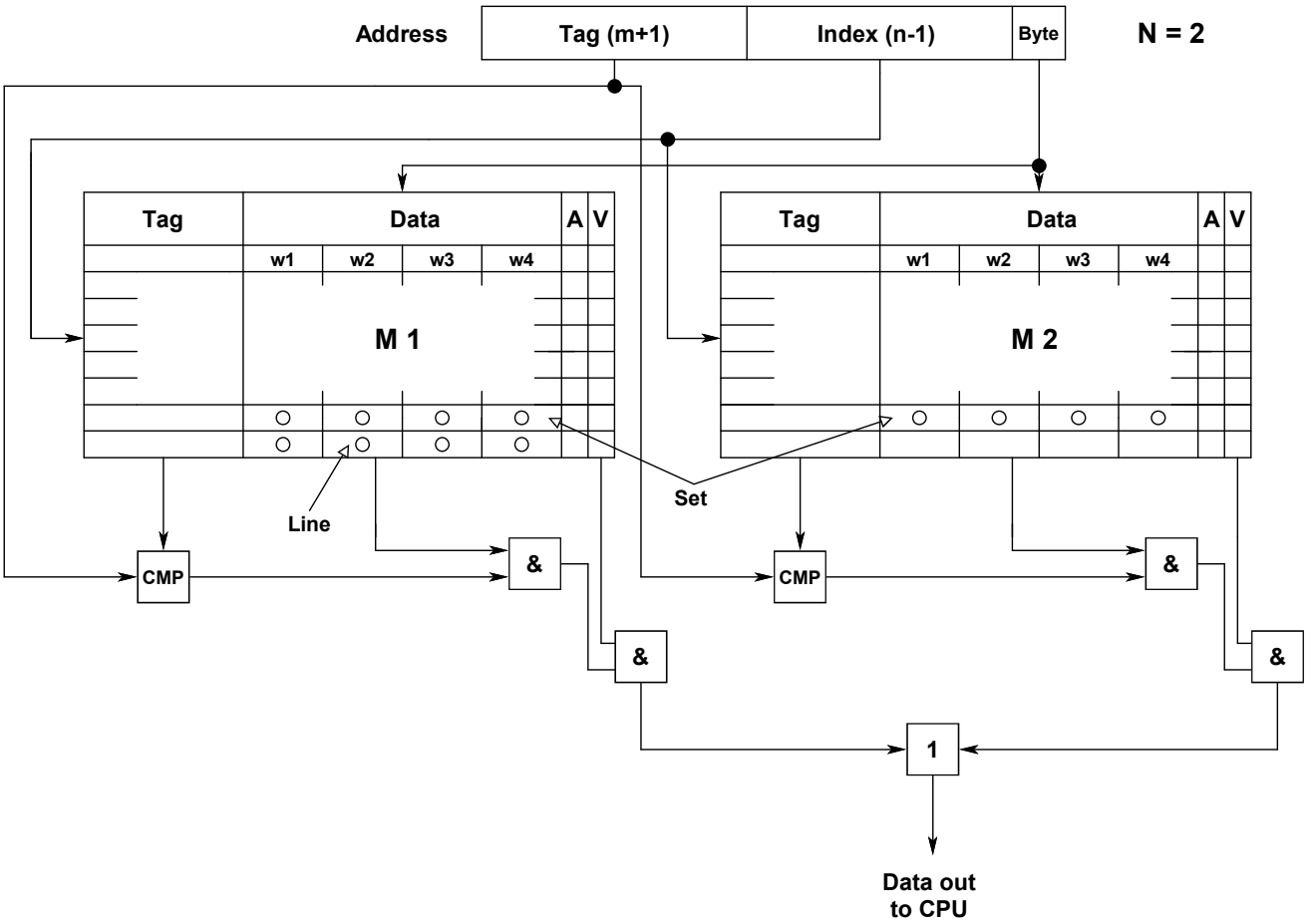
$$l = f \bmod r, \quad \text{where}$$

l – cache line number,
f – main memory block(frame) number.

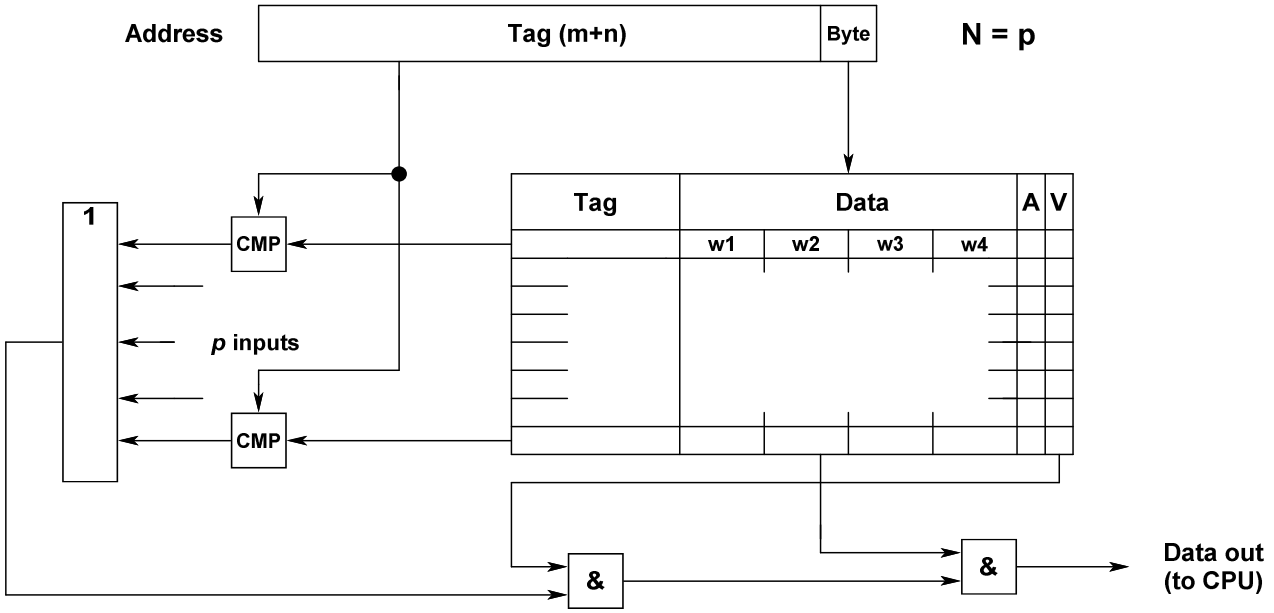
The tags contain the address information required to identify whether a word (data block) in the cache line corresponds to the requested word (data block). The tag uses the upper portion of the address.

There is no two blocks that map into the same line numbers have the same tag numbers.

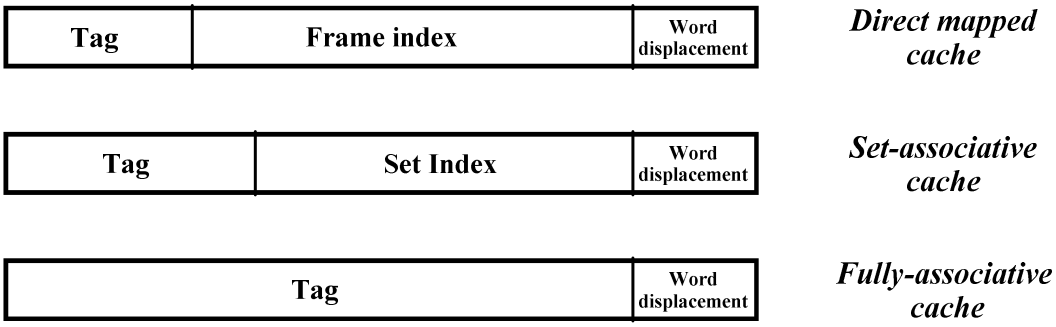
Set Associative Cache



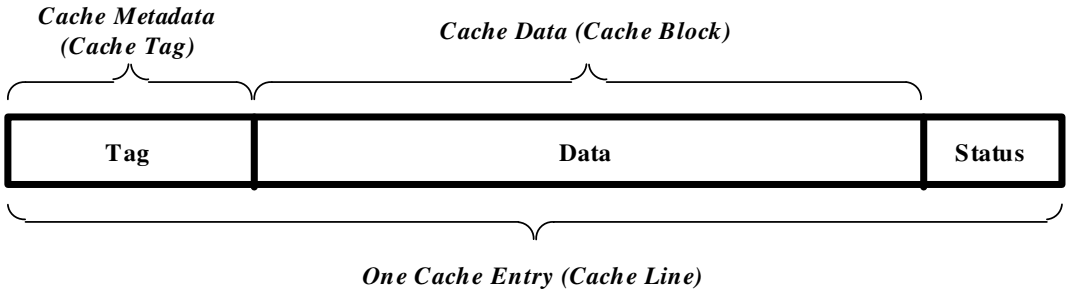
Fully Associative Cache



Cache Address Formats



Cache Entry Basic Structure



A fully-associative cache has two distinct advantages over set-associative cache:

1. **Conflict-miss minimization**;
2. **Global replacement**.

There is an inverse relationship between the number of conflict-misses in a cache and the associativity of the cache. A fully-associative cache minimizes conflict-misses by maximizing associativity.

Global replacement allows a fully-associative cache to choose the best possible victim every time, limited only by intelligence of the replacement algorithm.
Accessing a fully-associative cache is impractical; as it requires a tag comparison with every tag-store entry (prohibitively increases the access latency and power consumption.

Dirty Data – when data is modified within cache but not modified in main memory, then the data in the cache is called "dirty data".

NB! The ***dirty bit*** is needed only in **write-back caches**.

Stale Data – when data is modified within main memory but not modified in the cache, the data in the cache is called "stale data".

Data Transfer in Cache System

a. In asynchronous cache

The oldest and slowest cache type,

Data transfers are not tied to the system clock.

b. In synchronous cache

Data transfers are tied to the memory bus clock**.**

c. In pipelined burst cache

Cache has special circuitry that allows the four data transfers occur in a burst. The second transfer begins before that the first one is finished.

Cache type	Transfer formula (w1, w2, w3, w4)
Asynchronous	**3 - 2 - 2 - 2**
Synchronous	**3 - 2 - 2 - 2**
Pipelined burst	**3 - 1 - 1 - 1**

Cacheable and Non-Cacheable Data

Data that **can be cached** (**written into cache**) is a **Cacheable Data,** whereas data that cannot be cached is a **Non-cacheable Data**.

Usually non-cacheable data is dynamic information that changes regularly or for each user request and serves no purpose if it were cached (as for web pages that return the results of a search, because their contents are unique almost all the time).

Cache Memory Main Parameters

1. **Cache size** – is the total capacity of the cache
2. **Block** (**line**) **size** – is the data size that is both:
3. **Associativity** – partition cache blocks into **s equivalence classes** (**s** - the **number of sets**) (**Set size**) of a **n block frames each**.

Set - storage for lines with a particular index.

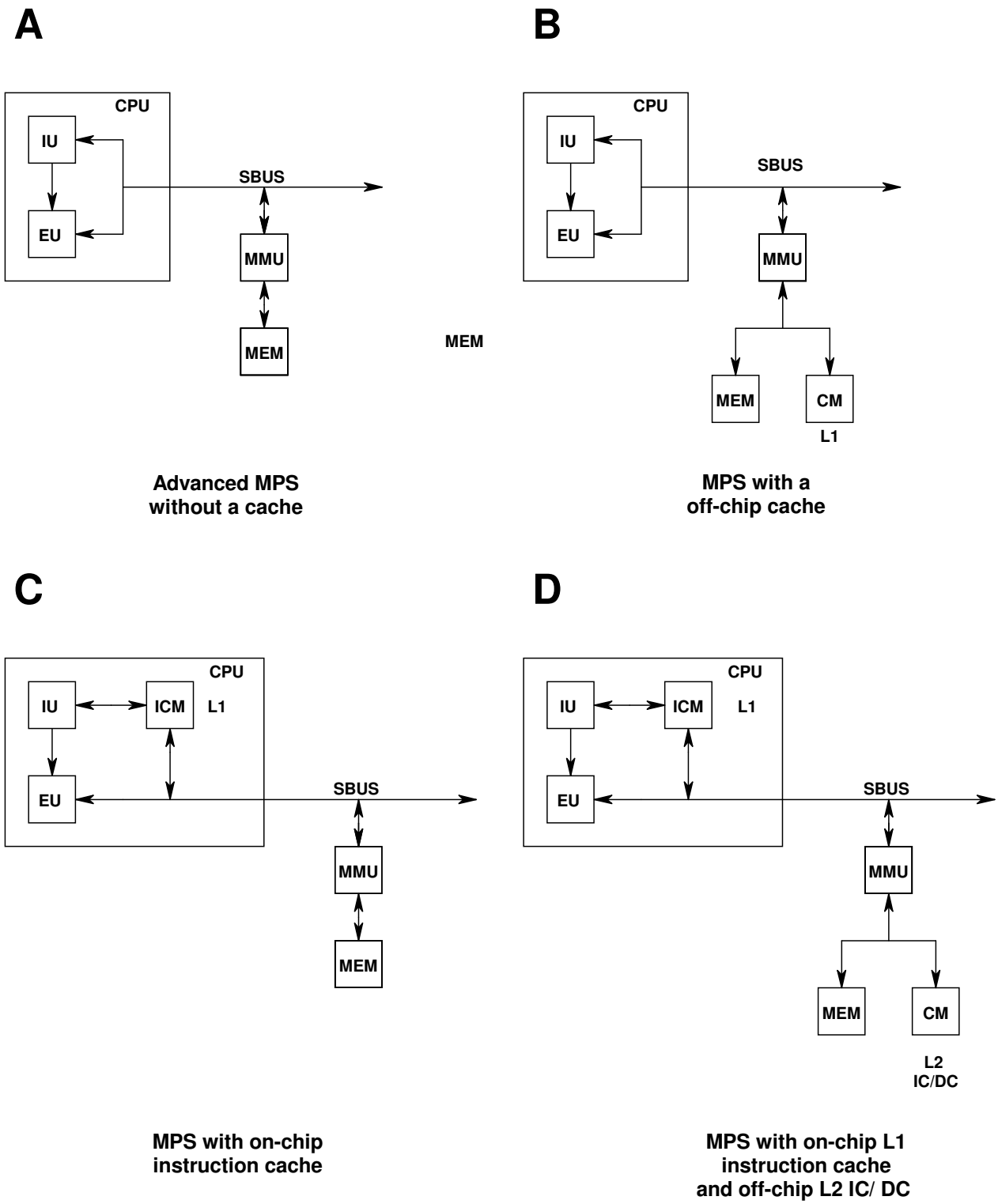
Typical values for **cache associativity**:

1. **Direct mapped** – **s=1**,
2. **S-way set-associative** – **s=2**,**4**,**8**,**16**,
3. **Fully-associative** – **s= all blocks**.

Higher associativity means lower miss rates, smaller associativity means lower cost, faster access (hit) time, but higher miss rate.

Higher levels of associativity (**s ≥ 32**) requires hardware overhead that slows down the cache, it is often the case that low levels of associativity with a larger capacity provide better performance overall.

Cache Memory System's Historical Development



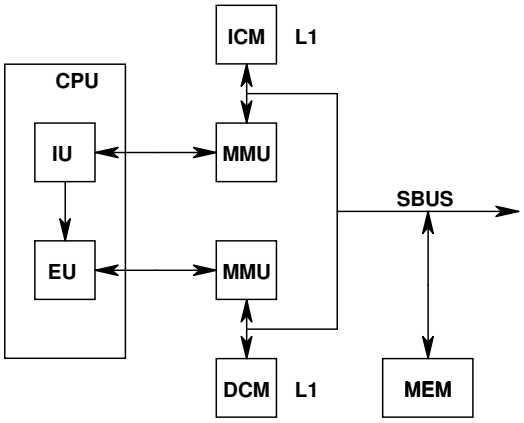
Advanced MPS without a cache

MPS with a off-chip cache

MPS with on-chip instruction cache

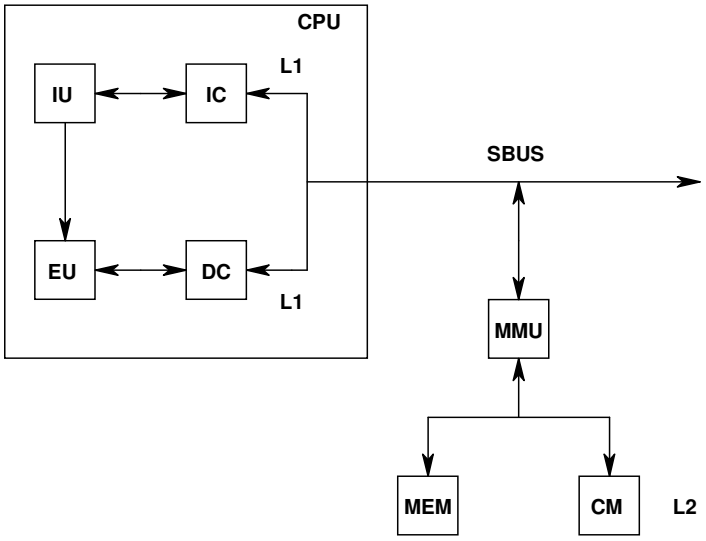
MPS with on-chip L1 instruction cache and off-chip L2 IC/ DC

E



**Dual-bus cache system
with off-chip
IC and DC caches**

F



**MPS with on-chip L1 IC and DC caches
and off-chip L2 IC/DC cache**

H

**Advanced MPS with
on-chip multilevel (L1, L2, (L3))
instruction (IC) and data (DC) caches
and off-chip multilevel
separated or united instruction/data caches**

Cache Memory Miss Types

Cache miss can occur during **read** and **write operations**.

1. **Compulsory** (cold start miss, the first reference miss) **miss**

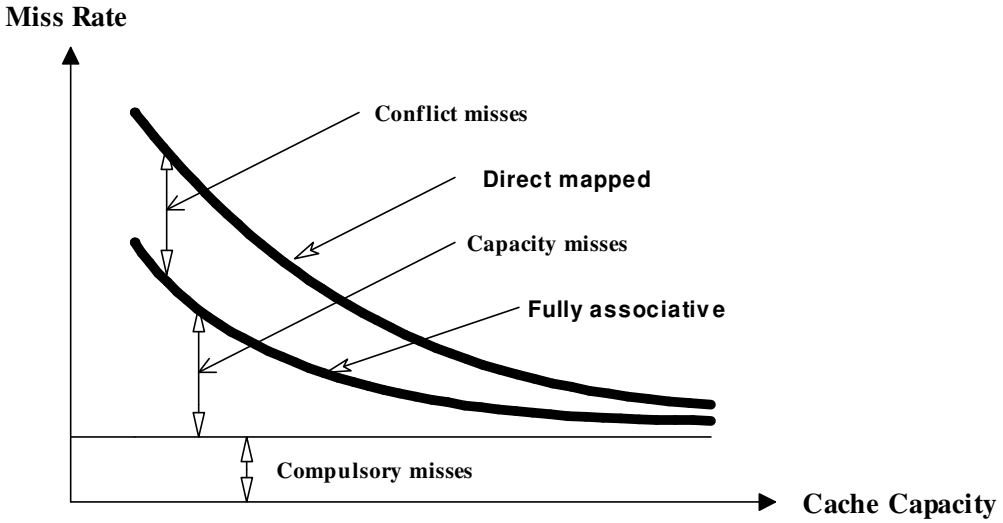
[*misses in infinite size of cache*] It occurs during the first access to a block that is not in the cache yet.

2. **Capacity miss**

[*misses due to size of cache*] The cache cannot contain all the data blocks needed during execution of a program. They will occur due to data blocks being discarded and later retrieved.

3. **Conflict** (collision miss, interference miss) **miss**

[*misses due to associative and size of cache*] Multiple memory locations mapped to the same cache location. Fully-associative placement avoids conflict misses.



Reducing Cache Misses

1. Large block size (*reduces compulsory and capacity misses*);
2. Large cache size (*reduces capacity and conflict misses*);
3. Higher associativity (*reduces conflict misses*)

Improving Cache Performance

- **Multi-ported cache** is a cache implementation where the cache provides for more than one Read or Write port for providing high bandwidth. Because of these multiple ports it results in servicing multiple requests per cycle.
- **Multi-banked cache** is a cache implementation where the cache is implemented as a banked structure for providing high bandwidth by providing the illusion of multiple ports. It results in servicing multiple requests per cycle if there are no bank conflicts.

- **Prefetching** is a technique that predicts soon-to-be used instructions or data and loads them into the cache before they are accessed.
  Prefetch is a technique used to avoid or reduce the time the processor is waiting for data to arrive in the registers. A commonly used method is sequential prefetching where it is predicted that data or instructions immediately following those currently accessed will be needed in the near future and are prefetched. Prefetching, especially aggressive prefetcing may reduce performance.

Non-blocking cache is a cache that allows the processor to make references to the cache while the cache is handling an earlier miss. Non-blocking cache is commonly used to hide the cache miss latency by using out-of-order processors. The processor continues to operate until one of the following events takes place:

1. Two cache misses are outstanding and a third load/store instruction appears;
2. A subsequent instruction requires data from of the instructions that caused a cache miss.

Cache Memory Fundamental Questions

1. **Block placement** (where can a block be placed in the cache?);
2. **Block identification** (how is a block found in the cache?);
3. **Block replacement**[2] (which block should be replaced on a miss?);
4. **Cache type** (what type of information is stored in the cache?).
5. **Write strategy** (what happens on a write?).

Write–through and Write-back Strategy

When to propagate new value to lower memory level - Write-through: immediately, Write-back: when block is replaced.
Unlike instruction fetches and data loads, where reducing latency is the prime goal, the prime goal for writes that hit in the cache is reducing the bandwidth requirements.

The write traffic into L2-level cache primarily depends on whether the L1-level cache is write-through (store-through) or write-back (store-in or copy-back).

[2] Not needed for direct mapped caches.

Write-through and Write-back Caches Advantages and Disadvantages

Feature	Write-through	Write-back
Traffic	More	Less
Additional Buffer	*Write-buffer* needed	*Dirty victim buffer* needed
Ability to handle Burst Writes	Write buffer can overflow	Usually OK, unless miss with dirty victims
Cycles required per write	1	1 to 2 (include probe)

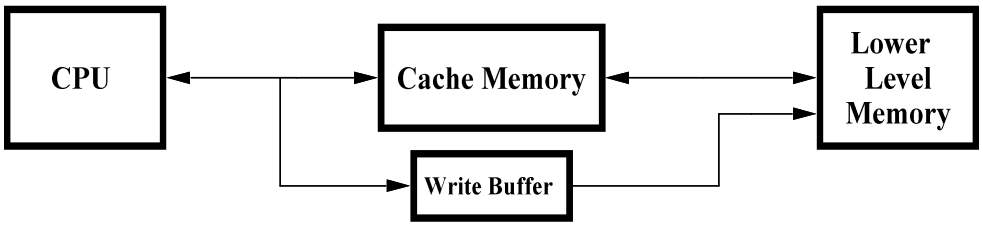
What should be done on a **write operation** that does **not hit in the cache (write miss)**? There are two common options on a write miss:

1. **Write allocation policy -** the cache line to which the write miss occurred is brought into the cache and also the block in main memory is updated.
   Method decreases read misses, but it requires additional bus bandwidth.
2. **Write no-allocation policy -** the write is propagated to main memory without changing the contents of the cache.
   Method generates potentially more read misses, but it needs less bus bandwidth.

Write-back caches generally use write allocate policy and write-through caches often use no-write allocate policy.

Avoiding Write-Through Write Stalls

Write stall – if CPU has to wait for write to complete during write through.
To reduce write stalls, write buffer is used, allowing CPU to proceed while memory is being updated.



Write buffer holds data awaiting write-through to lower level memory. The bursts of writes are used in write buffer. For eliminating RAW hazard it must be drained write buffer before next read, or check write buffer:

On reads, CPU checks cache and write buffer.
On writes, CPU stalls for full write buffer.

In write-through cache it helps with store misses. In write-back cache it helps with dirty misses. It allows to do read first. For this a write dirty block is sent to the write buffer. The new data block is read from memory to cache and then the content of write buffer is sent to memory.

Cache Miss Rate Reduction

Cache hit time – time to deliver a line in the cache to the processor (includes time to determine $\mathbf{t_{hit}}$ whether the line is in the cache)

A. Victim Cache

Victim cache is a small, fully-associative cache inserted between cache memory and its refill path. It reduces need to fetch blocks from higher-level memory by keeping recent victims, blocks discarded from cache memory. Victim cache reduces conflict misses.

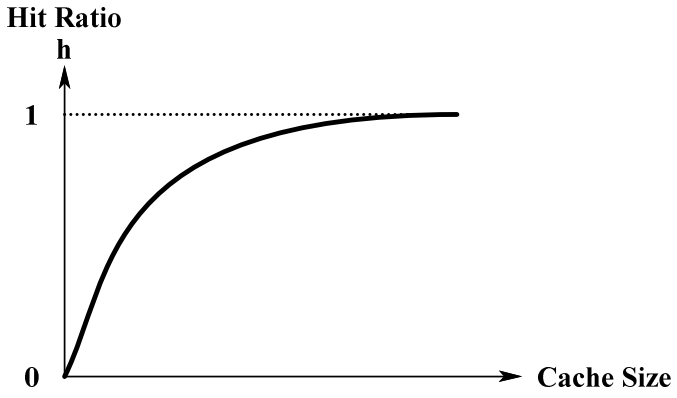
B. Pseudo-associative Cache

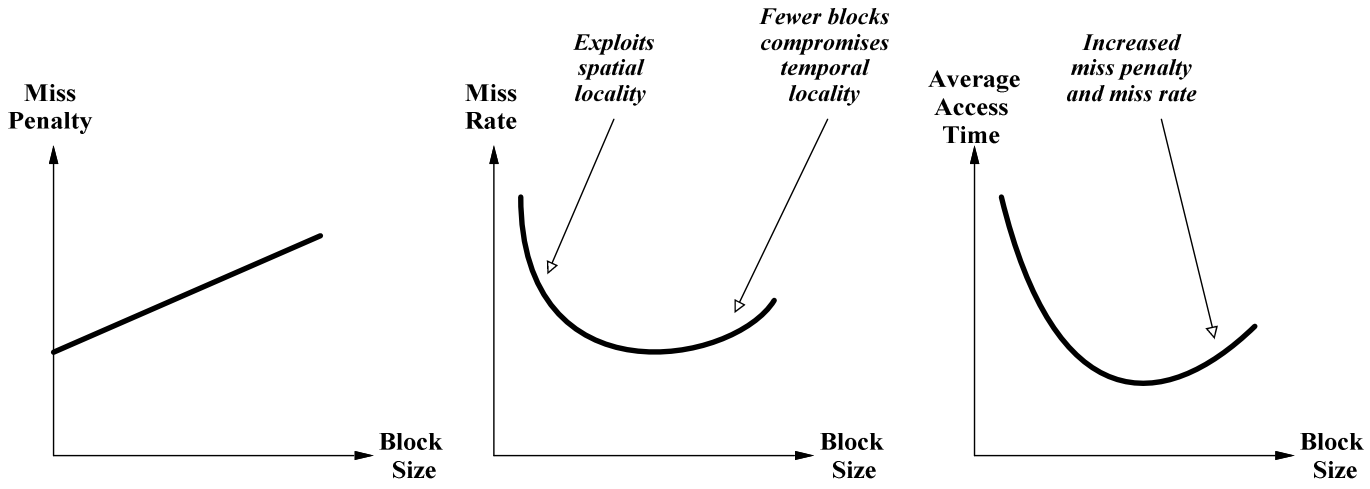
For getting the miss rate of set-associative caches and the hit speed of direct mapped is realized in pseudo-associative cache. When a hit occurs, the cache works like the direct-mapped cache. On a miss, before going to the next lower level of the memory hierarchy, another cache entry is checked to see if it matches there.

Hit Ratio *versus* Cache Size, Cache Type, Associativity and Block Size

30 Percent Rule

Doubling the cache size increases hit rate by about 30 percent.





Misses per 1000 Instructions for Instruction, Data and Unified Caches

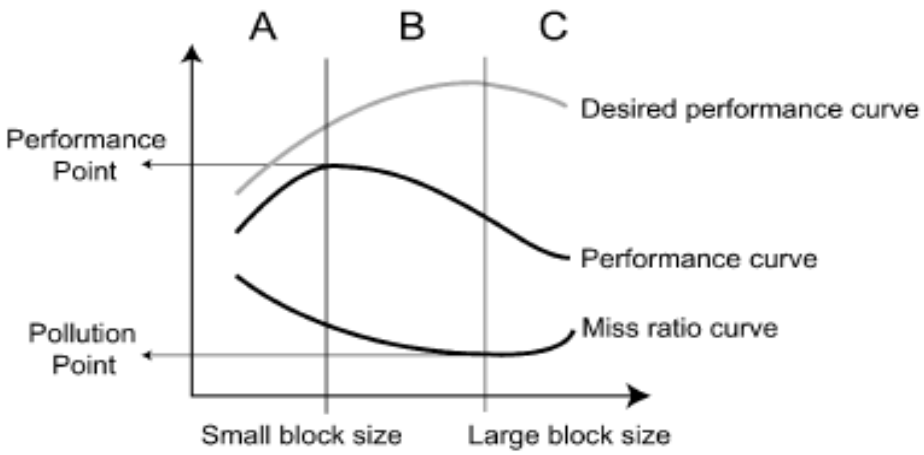
Cache Size	Instruction cache	Data cache	Unified cache
16 KB	3,82	40,9	**51,0**
64 KB	0,61	36,9	**39,4**
128 KB	0,30	35,3	**36,2**
256 KB	0,02	32,6	**32,9**

Associativity (↑)	**Block size (↑)**	**Capacity (↑)**
Decreases conflict misses (+)	Decreases compulsory misses (+)	Decreases capacity misses (+)
	Increases or decreases capacity misses (±)	
	Increases conflict misses (-)	
Increases $\mathbf{t_{hit}}$ (-)	No effect on $\mathbf{t_{hit}}$ (~)	Increases $\mathbf{t_{hit}}$ (-)

Block size, Performance Point and Pollution Point

For any cache, as the block size is increased, the effects of bandwidth contention will eventually overwhelm any reduction in miss rate.

Performance point – is the **block size** at which **performance is highest**.
Pollution point – is the **block size** at which the **miss rate is lowest**.



In region A of the figure, larger blocks improve the miss rate, which provides a performance gain that overcomes the performance drop due to increased channel contention, causing overall performance to rise.

In region B, the miss ratio continues to drop with larger blocks, but performance also deteriorates due to increased contention. The performance point resides at the boundary between regions A and B.

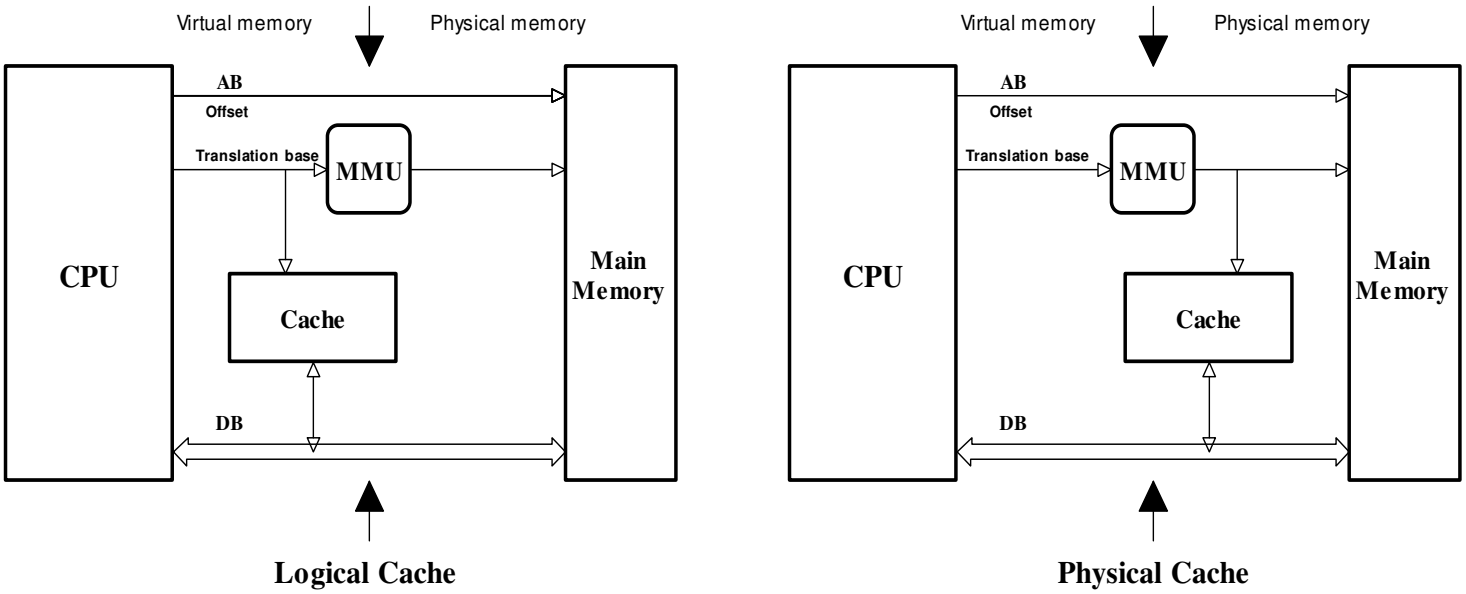
In region C, pollution causes the miss rate to begin to increase with larger blocks, causing a sharper drop in performance than in region B. The pollution point resides at the boundary between regions B and C.

Virtual Indexing and Virtual Tagging

The existence of different physical and virtual addresses raises the question of whether a particular cache is virtually or physically indexed, and whether it is virtually or physically tagged.

Virtual Indexing

Virtual indexing has lower latency, because the physical address is available only after the TLB has performed its translation. Most L1-level caches are virtually indexed.



Virtually indexed and/or tagged caches keep aliased //*rööpnimesus*// virtual addresses coherent by guaranteeing that only one of them will be in the cache at any given time. It is guaranteed by OS that no virtual aliases are simultaneously resident in the cache.

Whenever a new entry is added to a virtually indexed cache, the processor searches for any virtual aliases already resident and evicts them first. This special handling happens only during a cache miss. No special work is necessary during a cache hit.

Virtual Tagging

A physically tagged cache does not need to keep virtual tags. It is useful to distinguish the two functions of tags in an associative cache: they are used to determine which way of the entry set to select, and they are used to determine if the cache hit or missed.

If the TLB translation is slower than the L1-level cache access, then a virtually tagged cache will be faster than a physically tagged cache. It is the reason why some L1-level caches are virtually tagged.

The advantage of virtual tags is that, for associative caches, they allow the tag match to proceed before the virtual to physical translation is done.

Cache Memory Performance

The access latency in cache memories is not fixed and depends on the delay and frequency of cache misses. Average cache memory access time (effective latency) (t_{avg}) is equal to:

$$t_{avg} = t_{hit} + m_R \times t_{miss}$$
$$t_{miss} = t_{miss\ penalty};$$
$$m_R - \text{miss rate}$$

Miss penalty – time to replace a block in the higher level (closer to CPU) with a block from the lower level plus time to deliver this block's word to the processor.
Average memory access time (t_{avg}) can be calculated in **absolute time** or in **clock cycles**.

Impact of Cache Memory on the CPU Performance

Modified CPU performance equation to account for **CPU being stalled**

$$t_{CPU} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times t_{CLK}\ ,\quad \text{where}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times t_{miss\ penalty}\quad \text{or}$$

$$\text{Memory stall cycles} = IC \times \text{Memory references per instruction} \times m_R \times t_{miss\ penalty}$$

Cache can have enormous impact on performance, particularly for CPUs with low CPI and high clock rate (impact to CPU time!).

L2-Level Cache

L2-level cache contents are always superset of L1-level cache contents.

- We want large caches to reduce frequency of more costly misses.
- Large caches are too slow for processor.

Average access time in L2 cache (t_{avgL2}):

$$t_{avgL2} = t_{cacheL1} + miss_{L1} \times t_{cacheL2} + miss_{L1,2} \times t_{memory}$$

- The **speed of L1-**level cache dictates the CPU clock rate.
- The **speed of L2-**level cache will affect execution time.

Local miss rate - miss rate measured relative to the references arriving at a particular level of cache.
Global miss rate – overall miss rate of the **cache complex**.
Multilevel inclusion – describes a multiple level cache in which any data found in a higher level will also be find in the lower level.

Cache Coherency Mechanisms

Snooping – the process where individual caches monitor address bus for accesses to memory location that they have cached.

Snarfing – the process where individual caches monitor the address and data buses in an attempt update its own copy of a memory location, when it is modified in main memory.

Cache Memory *versus* **Virtual Memory**

Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	10^6-10^7 clock cycles
Access time	6-130 clock cycles	8×10^5-8×10^6 clock cycles
Transfer time	2-20 clock cycles	2×10^5-2×10^6 clock cycles
Miss rate	0,1-10 %	10^{-5}-10^{-3} %
Address mapping	25-45 bits physical addresses to 14-20 bits cache address	32-64 bits virtual address to 25-45 bits physical address

Memory Hierarchy Parameters Effects on Performance

Parameter variation	Advantage	Disadvantage
Large main memory or cache size	Fewer capacity misses	Longer access time
Large pages in main memory or longer cache lines	Fewer compulsory misses	Greater miss penalty
Greater cache memory associativity	Fewer conflict misses	Longer access time
More sophisticated data replacement policy	Fewer conflict misses	Longer decision time, more hardware
Write-through policy in cache memory	No write-back penalty, easier write-miss handling	Wasted memory bandwidth, longer access time

Summary

Cache Optimization Techniques

Technique	Miss penalty	Miss rate	Hit time	Hardware complexity	Comment
Multilevel caches	+			**2**	Costly HW; Harder if block size L1≅L2; Widely used
Giving priority to read misses over writes	+			**1**	Trivial for uniprocessor; Widely used
Victim caches	+			**1**	AMD's *Athlon* has eight entries
Larger block size	—	+		**0**	Trivial; *Pentium 4*'s L2-cache uses 128 bytes
Larger cache size		+	—	**1**	Widely used, especially for L2-caches
Higher associativity		+	—	**1**	Widely used
Pseudoasso-ciative caches		+		**2**	Used in L2-cache of MIPS *R10000*
Non-blocking caches	+			**3**	Used with all out-of-order CPUs
Compiler techniques to reduce cache misses		+		**0**	Software is a challenge; Some computers have compiler option
Hardware prefetching of instructions and data	+	+		**2 instructions**, **3 data words**	Many prefetch instructions; *UltraSPARC III* prefetches data
Compiler-controlled prefetching	+	+		**3**	Needs non-blocking caches; several processors support it
Small and simple caches		—	+	**0**	Trivial; Widely used
Pipelined cache access			+	**1**	Widely used
Trace cache			**+**	**3**	In *Pentium 4*

Summary

Cache and Main Memory Interaction

Access type	The number of access cycles			
	Cache ***Read***	**Cache** ***Write***	**Main Memory** ***Read***	**Main Memory** ***Write***
Read hit	**1**	**0**	**0**	**0**
Read miss	**1**	**n** (line size in words)	**n** (line size in words)	**0**
Write hit	**0**	**1**	**0**	**1**
Write miss	**1** (tag read)	**(0); [1]** (no cache update) [cache update]	**0**	**1**

INPUT-OUTPUT SYSTEM

Input/Output Bottleneck

The input/output bottleneck is a discrepancy between the large speed of processing elements in the system and the smaller speed of input/output elements of the system.
The main problem is that I/O transfers potentially have side-effects, such as initiating a DMA transfer, clearing status bits, or removing data from a receive queue. This forces I/O transfers to be done:

1. **In strict program order;**
2. **Non-speculatively;**
3. **Exactly once**.

The overall system performance is frequently limited by I/O devices.
The slowdown main reasons:

a. Monitoring of the I/O process consumes processor cycles;
b. If I/O supplies input data, then the processing must wait until data are ready.

Input/Output Units

Input/Output units are the computing system's means of communication with the world outside of primary (system) memory.

Features:

1. The I/O devices **vary** tremendously in **form** and **functions**.
2. The I/O devices **vary** enormously in the **speeds** at which they operate.
3. The I/O devices are the **main system resources that must be shared** among multiple users.

Functions:

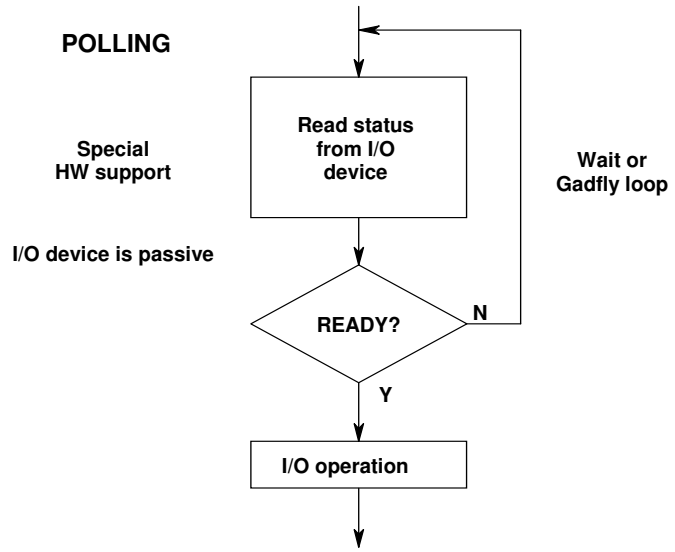
1. **Information transferring.**
2. **Sharing the I/O resources.**
3. **Handling and recovering errors in I/O operations.**

I/O Device Types

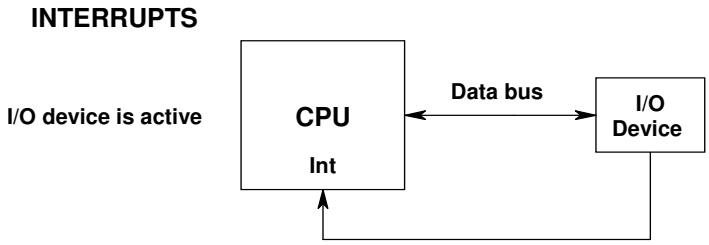
1. **Data presentation devices** at the **user interface** (*for* ***processor-user communications***);
2. **Data transport devices** at the **network interface** (*for* ***processor-processor communications***);
3. **Data storage devices** at the **storage interface** (*for* ***processor-storage communications***).

The I/O Devices Organization Ways

1. **Program-controlled;**

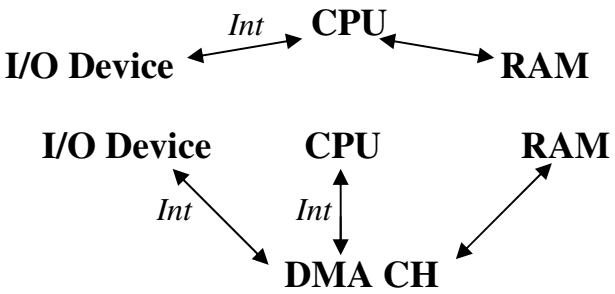


2. **Interrupt-driven;**
   (incl. **I/O processors**)



3. **DMA-managed**.
   (DMA controller managers single block transfers)

DMA CHANNEL



I/O Performance Measurement

I/O system performance common metrics are:

1. **Throughput** (**I/O bandwidth**) ⇒ useful for file servicing and transaction processing ;
2. **Response time** (**latency**).

Response time = Controller time + Wait time + $\frac{Number\_of\_bytes}{Bandwidth}$ + CPU time – Overlapping time

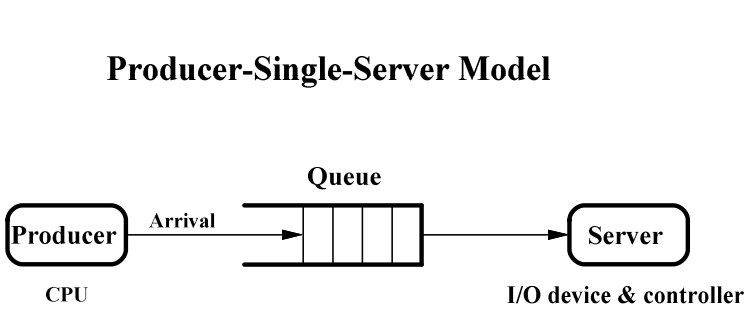
Little`s Law

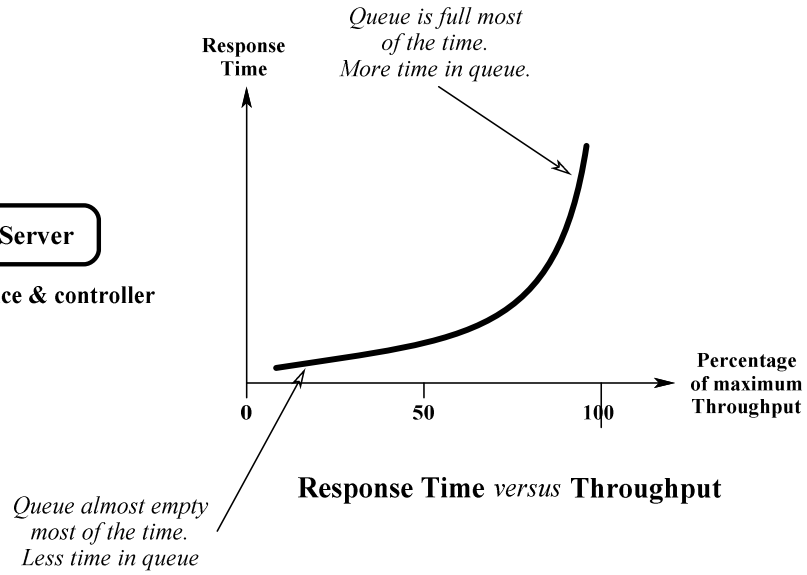
The mean number of tasks in a stable queuing system (over some time interval) is equal to the mean arrival rate (λ) of new tasks to that system, multiplied by their mean time spent in that system (T), i.e. mean response time:

$$\mathbf{N = \lambda \times T}$$

NB! Arrival rate (λ) and the **response time** (T) must use the **same time unit**.

Arrival rate (λ) is measured in *messages per second*.





Queue – the area where the tasks accumulate, waiting to be serviced.

Server – the device performing the requested service.

Throughput - the number of tasks completed by the server in unit time.

In order to get the highest throughput, the server should never be idle and the queue should never be empty.

Response time - begins when task is placed in the queue and ends when it is completed by the server.

In order to minimize the response time the queue should be empty and the server will be idle.

$\mathbf{T_{Sys}}$ – **response time** (average time/task in the system);

T_{Queue} – average time per task in the queue;
T_{Server} – average time to service a task;

$$T_{Sys} = T_{Queue} + T_{Server}$$

I/O System Architecture

There are two types of communication that are used during I/O operation with input-output devices:

1. **Control flow;**
2. **Data flow**.

A. Control flow

Control flow can be broken down into **commands** which flow from the processor to the I/O device (**outbound control flow)** and back (**inbound control flow**).

B. Hierarchical data paths

Hierarchical data paths organization divides bandwidth going down the MPS hierarchy.
Often buses are at each level of MIPS's hierarchy.

C. Bus switching methods in data paths:

I. Pended //*rippuvedastusega siin*// or circuit-switch buses

Bus is held until request is completed;
"+" simple protocol;
"–" latency of devices affects bus utilization.

II. Pipelined buses //*konveieredastusega siin*//

Multiple requests outstanding;
Fixed timing of reply;
Slave must response if it is not ready to reply.
"+" simpler to implement than packet switched;
"–" may waste bandwidth.

III. Split transaction or packet-switched //*jaos- e pakkedastusega siin*// buses

Bus is released after request is initiated;
Others can use bus until reply comes back;
"+" better bus utilization;
"–" complex bus control.

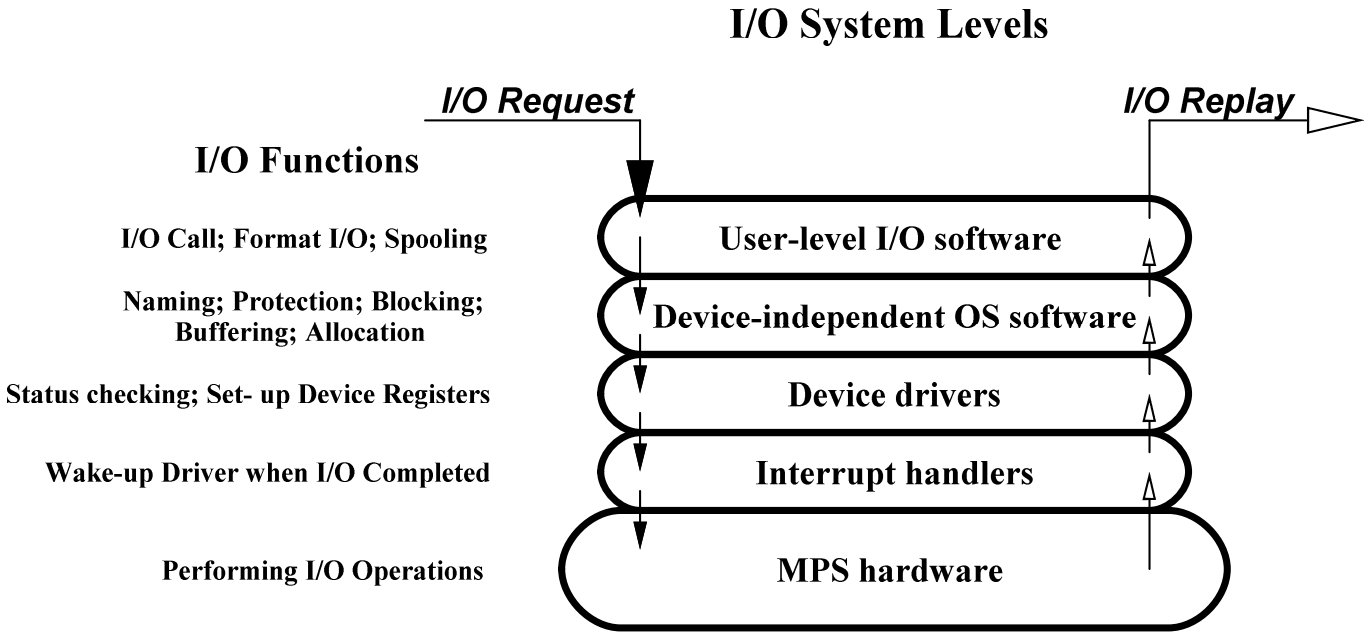
D. I/O software functions

1. **Device independence;**
2. **Buffering;**
3. **Error handling;**
4. **Synchronous and asynchronous data transfers**.

E. I/O software layers

Control layers between the user program and I/O devices hardware are:

OS kernel => I/O subsystem => Device driver => Device controller => Device



Interfaces and Buses

Interface – a physical /conceptual/ structure for channeling and facilitating communications (a boundary between systems or devices).

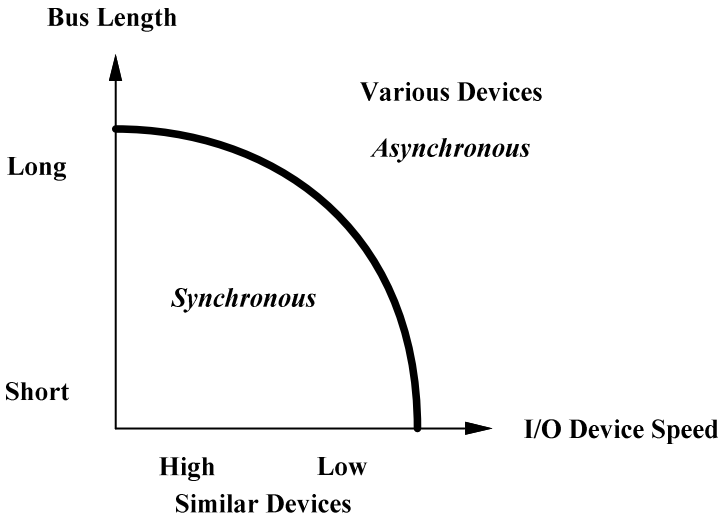
Taxonomy

1. By topology:
Magistral (Bus),
Radial (Star),
Daisy chain,
Mixed.

2. By transmission mode:
Serial,
Parallel,
Mixed.

3. By transmission direction:
Simplex,
Half-duplex,
Duplex.

4. By synchronization:
Asynchronous,
Isochronous,
Synchronous.



5. **By functionality**:
System interface,
I/O interface,
Peripheral interface,
Local and Distributed network.

Buses

Bus ⇒ a shared communication link between subsystems.

Bus System ⇒ a collection of wires and connections for data transactions among processors, memory modules and peripheral devices.

The bus is used for only one transaction at a time between a source (master) and a destination (slave). In case of multiple requests the bus arbitration logic allocates (de-allocates) the bus servicing the request one at a time. For this reason the digital bus is called contention bus or a time-sharing bus.

Bus width refers to the data and address bus widths.
System performance improves with a wider data bus adding more address lines improves addressing capacity.

Bus Types

- **System bus** or **internal bus** represents any bus within a microprocessor system. System bus connects system components together.
- **External bus** is used to interface with the devices outside a microprocessor system.
- **Processor-memory bus** connects processor and main memory (no direct I/O interface).
- **I/O bus** connects I/O devices (no direct processor-memory interface).
- **Backplane bus** – processor, memory and I/O devices are connected to same bus.
  It is an industry standard, but the processor-memory performance is compromised.

Bus System Design Goals

1. **High performance**;
2. **Standardization**;
3. **Low cost**
   Processor-memory bus emphasizes performance, then cost.
   I/O bus and backplane bus emphasize standardization.

Bus System Main Questions

1. Bus width and multiplexing (are bus lines shared or separate);
2. Clocking (is bus clocked or not: asynchronous, synchronous);
3. Switching (how and when is bus control acquired and released, atomic or split transactions);
4. Arbitration (who gets the bus next: daisy-chain, centralized or distributed).

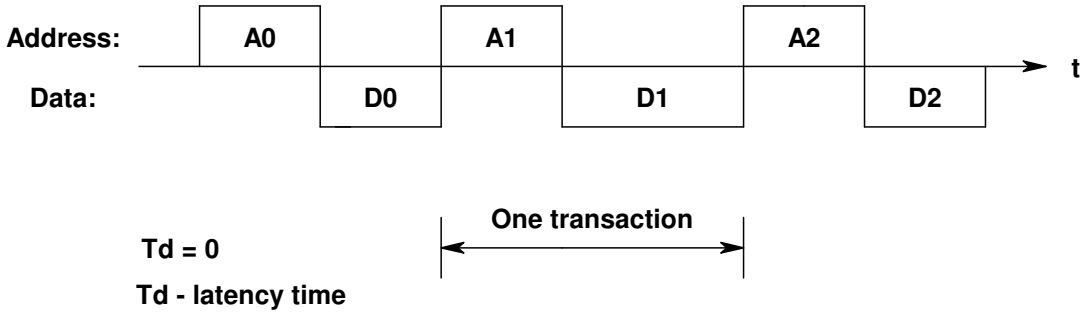
Data Transactions via a Bus-system

Simple bus

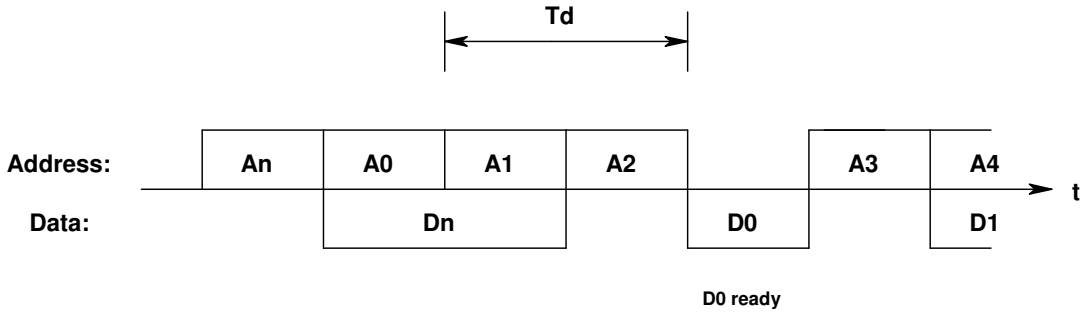
1. Only one bus transaction at a time is in progress,
2. An arbiter decides who gets the bus,
3. Unified data and address bus.

Address cycle
Master device asserts the request line it drives the address bus when the request is granted (RQ → BG).

Data cycle
The master and selected slave devices communicate for one or more cycles.



Pipelined bus

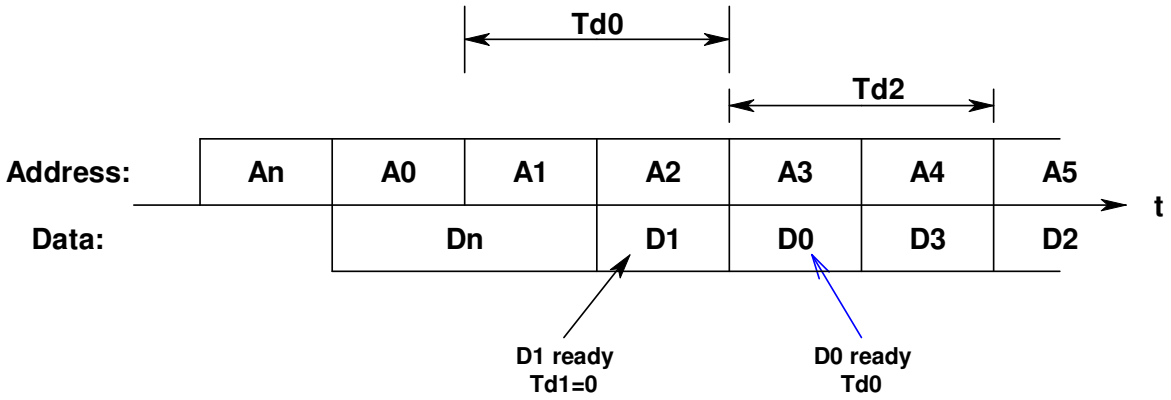


1. Increases bus bandwidth by pipelining address and data cycle,
2. Separate address and data buses,
3. New address cycle can start immediately after the previous address cycle.

Split-phase bus

1. Bus bandwidth increased by allowing **out-of-order completion of requests**,
2. Low-latency device can respond sooner,

3. Requires completely separate arbitration of address bus and data bus,
4. Devices recognize the appropriate data cycle by appropriate **bus transaction tags**,
5. Bus transaction tag size limits the number of outstanding transactions.



Bus Options

Option	High Performance	Low Cost
Bus Width	Separate Aaddress and Data lines	Multiplex Address and Data lines
Data Width	Wider is faster	Narrower is cheaper
Transfer Size	Multiple words (less bus overhead)	Single-word transfer
Protocol	Pipelined	Serial
Bus Masters	Multiple (requires bus arbitration)	Single master
Clocking	Synchronous	Asynchronous

Beyond buses ⇒ **Interconnection networks**

Bus Arbiter

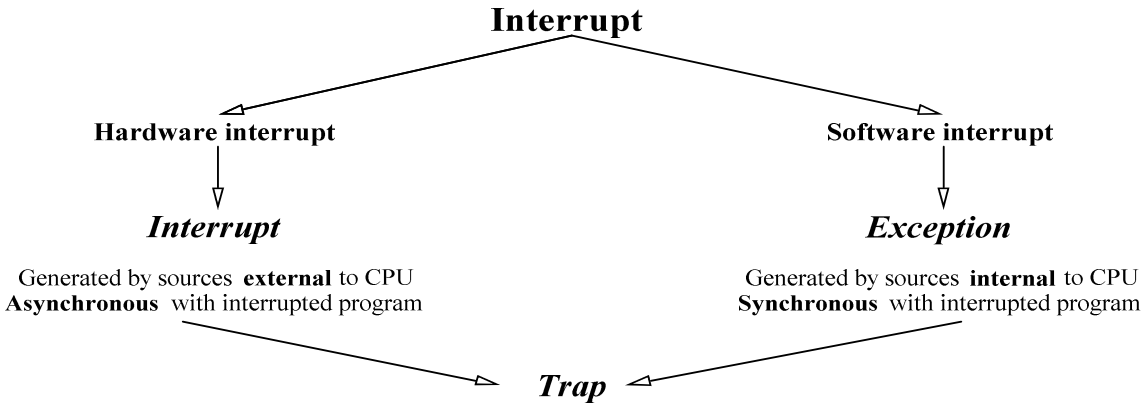
Arbitration – the process of assigning control of the data bus to a requester.
The requester is a master, and the receiving end is called a slave.
Arbiter – a functional unit that accepts bus requests (RQ) from the requester unit
and grants control of the data bus (BG) to one requester at a time.
Bus timer – measures the time each data transfer takes on the data bus and terminates
the data bus cycle if a transfer takes too long.

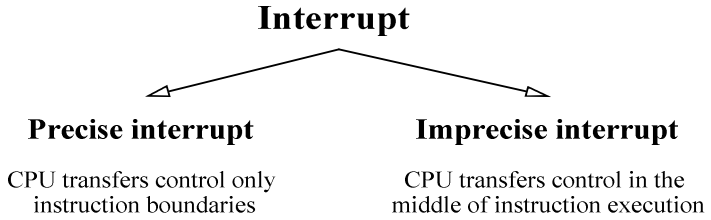
Priority System in Arbiter

a. **Fixed priorty arbiter**
b. **Variable priority arbiter**

Interrupts and Exceptions

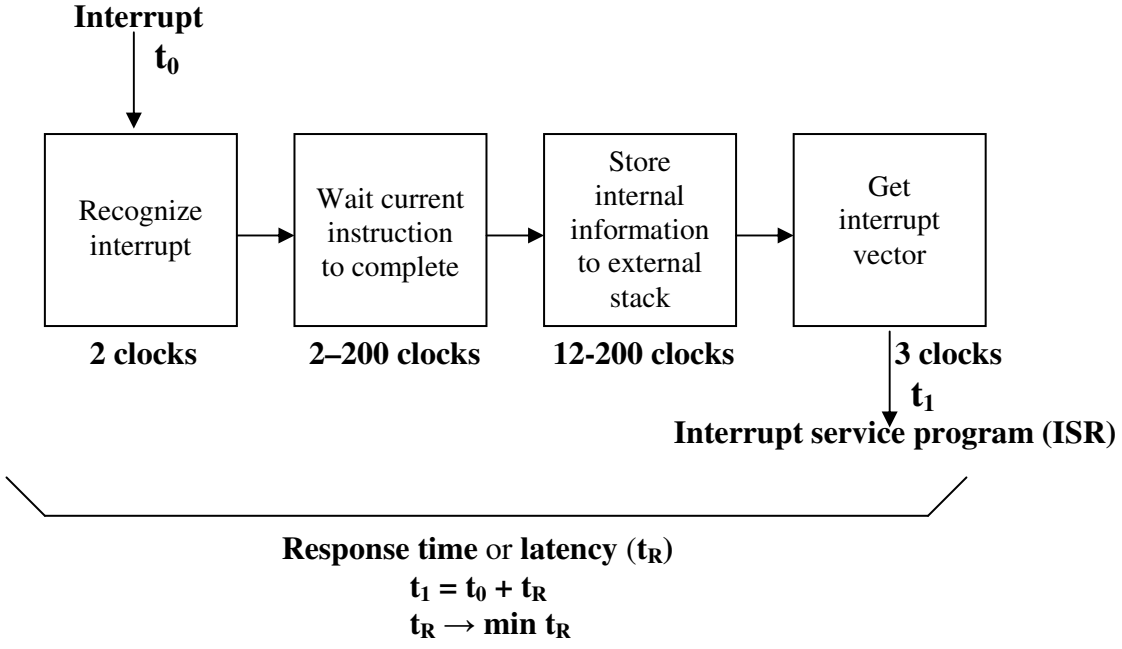
The terms interrupt //*katkestus*// and exception //*eriolek*// is used, not in a consistent fashion.





There is an operating system service routine, called interrupt handler (Interrupt Service Routine (ISR)), to process each possible interrupt. Each interrupt handler is a separate program stored in a separate part of primary memory.

Processor Interrupt Sequence



Response deadline – the maximum time that a interrupt handler (ISR) can take between when a request is issued and when the device must be serviced.

Interrupt density – the maximum number of the interrupts which can be activated at the same time.

t_i – the time taken by the interrupt service routine;
t_{pi} – the time (interval) between interrupts;
t_i/t_{pi} – determines whether a processor is fast enough to handle all interrupts well, i.e.,

($\mathbf{t_i < t_{pi}}$).

Interrupt Density Inequality

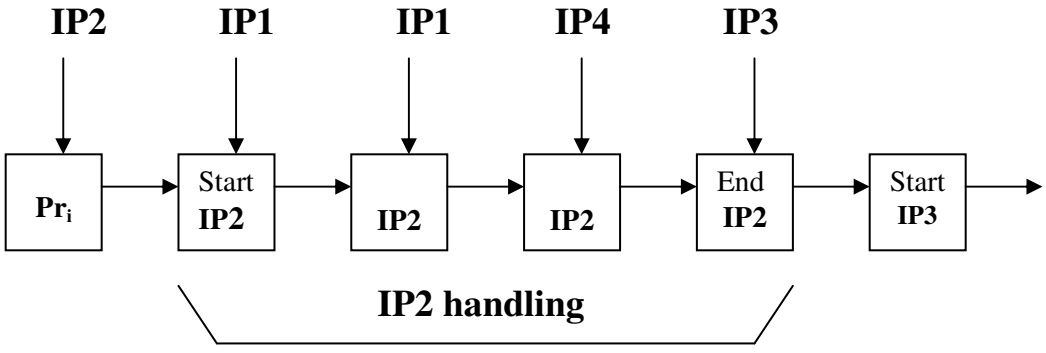
$$t_1/t_{p1} + t_2/t_{p2} + \ldots + t_i/t_{pi} < 1,$$

$$\sum_i \frac{t_i}{t_{pi}} < 1$$

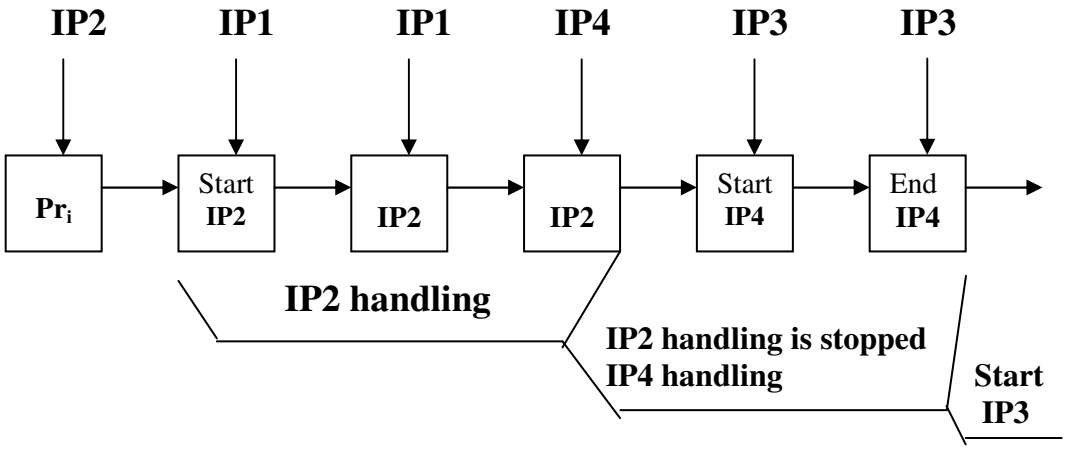
Interrupt Priority Systems

A. Relative Priority System

The priority levels **s** (s = 4) are:
IP1 (the lowest) **< IP2 < IP3 <IP4** (the highest),
Interrupted process **i** in processor **- Pr_I**



B. Absolute Priority System



C. Mixed Priority System

a. Test the **set of interrupt priority level groups – PG**

PG = {PG_1, PG_2, …, PG_k} **[absolute priorities].**

b. Test the **set of priority levels in the group PG_j**

PG_j = {$IP1_j$, $IP2_j$, …, IPs_j} **[relative priorities].**

Hardware interrupts

1. Normal interrupts (**IRQ**);
2. Non-maskable interrupts (**NMI**);
3. Fast interrupts (**FIRQ**)

Exception (Interrupt) Types

The exceptions may be user requested or coerced //*pealesunnitud*//. Coerced exceptions are caused by some hardware event that is not under the control of the user program.
Coerced exceptions are harder to implement because they are not predictable.

If the program's execution always stops after the interrupt, then it is a terminating event.
If the program's execution continues after the interrupt, it is a resuming event.

It is easier to implement exceptions that terminate execution, since the processor need not be able to restart execution of the same program after handling the exception.

Exception type	Synchronous or asynchronous	User request or coerced	Within or between instructions	Resume or terminate
I/O device request	Asynchronous	Coerced	Between	Resume
Invoke operating system	Synchronous	User request	Between	Resume
Tracing instruction execution	Synchronous	User request	Between	Resume
Breakpoint	Synchronous	User request	Between	Resume
Arithmetic overflow or underflow	Synchronous	Coerced	Within	Resume
Page fault or misaligned memory access	Synchronous	Coerced	Within	Resume
Memory protection violation	Synchronous	Coerced	Within	Resume
Hardware malfunction	Asynchronous	Coerced	Within	Terminate
Power failure	Asynchronous	Coerced	Within	Terminate

Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement.

Identifying Interrupt Source

The process of identifying the source of the interrupt and locating the service routine associated is called interrupt vectoring.

Vectoring Hardware Interrupts

Non-vectored interrupts

A fixed address is assigned in memory to the hardware interrupt request line. If there are multiple interrupt lines, a different fixed address could be assigned. Whenever interrupt occurs the CPU goes to that address and begins executing code from there.
If there are multiple devices, there could to be a generic interrupt handler that would query the devices in priority order to determine the source of interrupt.

Vectored interrupts

If the hardware devices are smart enough, then the CPU responds to each interrupt request with an interrupt acknowledge sequence.

During the acknowledge cycle the CPU indicates that it is responding to an interrupt request at a particular priority level, and then waits for the interrupting device to identify itself by placing its device number on the system bus.

Upon receiving this device number, the CPU uses it to index into an interrupt vector table in memory. The entries in the interrupt vector table are generally not the interrupt service routines themselves, but the starting address of the service routine.
The interrupted device number is offset into interrupt vector table.

Auto-vectored interrupts

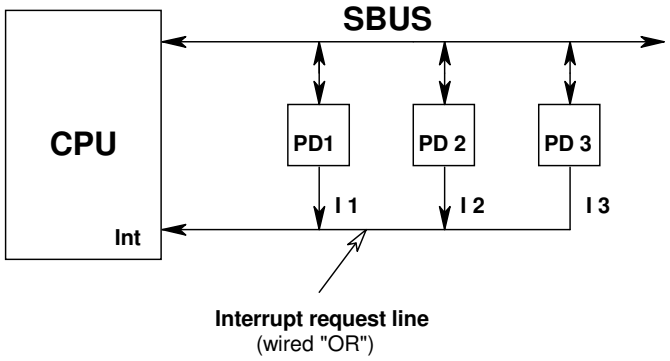
If “dumb” devices are used in the system, then a variation of the vectored interrupt scheme can be used – auto-vectoring.
In a system with auto-vectored interrupts, a device that is not capable of providing an interrupt vector number via the system bus, simply requests a given priority level interrupt, while activating another signal to trigger the auto-vectoring process.
The CPU internally generates a device number based on the interrupt priority level that was requested.

Vectored and auto-vectored interrupts can be used in the same system concurrently.

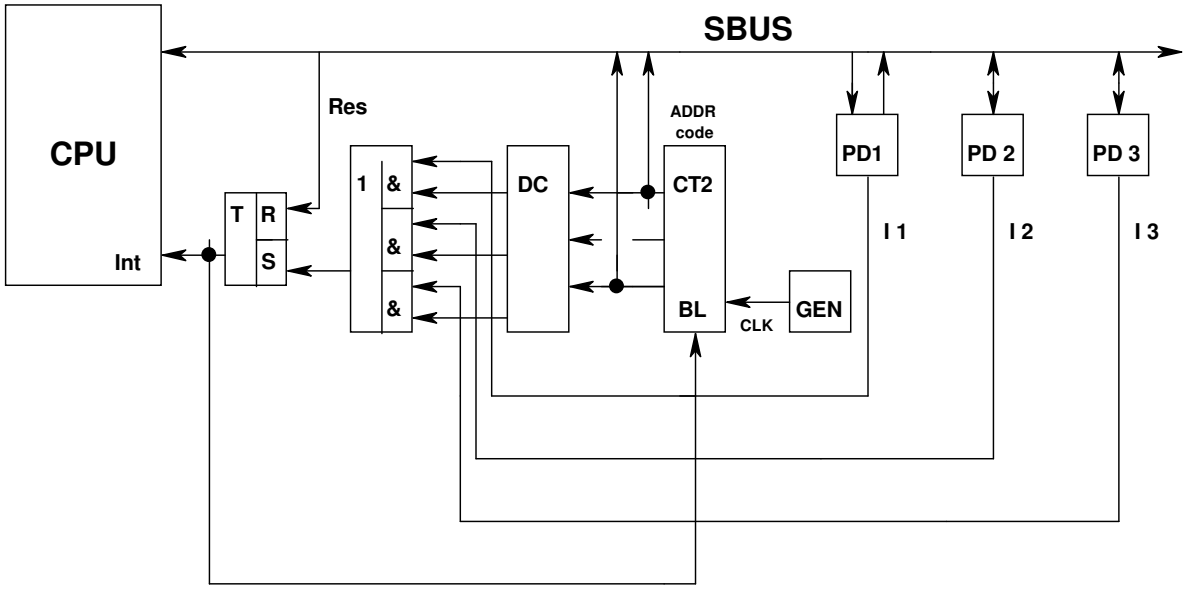
Recognition interrupt source

Serial polling

A. Software polling

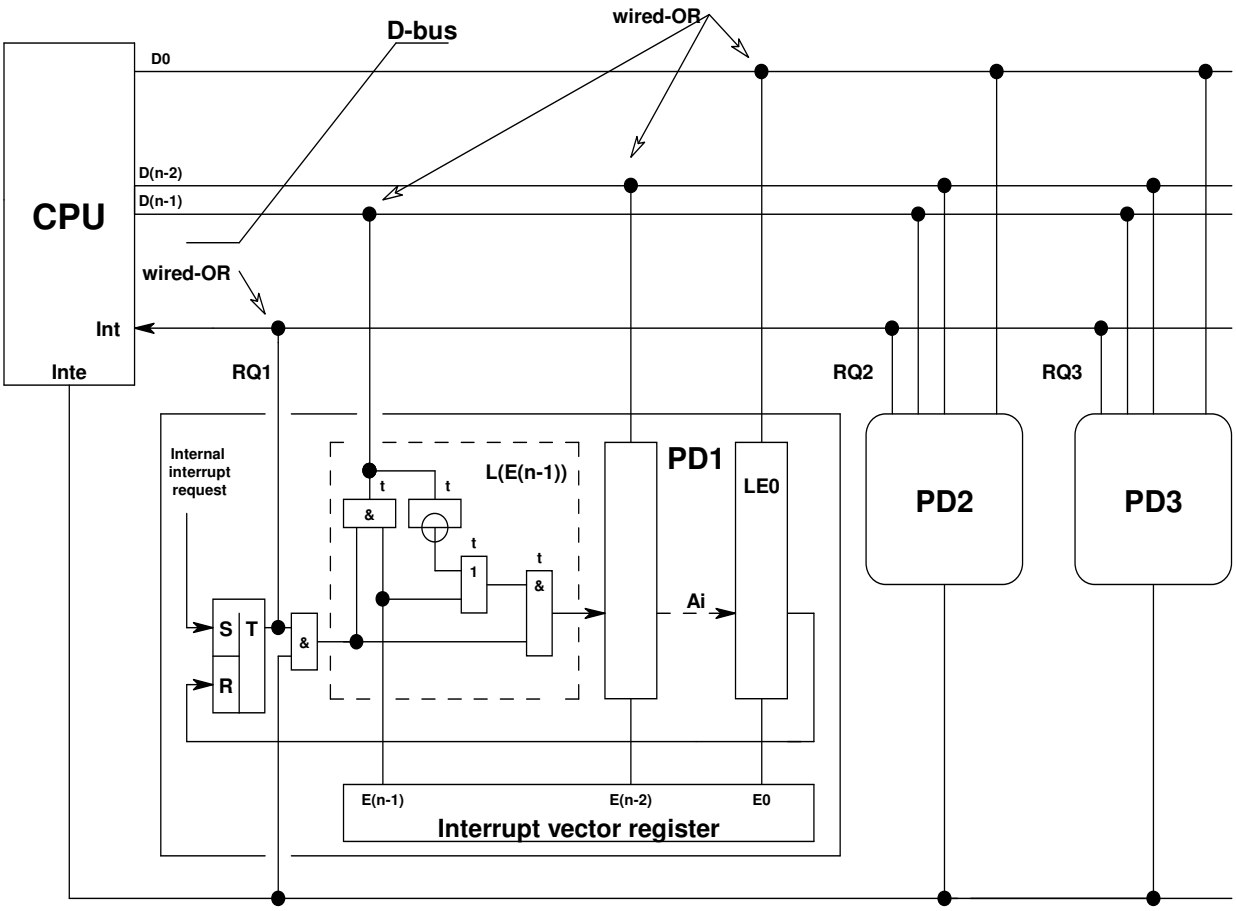


B. Hardware polling



After CPU initialization or I/O interrupt handling routine ending automatically is generated signal "Res := 1"

Parallel arbitration



Comments

Tarb = 4tn

Tarb # f(the number of PDs)

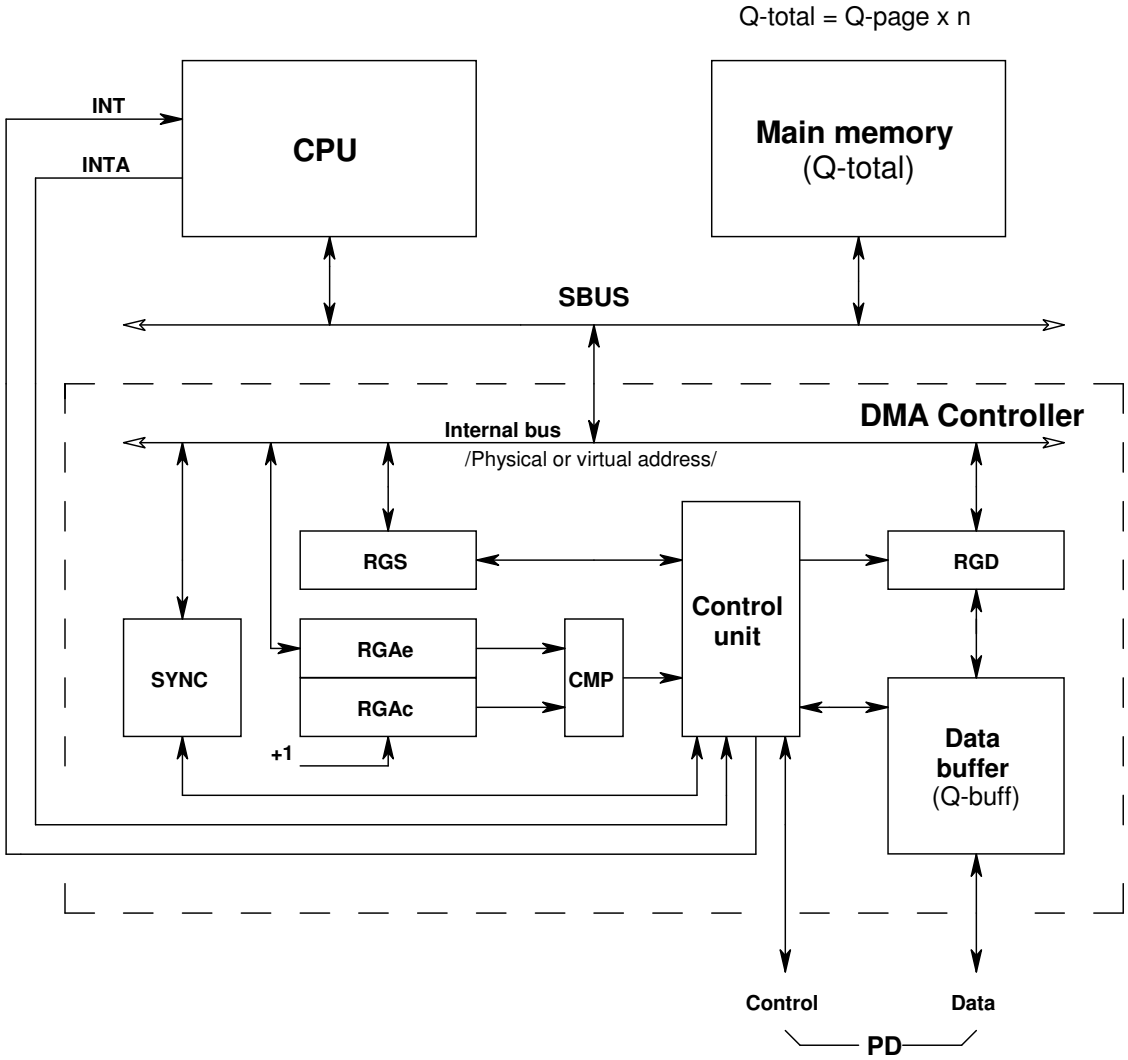
Tarb < Tclk(cpu)

Ei	Di	Ai	Activity
0	0	1	continue
1	1	1	continue
0	1	0	fail i
1	0	-	impossible !

Direct Memory Access (DMA)

DMA – a direct rapid link between a peripheral and a computer's main memory, which avoids accessing routines for each item of data read.
Link – a communications path or channel between two components or devices.

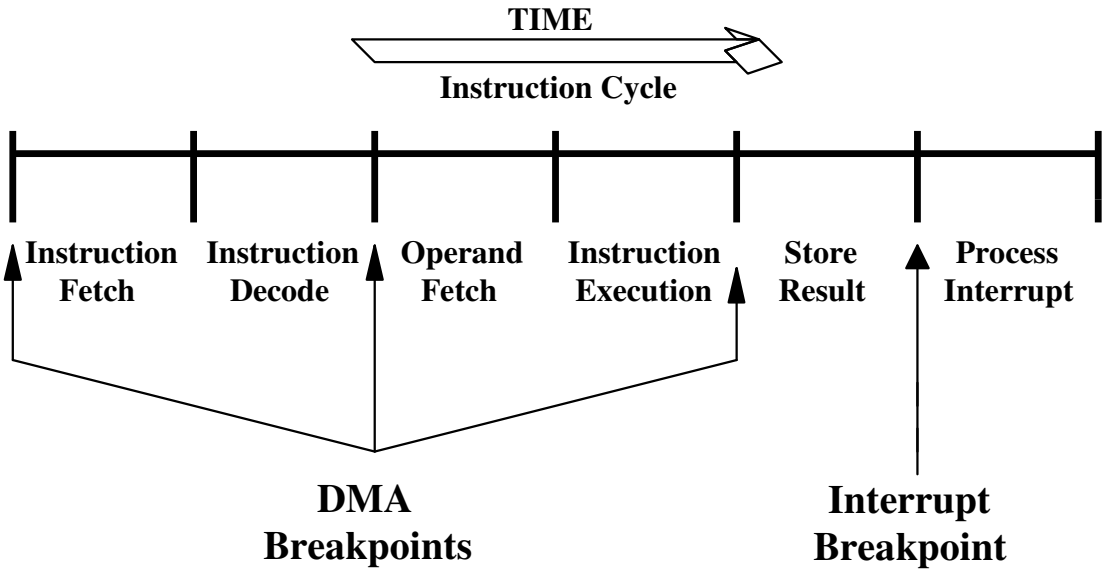
DMA Transfer Structure



DMAC – Direct Memory Access Controller

If processor whishes to read or write a block of I/O data, it issues a command to the DMA unit. After that the processor continues with other work. The DMA unit transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.
When the transfer is complete, the DMA unit sends an interrupt signal to the processor.
The processor is involved only at the beginning and end of the transfer.

The **DMA unit can interrupt the processor's instruction cycle only in certain points.**



DMA's breakpoint is not an interrupt; the processor does not save a context and do some thing else.
The processor pauses for one bus cycle.

Sometimes this requires performing atomic updates to the shared critical sections (regions).
Critical section – it is the part of the program where shared data is accessed.

Some processors also support disabling DMA operations by using locked bus cycles.
The processor could execute *lock instruction* to disable external bus grants (BG).
When critical region updates have been completed, the *unlock instruction* is used to allow bus grants.

Data Transfer Organization in DMA

The main **DMA transfer modes** are:

1. **Register mode (A);**
2. **Descriptor mode (B)**.

When the DMA runs in register mode, the DMA controller uses the values contained in the DMA channel's registers.
In the description mode, the DMA controller looks in memory for its configuration values.

A. In a **register-based DMA**, processor programs DMA control registers to initiate transfer. Register-based DMA provides DMA controller performance, because registers don't need to keep reloading from descriptors in memory. **Register-based DMA** consist of two **sub-modes**:

1. **Autobuffer mode (autobuffer DMA)**;
2. **Stop mode**.

In a**utobuffer mode,** when one transfer block completes, the control registers automatically reloaded to their original setup values and the same DMA process restarts, with zero overhead. Autobuffer DMA especially suits performance-sensitive applications.
Stop mode works identically to autobuffer DMA, except registers don't reload after DMA completes the entire block transfer. DMA transfer takes place only once.

B. The **descriptor** contains all of the same parameters normally programmed into the DMA control register set. Descriptors allow the chaining together multiple DMA sequences.
In descriptor-based DMA operations it can be programmed a DMA channel to automatically set up and start another DMA transfer after current sequence completes.
The descriptor-based mode provides the most flexibility in managing a system's DMA transfers.

Problems with DMA Channel

1. **Writing data from data buffer to main memory**.
2. The **OS removes some** of the **pages from main memory** or **relocates them**.
3. **DMA** and **caches -** a **coherence problem** for DMA.
   a. Input problem
   b. Output problem (exists only in write-back caches only)

Solutions:
- **a.** Route all DMA I/O accesses to cache;
- **b.** Disallow caching of I/O data;
- **c**. Use hardware cache coherence mechanisms. A HW at cache invalidates or updates data as DMA operation is done (*expensive!*).

Virtual DMA

The virtual DMA allows use virtual addresses that are mapped to physical addresses during the DMA operation. Virtual DMA can specify large cross-page data transfers.
The DMA controller does address translation operations internally. For these operations the DMA controller contains a small translation buffer, which content is initialized by OS, when it requires an I/O transfer.

Connection DMA Channel to the System

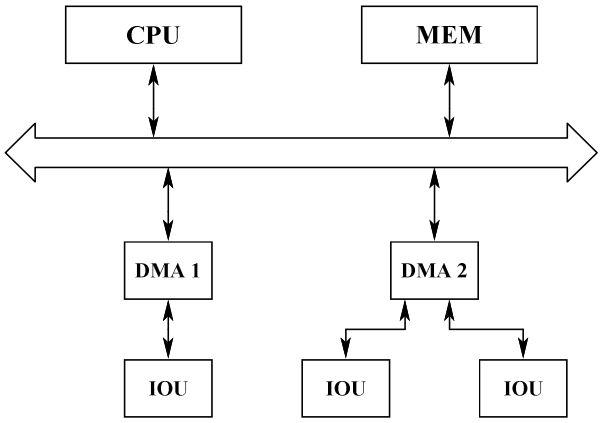
The main task is to: balance system's hardware and software. The DMA logic can be organized in a system in two ways:

1. **Each I/O device** has its **own interface with DMA** capability (fig. **A**);
2. **DMA controller can be separate** from **one or more devices** for which it performs **block transfers**. This **separate DMA controller** is called **channel** (fig. **B**).

Channel Types

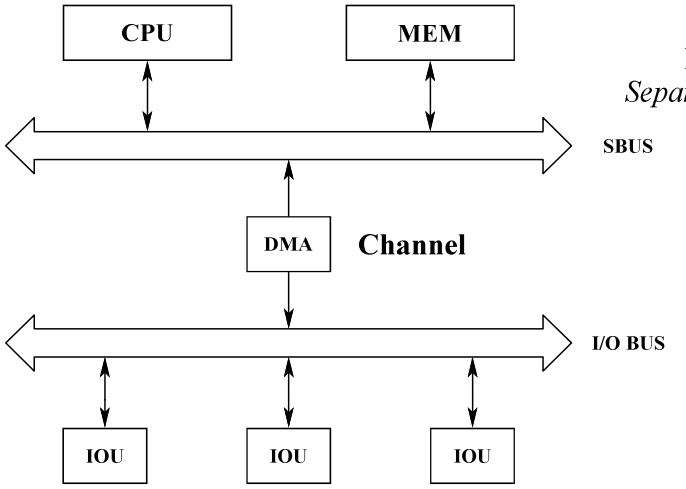
1. If a channel can do block transfers for several devices, but only one at a time, then it is a selector channel;
2. If it can control several block transfers at once, it is a multiplexer channel.
3. Channels can be made more complex and capable.
In the limit, the channel becomes a special purpose I/O processor (IOP), which can fetch operations from memory and execute them in sequence.

A.



Single Bus System
Integrated DMA and I/O Units

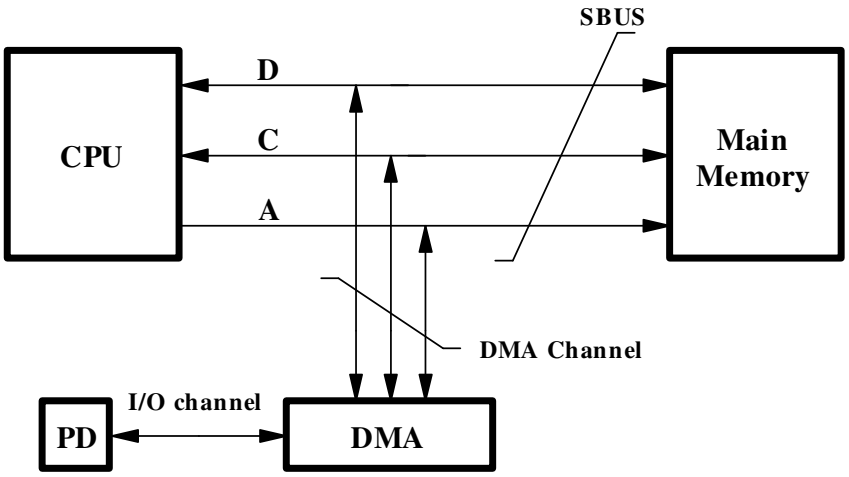
B.



Double Bus System
Separated DMA and I/O Units

DMA-channel Connection Schemes

1. Traditional connection, **one united data channel**.

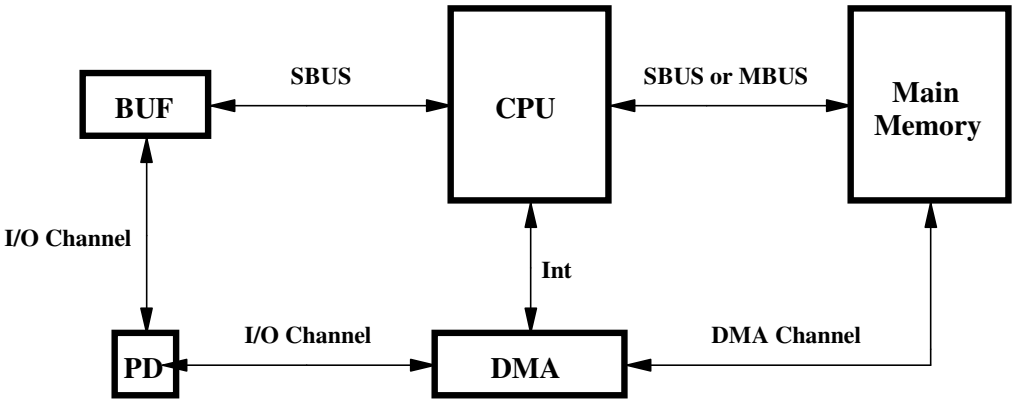


System-mapped DMA Channel

Connection variations:

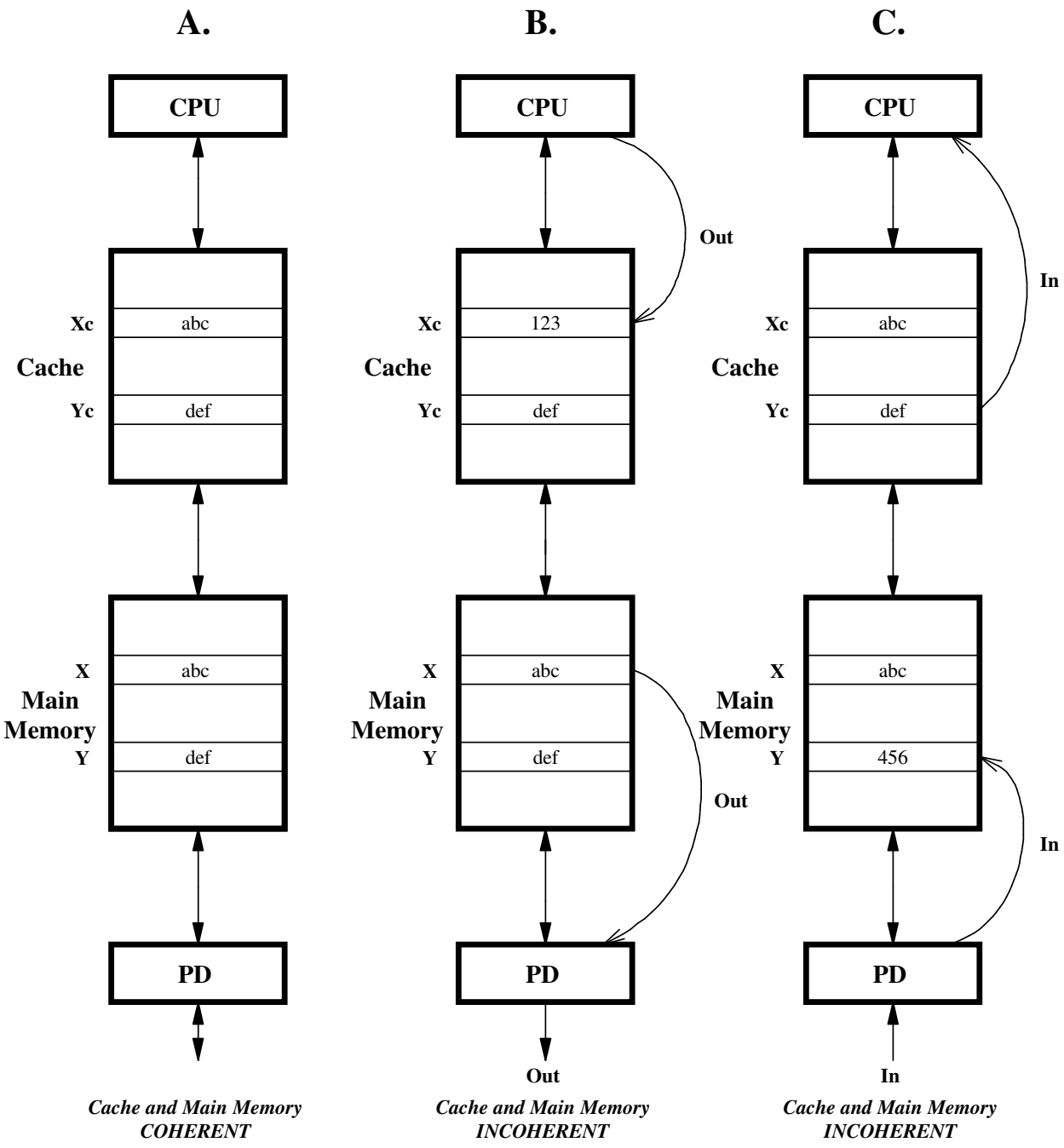
1. **Burst mode transfer**,
2. **Shared mode transfer** (***cycle stealing***)

2. **Two independent data channels** (typical in the high-performance MP systems).



Separate DMA and Main Memory Channels

Input-output Operations and Cached Data Consistency



The goal is to prevent the stale-data problem.

INPUT-OUTPUT SYSTEM CONTROLLERS (PROCESSORS)

I/O Controllers Types

1. **Transparent** (harmonizing electrical parameters);
2. **Unifying timing** (harmonizing timing parameters)
3. **Unifying transfer protocols**;
4. **Data handling** (all previous features + temporal data buffering).

I/O Controller Main Functions

1. Establishing a communication between I/O controller and memory;
2. Establishing connection between **I/O interface** and I/O controller;
3. Data exchange with I/O device;
4. Notifying the CPU of completion of I/O operation.

I/O Interface Main Functions

1. Recognition addresses;
2. Resolving the characteristics differences.

Evolution Stages

1. **I/O circuits** on SSICs or MSICs (non-unified)*;

2. **Special ICs** for controlling I/O operations
   (TIM, SIO; PIO, DMA, etc.);

3. **Special** (dedicated) purpose **I/O controllers**
   (display controller, FDD controller, HDD controller);

4. **Universal programmable I/O controllers**
   Simple processing unit and PIO on chip
   For low-speed I/O devices,
   Master/Slave (M/S)-system capabilities.
   (*Intel 8041* (1979) was the first programmed I/O device)

5. **I/O processors** (**IOP**) or **peripheral processing unit** (**PPU**)

I/O processors are also called peripheral processors or sometimes *front-end processors* (FEP)
(graphics PR - *82786*, DMA PR - *82259*, I/O RISC-PR - *i960*)

IOP Main Functions:

1. Performing a group operations;
2. Handling of data to be input/output;
3. Data editing, debugging, validating, correction.

Data transfer b**etween** the **IOP** and the **central processor** can be organized:

1. Through the disk system;
2. Through the shared memory.

In **disk coupled system** the IOP stores data on the disk unit, which in turn are processed by the central processor.

IOP Main Features:

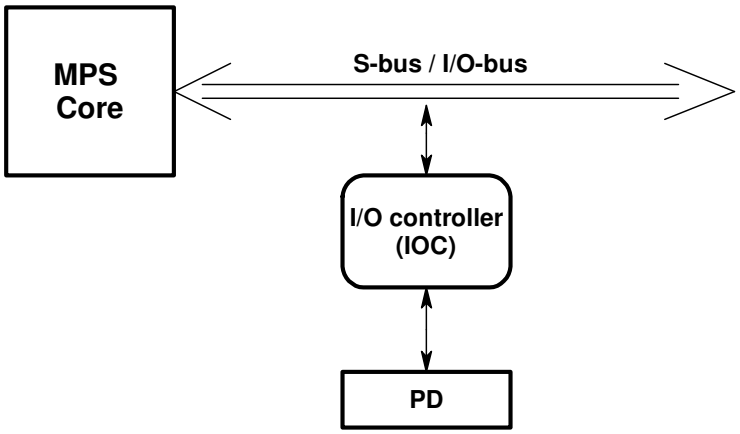
1. Internal DMA channel(s);
2. Advanced Master/Slave-system capabilities.

6. Transputers

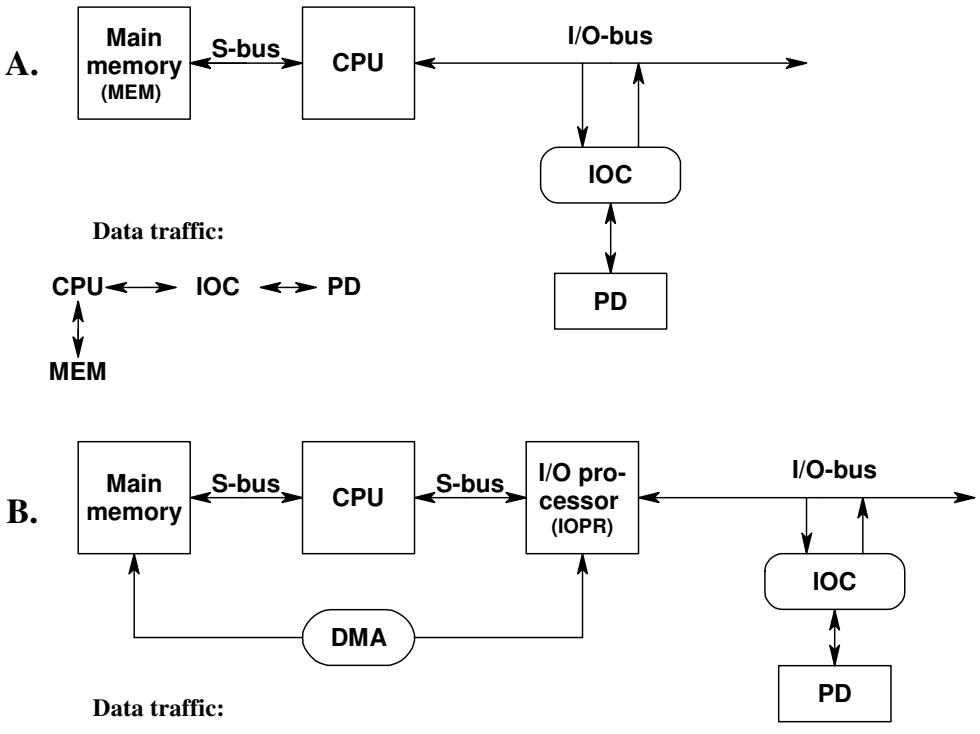
High-performance pipelined parallel I/O and network processors.
(IMS *T222, T800, T9000*).

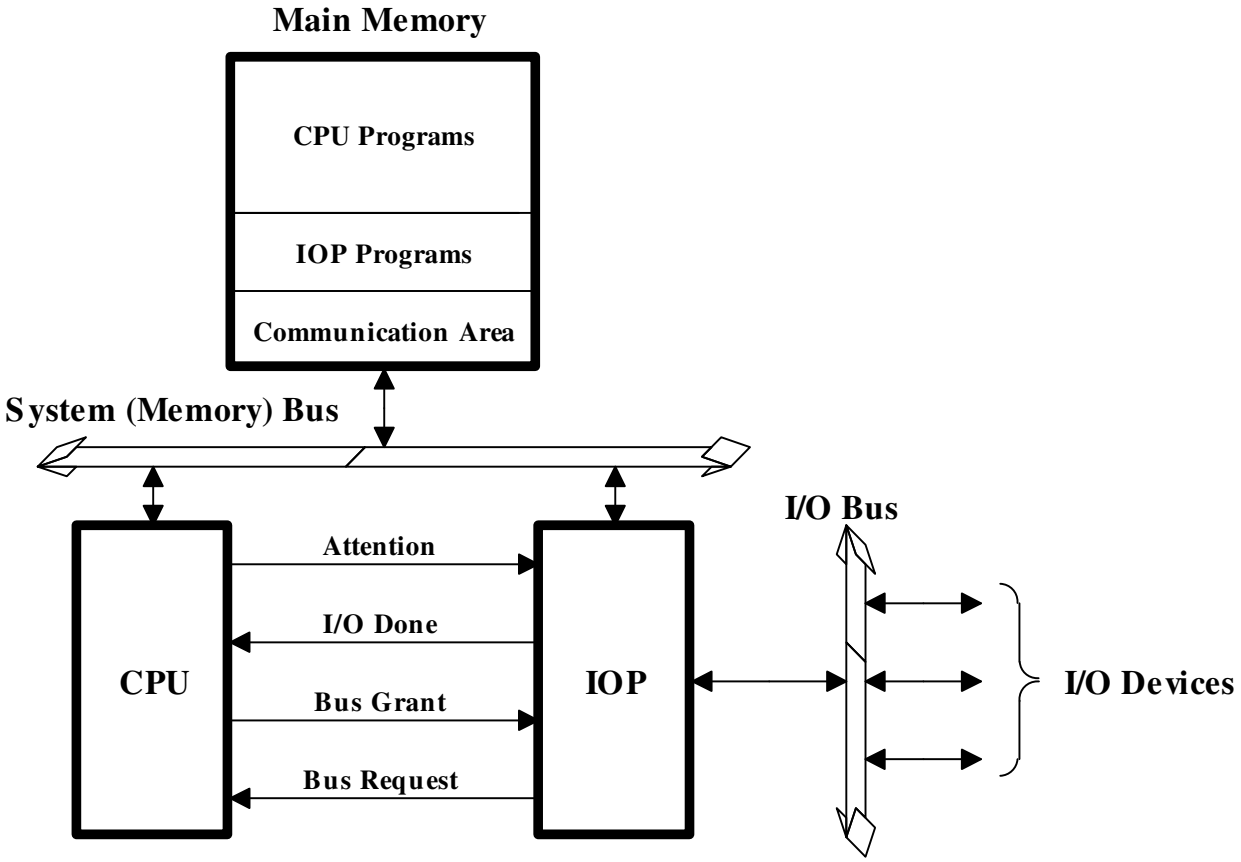
I/O Controller

Controller - a module or a specific device which operates automatically to regulate a controlled variable or system.



I/O Controller Connection to the CPU





CPU-IOP Interface Model

The communication area in main memory is used for passing information between CPU and IOP in the form of messages.
Typically CPU has three I/O-oriented instructions (each of which will be a CW) to handle the IOP:

TEST I/O*, *START I/O*, *STOP I/O.

The instruction set of an IOP consists of data transfer instructions (*READ*, *WRITE*), address manipulation instructions and IOP program control instructions.

Coprocessors

Accelerator – is a separate architectural substructure that is architected using a different set of objectives than the base processor, where these objectives are derived from the needs of a special class of applications.

The accelerator is tuned to provide higher performance at lower cost, or at lower power consumption.

Coprocessor – a **secondary processor** that is used to speedup operations by taking over a specific part of the main processor's work.

The First Generation Coprocessors

CISC micros with loose-coupled FP-processors
(*80286* + *80287*; *80386* + *80387*; *68020* + *68881*; *68030* + *68882*)

1. CPU fetch instruction and data,
2. Fetched information is sent to the coprocessor,
3. Microcoded FP operations consume several tens of instruction cycles.

The Second Generation Coprocessors

RISC micros with tight-coupled FPUs.
(MIPS *R2000*; *R3000*; *R6000*; *SPARC*, HP-*PA RISC*)

1. Separate CPU and FPU chips (the lack of die space),
2. Off-chip SRAMs are used as caches,
3. FPU is hardwired,
4. The CPU and FPU are watching the instructions (on the data bus) in parallel and each does their own part of the work separately.

The Third Generation Coprocessors

RISC or post-RISC micros with integrated (CPU + FPU + MMU + Cache) on one chip.
(*Pentium*; *68040*; *68060*; *Super SPARC*, *M10000*; *Alpha*)

1. Tight-coupled coprocessors watch the instruction stream,
2. Multiple-issue instructions (heavily pipelined coprocessor chip).

Graphics Processor**

Graphics – the creation and manipulation of picture images in the computer.

Typical Graphics Operations:

1. Pix-bit operations;
2. Transferring pixels linear addresses into (x-y) – coordinates;
3. Clipping;
4. Masking bit planes;
5. Special graphic functions, as for:

   - **Rendering polygons**;
     Rendering is the process of producing the pixels of an image from

higher-level description of its components.

- **Translating vertices** into different coordinate systems;
- **Geometric calculations**;
- **Texture mapping**

  Texel ⇒ texture pixel.
- **Programmable shadowing** to determine the final surface properties of an object or image.

Universal Microprocessors Drawbacks:

1. Universal CPU executes graphics operations very slowly;
2. Universal CPU has limited number of internal registers
3. Universal CPU's standard interfaces are not fitted for graphics data transmission.

Graphics Controller (GC)

Features:

1. Bit-bit operations;
2. Graphics manipulations;
3. Graphics controller (**GCNT**) chip integrates **display control** and **graphics processing units**;
4. Information transferring path:

CPU < = > GCNT < = > Display Unit

Drawbacks:

1. Fixed number of graphics instructions;
2. CPU is burdened;
3. Many external IC chips.

Graphics Processor (GP)

Graphics processor (**GP**) or **graphics engine** is a **secondary processor** (**coprocessor**) used to speed up the display of graphics.

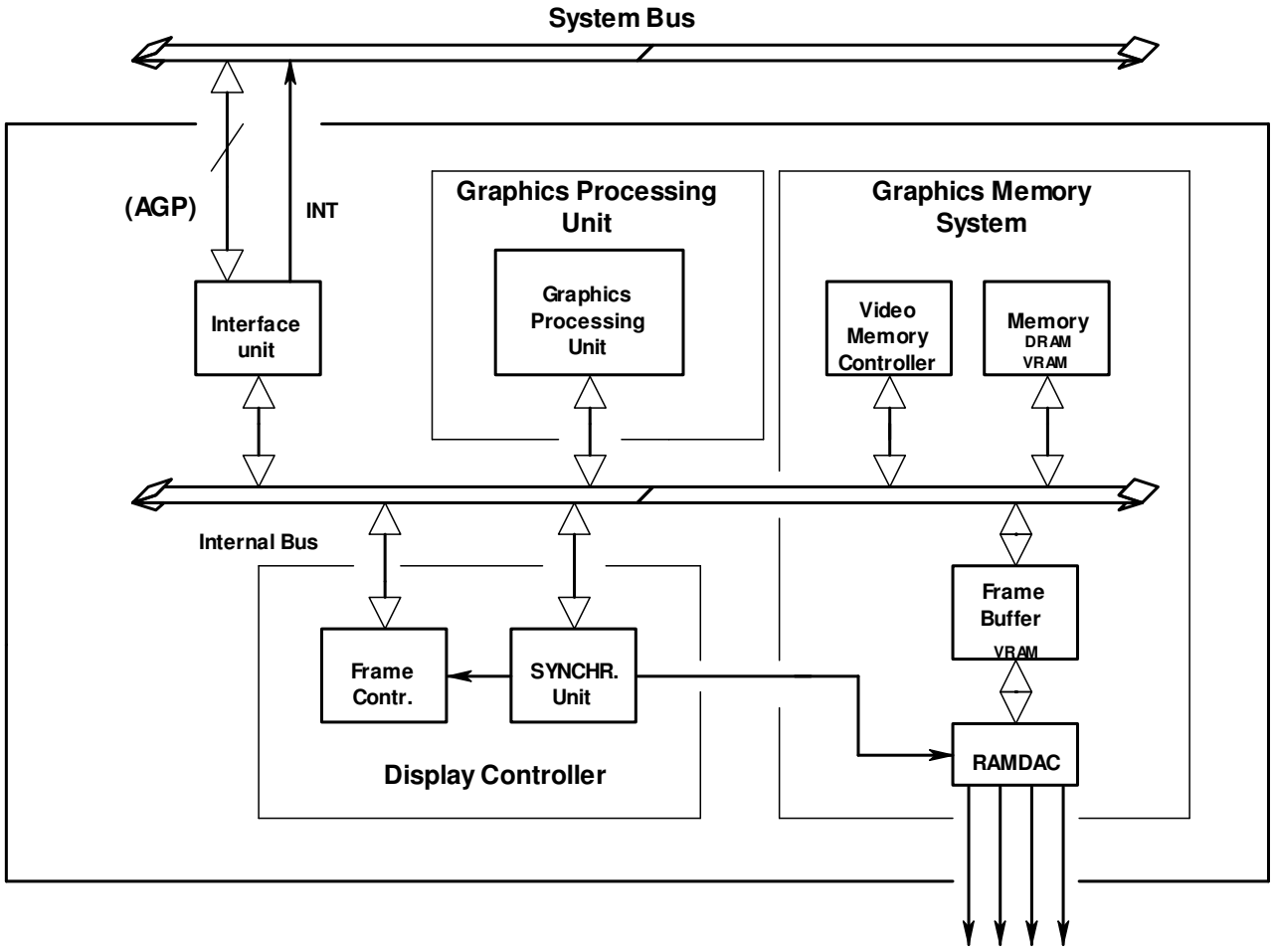
Features:

1. Pix-bit operations;
2. Universal software;
3. GP is the universal device;
4. Universal information transferring path:

CPU <···> GP < = > Display Unit

Specific features:

a. Microprogrammed control;
b. Different text displaying modes;
c. Window overlapping;
d. Generating high quality moving objects;
e. Effective resolution is **1024×2048** or more pixels;



Graphics Sub-system's Model

Graphic Processor Units (**GPU**) have become an essential part of every computer system available today.
GPU has been evolving faster than the CPU, at least doubling performance every six months.
GPUs are stream processors that can operate in parallel by running a single kernel on many records in a stream at once.

Memory Bandwidth and Computational Performance between the CPU and GPU

	CPU	GPU
Memory Bandwidth	**6,4** GB/s	**35,2** GB/s *
Peak Computational Performance	**6** GFLOPS **	**48** GFLOPS ***

* Comparable to the CPU L2 cache bandwidth
** 3,2 GHz *Pentium 4* (theoretically)
*** GeForce *FX6800* (equivalent to a 24 GHz *Pentium 4*)

3D Images Processing in GPU

GPUs can implement many parallel algorithms directly using graphics hardware.
Most real-time graphics systems assume that everything is made of triangles, any more complex shapes, as for curved surface patches are formed from triangles.
A 3D application uses the CPU to generate geometryto send to the GPU for processing, as a collection of vertices.

Vertex is a point in space defined by the three coordinates **x**, **y** and **z**..

A vertex description consists of attributes that define its position in 3D space (usually relatively).

Pixel shader program allow graphics engine to process spectacular effects.
There are two forms of shaders – vertex shader and pixel shader.

Vertex processor (vertex shader) affects only vertexes, which are less relevant to overall performance than the pixel shader.
The vertex shader program processes and alters the attributes of the vertex, on a vertex-by-vertex basis, before they passed to the next step in the rendering process, by the vertex processing hardware.
The vertex shader is used to transform the attributes of vertices such as color, texture, position and direction from the original color space to the display space.
Vertex processor allows the original objects to be reshaped in any manner.
The output of a vertex shader, along with texture maps, goes to an interpolation stage.

Pixel processor (pixel shader) is devoted exclusively to pixel shader programs. Pixel shaders do calculations regarding pixels. They are used for all sorts of graphical effects.

As vertex geometry and pixel shader code structures are functionally similar, but have dedicated rolls, and then it is possible to create the unified shaders.

Texture mapping units (TMU) work in conjunction with vertex and pixel shaders. TMUs job is to apply texture operations to pixels.

The process of rasterization takes the geometry processed by the vertex processor and converts it into screen pixels to be processed by the pixel shader //*pikslivarjusti*// or pixel fragment hardware (fragment processor.
The rasterization is the conversation of geometry into screen pixel fragments, generated by walking over the geometry lists and analyzing them to see where they lie on the screen.
Each pixel can be treated independently from all other pixels.
The actual color of each pixel can be taken directly from the lighting calculations, but for added realism, images called textures are often draped over the geometry to give the illusion of detail.
GPUs store these textures in high-speed memory (texture buffer).
Rasterization units (ROP) are responsible for writing pixel data to memory.

Processed pixel fragments are stored in frame buffer. The content of frame buffer is converted into binary representation and directed to the monitor unit.
The Frame Buffer Controller (FBCNT) interfaces to the physical memory used to hold the actual pixel values displayed on the display unit screen.
The Frame Buffer Memory is often used to store graphics commands, textures and other attributes associated with each pixel.

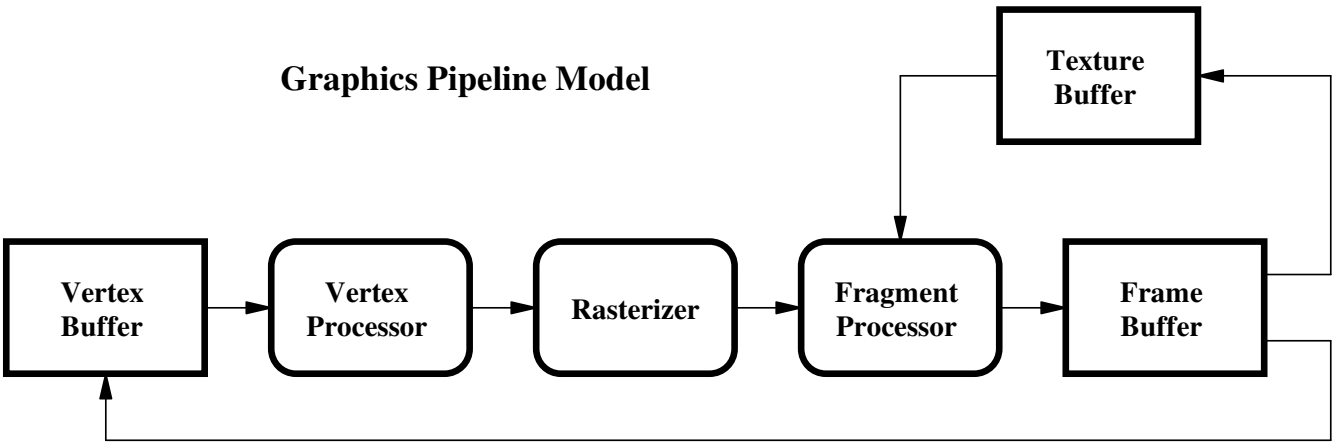
Graphics Pipeline

The GPUs uses a hardwired implementation of the graphics pipeline. Within a graphic processor, all pipeline stages are working in parallel.

Pipeline is a **term** used to **describe graphics engine architecture**.

There are different pipelines within a graphics processor. Today the term “pipeline” does not longer describe accurately the newer graphics processor architecture. The newer graphics processors have a fragmented structure – pixel processors are no longer attached to single TMUs.

Graphics Pipeline Model



Whereas CPUs are optimized for low latency, **GPUs are optimized for high throughput**.

GPUs are characterized by:

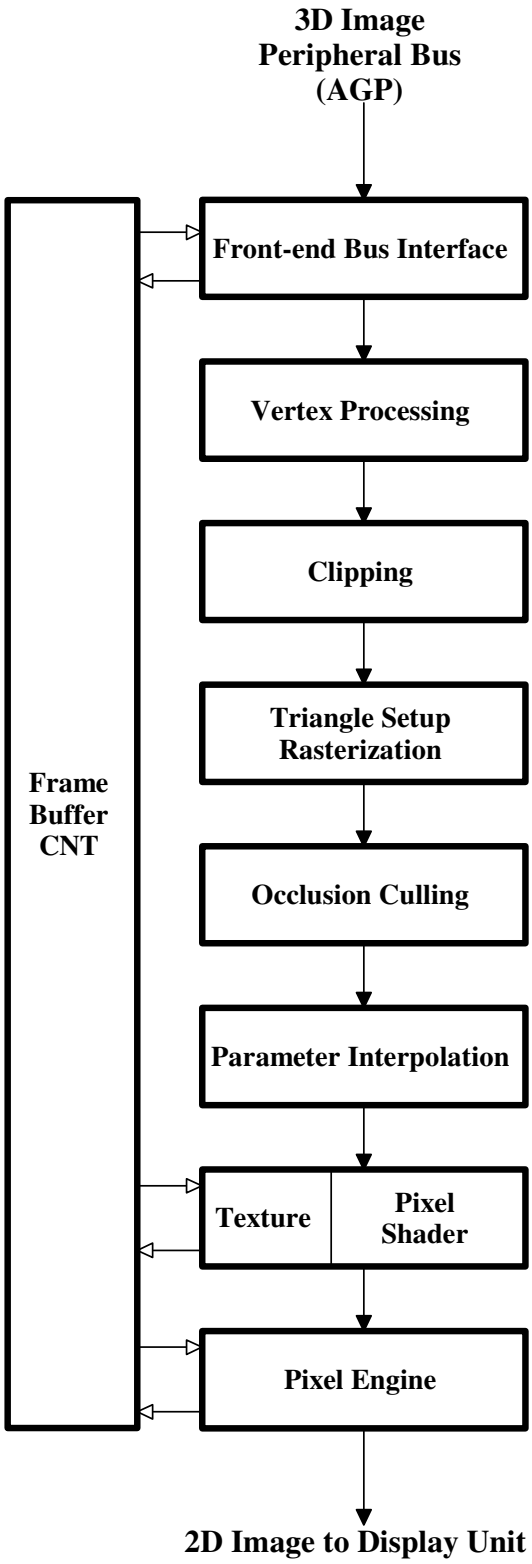
1. GPUs are highly parallel;
2. GPUs have more than 24 processors;
3. GPUs are highly threaded;

Example

There are **various stages,** which are all work in parallel**,** in the **typical pipeline of a GPU:**

1. **Bus Interface** (**front-end**);
2. **Vertex Processing;**
3. **Clipping;**
4. **Triangle Setup and Rasterization;**
5. **Occlusion Culling;**
6. **Parameter Interpolation;**
7. **Pixel Shader and Texturing;**
8. **Pixel Engine;**
9. **Frame Buffer Controller** (**FBCNT**).

Graphics Pipeline



Process commands

Transform vertices to screen 2D space position. A programmable vertex shader enables the application to perform custom transformations for effects such as wrapping or deformation of a shape.

Delete unseen pixels in the 2D screen view

Verteces are collected and converted into triangles. The triangles are filled with pixels known as fragments.

Delete hidden (occuluded) pixels

The values for each pixel that were rasterized are computed, based on color, texture, etc.

Stage adds textures and final colors to the fragments. A fragment shadder enbles the application to combine a piel`s attributes (color, depth and position on screen) with textures in a user-defined way.

Mathematically combines the final fragment color, its coverage and degree of transparency with existing data stored at the associated 2D location in the frame buffer to producefinal color for the pixel to be stored at that position. Output in a depth value (Z) for the pixel.

Transputer**

TRANS**istor + com**PUTER** ⇒ **TRANSPUTER

Transputers Main Features

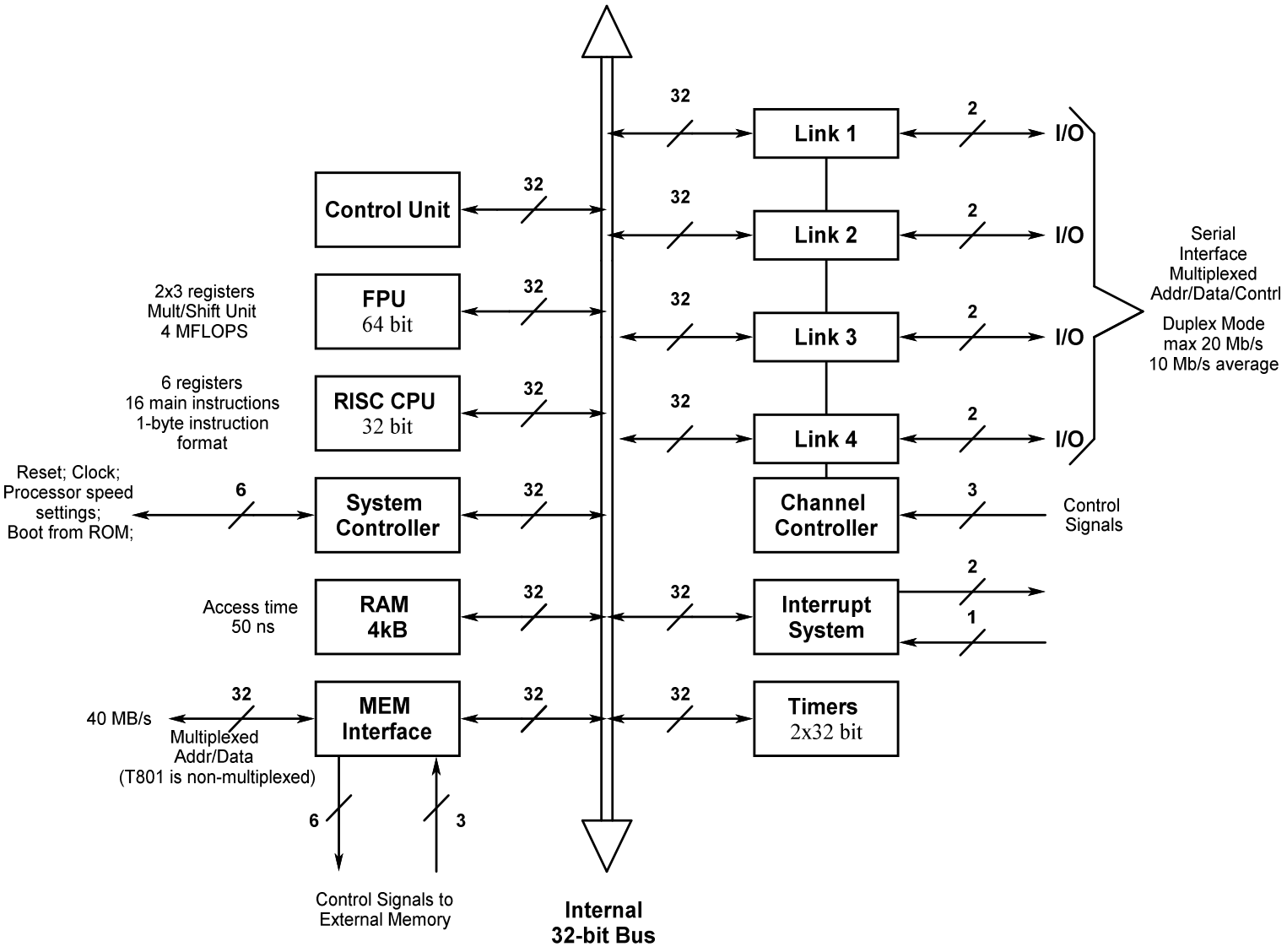
1. The Transputer architecture was introduced by INMOS in 1985. It is single chip microcomputer architecture, optimized for parallel use in MIMD configurations.
2. As controller, the Transputer include CPU, memory and I/O in one package.
3. The Transputer has a high degree of functional integration. External support requirements are minimal.
4. The Transputer's on–chip RAM memory may be allocated for cache, data or instructions. On-chip RAM provides single cycle access, while external memory is a minimal of three cycle access.
5. A Transputer external memory interface is von Neumann, using a 32-bit wide address and data paths. No virtual memory features is included. The memory addresses space basis on linear address space.
6. Neither memory management unit nor special cache is provided in the Transputer architecture.
7. The Transputer is a stack machine. This provides for very fast task context switch for interrupt and task switching.
8. A Transputer has a number of simple operating system functions built into the hardware. A microcoded scheduler maintains time sharing between processes running on the hardware).
9. Instructions are decoded and issued to Transputers on-chip FP coprocessor (FPU) by hardware in CPU. Calculations of the operand addresses and loading the operands into the FPU are done by hardware in the CPU.
10. The Transputer features high speed interconnects by means of full duplex asynchronous serial communications. The four serial links (20 Mbit/s) support bidirectional asynchronous point-to-point concurrent communications.
11. The Transputer has two 32-bit timers. The high priority process timer is incremented every microsecond. The low priority timer is incremented every 64 microseconds. Timers are used for process scheduling.
12. The Transputer can boot from a ROM or from one of the serial links.
13. No JTAG support is included.
14. All Transputer's instructions are 1-byte in size. The subset of 31 single byte instructions is used about 80% of the time. An on-chip instructions queue handles 4 byte-sized instruction fetched simultaneously from memory over the 32-bit bus.
15. The instruction format is a 4-bit operation code, followed by a 4-bit data value.
16. Instruction OPR (Operate-general way to extend instruction set) is used to define additional 16 secondary zero-operand instructions.
17. The CPU of the Transputer has three registers, organized as a stack. Similarly, the FPU has a three register stack.
18. The CPU includes a workspace pointer to local variable memory, an instruction pointer (program counter), and an operand register.

19. The Occam language (a trade mark of the INMOS Group Companies) implements a simple syntax for parallel process, and for communication between processors. The architecture of the Transputer is defined by reference to Occam. Multiple processes can be implemented on one Transputer, or on multiple Transputers, the communication method at the software level is the same.
20. The Occam language was developed to program concurrent, distributed systems.
21. Occam enables a system to be described as a collection of concurrent processes which communicate with each other and with peripheral devices through channels.
22. Concurrent processes do not communicate via shared variables, and the Occam is a suitable language for programming systems where there is no memory shared between processors in the system.
23. Concurrent programs can be expressed with channels, inputs, and outputs combined in parallel and alternative constructs.
24. Each Occam channel provides a communication path between two concurrent processes. Communication is synchronized and takes place when both the inputting and outputting processes are ready.
25. Data to be output is then copied from the outputting process to the inputting process, and both processes continue. Data is sent over one of the two wires forming the link.
A transmitted byte is framed by service bits.
26. The Transputer does have an assembly language, but its use discouraged.
27. A process can be active or inactive:

- **a.** An active process can be executing or awaiting execution;
- **b.** An inactive process may be waiting for I/O resources (ready to input, ready to output), or waiting for a particular time to execute.

28. Interrupts or exceptions are termed events in the Transputer.
29. The most powerful Transputer was/is *T9000* (1991) [200 MIPS or 25 MFLOPS/50 MHz].
T9000`s hardware supports the virtual link mechanism. It makes possible to use one physical link to conduct exchanges between any numbers of process pairs taking place in different Transputers.
30. The Transputers are used for image processing, pattern recognition, artificial intelligence, systems simulation, supercomputers, and Transputer networks.

The modern nearest equivalent to the Transputer like technology is the *HyperTransport* processor (2001) interconnection fabric designed by AMD.



TRANSPUTER T800

1987

15 MIPS (f_{clock} = 30 MHz)

Literature

V. Korneev, K. Kiselev. Modern Microprocessors. Third edition. Charles River Media, Inc., Hingham Massachussets, 2004 [e-book].

MICROPROCESSOR SYSTEM'S PERFORMANCE

The CPU time is the time the CPU is computing, not including the time waiting for I/O or running other programs. The CPU time can be divided into the CPU time spent in the program (user CPU time), and the CPU time spent in the operating system performing tasks requested by the program (system CPU time).
The CPU performance refers to user CPU time, while the system performance refers to elapsed time on an unloaded system. System's performance is limited by the slowest part of the path between CPU and I/O devices. The system's performance can be limited by the speed of:

1. The CPU,
2. The cache memory system,
3. The main memory and I/O bus,
4. The I/O controller (I/O channel),
5. The I/O device,
6. The speed of the I/O software,
7. The efficiency of the used software.

If the system is not balanced, the high performance of some components may be lost due to the low performance of one link in the chain!

CPU Performance

The **CPU performance** is determined by several **factors** as for:

1. Instruction cache miss cycles - $\mathbf{D_{IC}}$,
2. Data cache miss cycles - $\mathbf{D_{DC}}$,
3. Bad branch prediction penalty cycles - $\mathbf{D_{BR}}$,
4. Scheduling unit stall cycles - $\mathbf{D_{SU}}$,
5. Loss of potential instructions from fetch inefficiency - $\mathbf{L_{FH}}$
   It is mainly caused by instruction cache misses and branch instruction disturbings in the control flow.
   In multiple-issue processors the instruction buffer is not filled or partly filled with instructions.

The instructions per cycle (IPC) is related to the size of the issued instruction block (S_{IB}) and utilization percentage (UT) //*rakendatuse määr*// by:

$$\mathbf{IPC=S_{IB} \times UT,} \text{ where}$$
$$\mathbf{UT=100\% - (\%D_{IC}+\%D_{DC}+\%D_{BR}+\%D_{SU}+\%L_{FH})}$$

All percentages are relative to the total number of cycles. These percentages change dynamically, but should not vary widely.

The **average UT** is approximately **64%** (where $\mathbf{D_{IC}}$**~2,1%**, $\mathbf{D_{DC}}$**~1,8%**, $\mathbf{D_{BR}}$**~8,6%**, $\mathbf{D_{SU}}$**~5,3%**, and $\mathbf{L_{FH}}$**~18,6%**).

> The **CPU's performance** is not the same as the **SYSTEM'S PERFORMANCE!**

Overall System Performance

$$\mathbf{T_{exe} = t_{clk} \times IC \times CPI + T_{Mem} + T_{I/O},}$$ where

$\mathbf{T_{exe}}$ – the total execution time

Improving System Performance

1. Improve the clock rate ($\mathbf{t_{clk}} \downarrow$):
   - Faster technology
   - Pipelining
2. Reducing the total number of instructions executed (**IC** ↓).
3. Increase the parallelism (**CPI** ↓ or **IPC** ↑):
   - Superscalar architecture
   - VLW architecture
   - Multiple processors
   - Speculative execution
   - Out-of-order execution

But there are the **bottlenecks** – the **memory delay** ($\mathbf{T_{Mem}}$) and **input-output system delay** ($\mathbf{T_{I/O}}$).

Overall System Speedup

The overall system speedup (S_{sys}) due to enhancements in CPU speed (S_{CPU}), memory speed (S_{Mem}) and I/O speed ($S_{I/O}$) can be expressed as:

$$S_{sys} = \frac{1}{\frac{f_{CPU+Mem}}{S_{CPU+Mem}} + \frac{f_{I/O}}{S_{I/O}}}, \text{ where}$$

$\mathbf{f_{CPU+Mem}}$ – fraction of total execution time during which an instruction is executing in the CPU and the main memory, including the time for any overhead in these subsystems;

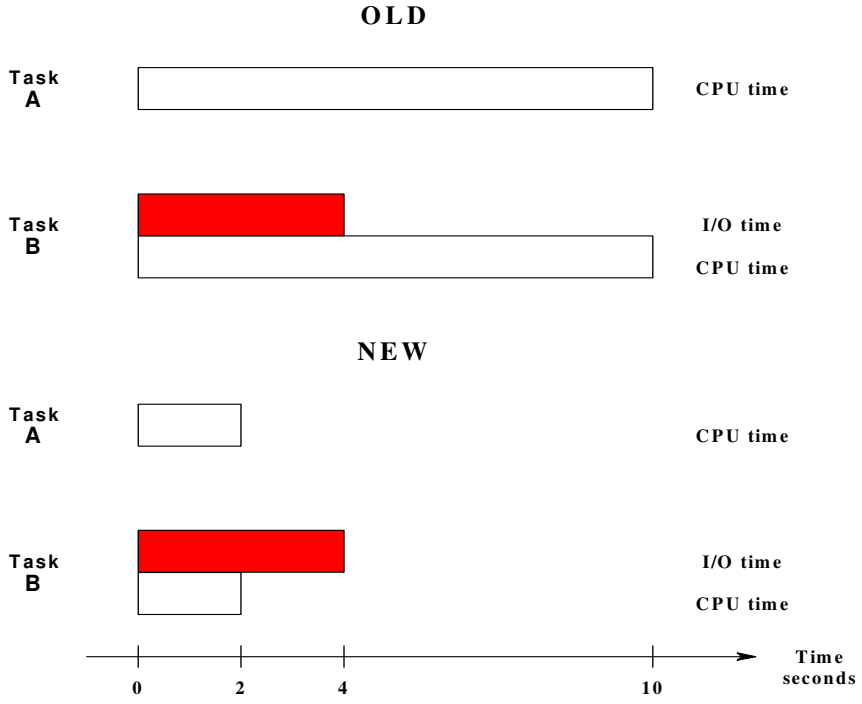
$\mathbf{f_{I/O}}$ – fraction of the total execution time during which I/O takes place, including the time for I/O overhead ($\mathbf{f_{I/O} = 1 - f_{CPU+Mem}}$)[3].

Overlapping

Overlapping is the phenomenon of **concurrent processing.**
In a microprocessor system there can be overlap between the *CPU* and *I/O unit* activities.
Within the **CPU** there can be overlap between instructions *fetching* and *execution.*

There are two tasks **A** and **B**. The both tasks take **10** seconds to run. Task A needs very little I/O, so it is not mentioning. Task B keeps I/O devices busy for **4** seconds; this time is completely_overlapped with CPU activities. The old CPU is replaced by a newer model with **5×** the performance.

Replacement causes the following effect:



The NEW system is not BALANCED!

[3] It is assumed, that there are non-overlapping (CPU+Mem) and (I/O) systems.

System balance is the ability of the system to maximize processor productivity.
To maximize processor performance, the exchange of data must be fast enough to prevent the compute processors from sitting idle, waiting for data.

The faster the processors, the greater the bandwidth required and the lower the latency that can be tolerated.

***Balanced systems take into account the needs of the applications and matches memory, I/O and interconnect performance with computing power* to maximize *the processors utilization*.**

The elapsed execution time of a task T_t, is:

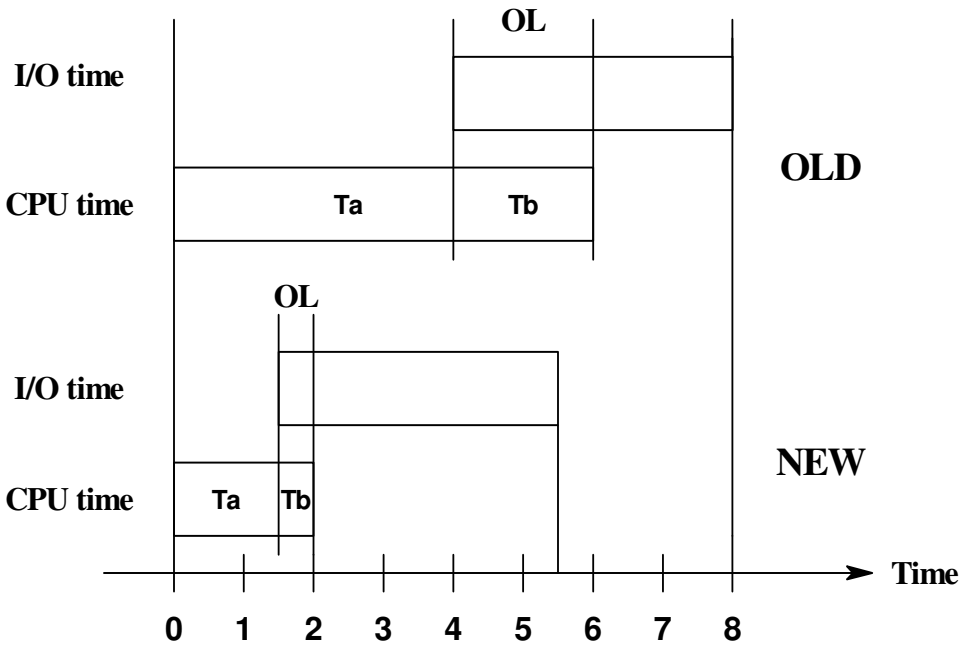
$$T_t = T_{CPU}+T_{IO}-T_{OL}, \quad \text{where}$$

T_{CPU} - CPU is busy,
T_{IO} - I/O system is busy,
T_{OL} - overlap time.

Balanced System

A. Speeding up the CPU

OLD: T_{CPU}=6sec, T_{IO}=4sec, T_{OL}=2sec, T_a/T_b=4/2=2, $\mathbf{T_t=8sec}$.
NEW: $\mathbf{S_{CPU}=3}$.
OL – Overlapping in the CPU and I/O subsystem activities



$$T_t^{CPU} = T_{CPU}/S_{CPU} + T_{IO} - T_{OL}/S_{CPU}, \quad \text{where}$$

S_{CPU} - speed up CPU.

$$T_{CPU} = T_a + T_b,$$

$$(T_a/T_b)^{OLD} = (T_a/T_b)^{NEW}$$

T_t^{CPU} = 6/3 + 4 - 2/3 = **5,34 sec**. (T_{OL} is now 2/3s, where T_a=1,34sec and T_b=0,66sec)

B. Speeding up the I/O

$$T_t^{IO} = T_{CPU} + T_{IO}/S_{IO} - T_{OL}/S_{IO}, \quad \text{where}$$

S_{IO} - speed up I/O.

Example

The task takes 60 seconds to run. The CPU is busy 40 seconds and the I/O system is busy 30 seconds. How much time will the task take if the CPU is replaced with one that has two times the performance?

T_t^o=60 sec
T_{CPU}=40 sec
T_{IO}=30 sec
S_{CPU}=2
T_t^n=?

$$T_t = T_{CPU}/S_{CPU} + T_{IO} - T_{OL}/S_{CPU}$$

$$T_{OL} = |T_t - (T_{CPU} + T_{IO})|$$

$$T_{OL} = \left| 60-(40+30) \right| = 10 \text{ sec}$$

$$T_t^n = 40/2+30-10/2 = 45 \text{ sec}$$

T_t^o=60s, T_t^n = 45 sec,

C. Speeding up the CPU and the I/O Subsystems Concurrently

$$T_t^S = T_{CPU}/S_{CPU} + T_{IO}/S_{IO} - T_{OL}/(\max(S_{CPU},S_{IO}))$$

The overlap period is reduced by the largest value of the speed up, i.e. **$\max(S_{CPU},S_{IO})$.**

At a system level bandwidths and capacities should be in balance. Each functional unit in the MPS is capable of demanding bandwidth and supplying bandwidth.

There is a relationship between storage capacity and bandwidth requirement:

Amdahl/Case Rule for a Balanced System

1 MIPS <=> 1 MByte Memory <=> 1 Mbits/s I/O

If corrected to **1 Mbyte/sec** of **I/O**, then the rule is applicable for modern systems.

INSTRUCTION FETCH AND EXECUTION PRINCIPLES

Von Neumann conceived a program as comprising two orthogonal sets of operations that worked in conjunction:

1. A dataflow which did the physical manipulation of the data values,
2. A control flow, which dynamically determined the sequence in which these manipulations were done.

Principle of orthogonally specifies that each instruction should perform a unique task without duplicating or overlapping the functionality of other instructions.

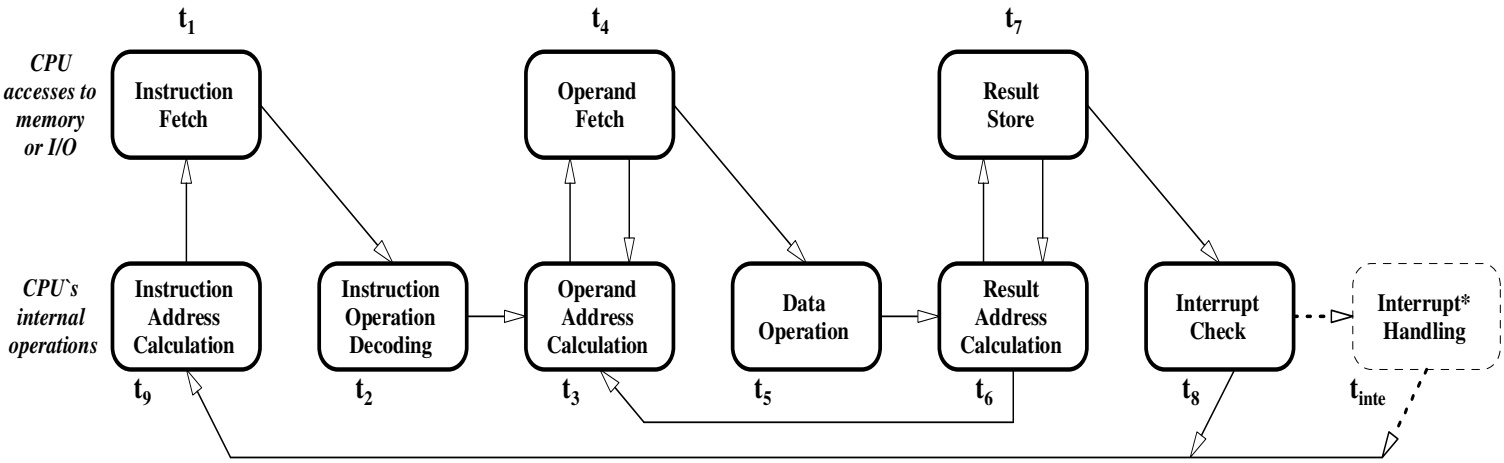
Instruction — a coded program step that tells the processor what to do for a single operation in a program.

Instruction cycle - the process of fetching and executing an instruction.
Instruction cycle time (T_{ICY}).
Instruction types:

1. **Data movement** instructions;
2. **Data processing** instructions;
3. **Branch** instructions;
4. **Environmental** instructions.

Instruction set – the **collection of instructions** that a **CPU can process**.

Instruction Cycle State Diagram (with interrupts*)



* Interrupt ⇒ interrupt or exception

$T_{ICY} = \varphi(t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9)$,

$T_{ICY} = var$.

max P (performance) **=> min** T_{ICY}

Example

Categories of Instruction Operators

Operator type	Example
Arithmetic/logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression or decompression operations (**Vertex** is a point in 3D space with particular location, usually given in terms of x, y, and z coordinates.)

Instruction Execution in Processor

1. **Single-cycle**
   - Each instruction executes in a single clock cycle.
2. **Multi-cycle**
   - Each instruction is broken up into a series of short steps.
3. **Pipelined**
   - Each instruction is broken up into a series of steps.
   - Multiple instructions execute at once.

Degree of Parallelism and Specialization

The degree of parallelism of a processor is the number of operations it can execute concurrently. There is a duality between pipelining and parallelism.

A pipelined n stages deep functional unit has the same degree of parallelism as n functional units, since both of them need n independent operations to use the resources fully. To deliver the same performance, a pipelined processor must run at n times the clock speed of the parallel processor.

Beside the degree of parallelism, it is also important the degree of specialization of the functional units available.

The more specialized the resources are, the harder it is to use them all efficiently.

Specialization can explain why the same degree of parallelism is harder to use in a pipelined processor than in a non-pipelined processor.

While an n-stage pipelined unit has the same degree of parallelism as n parallel units, then the hardware for each pipeline stage is specialized, whereas each of the n parallel units is capable of performing the entire function.

A. If a processor has a higher degree of parallelism; it needs a large number of independent operations to keep its resources fully utilized.
B. If it also has a higher degree of specialization, not all resources may be fully occupied even if there are many independent operations. Processor is slowed down by its busiest type of resource.

Instruction-level parallelism (ILP) refers to degree to which (on average) the instructions of a program can be executed in parallel.

> Instruction-level parallelism exists when instructions in a (program) sequence are independent and thus can be executed in parallel by overlapping.

Machine parallelism is a measure of the ability of the processor to take advantage of instruction-level parallelism.

Machine parallelism is determined by the number of instructions that can be fetched and executed at the same time and by the speed that the processor uses to find independent instructions.

Instruction Fetching

The instruction fetch unit is an important resource. Increasing the width of the instruction fetch and decode unit does not necessarily translate to a similar increase in the effective instruction bandwidth. This is due to low dynamic run lengths in programs.

Dynamic run length is the number of consecutive instructions executed up to and including the next taken branch.

Since in each cycle the processor only fetches the continuous instructions in parallel, the effective fetch bandwidth is limited by the dynamic run lengths in the execution.

Front-end //*preprotsessor*// – the mechanism(s) responsible for supplying instructions to the execution units.
Back-end //*postprotsessor*// – the execution units.

The **front-end** includes the fetch unit, the decode unit, the rename unit and the support structures.

The aim of a high-performance front-end is to keep the later stages of the processing pipeline busy by providing them with a sufficient number of instructions every cycle.
To improve fetch throughput, the mechanism must fetch a large number of instructions that are not consecutive in the static program representation. This can be accomplished in several ways:

1. Rearranging the static code so that basic blocks are consecutive in the static program. The rearrangement may be done statically or dynamically.
2. Using hardware that can read multiple cache lines simultaneously.
3. Observing the dynamic execution order as the program executes and caching instructions in their dynamic execution order.

Control Flow and Handling Branches

The control flow of a program is implemented by branch instructions. The branches are the essence of computation; their frequency (static or dynamic) is dominant among all instructions.
For typical applications, branches comprise between one-third and one-fifth of all instructions. The branches are crucial to performance.

The main problem during instruction fetch cycle is control transfer formed by branch, call, jump, return instructions or by interrupts.

A. There is a problem with target instruction addresses that are not aligned (misaligned) to the cache line addresses.
Misalignement occurs when a jump address is in the middle of a cache block. If a fetch group extends beyond the end of cache block, then another cache block must be read.
If we have a self-aligned instruction cache //*isejoondav käsuvahemälu*//, then it reads and concatenates two consecutive lines within one cycle to be able to always return the full fetch bandwidth.

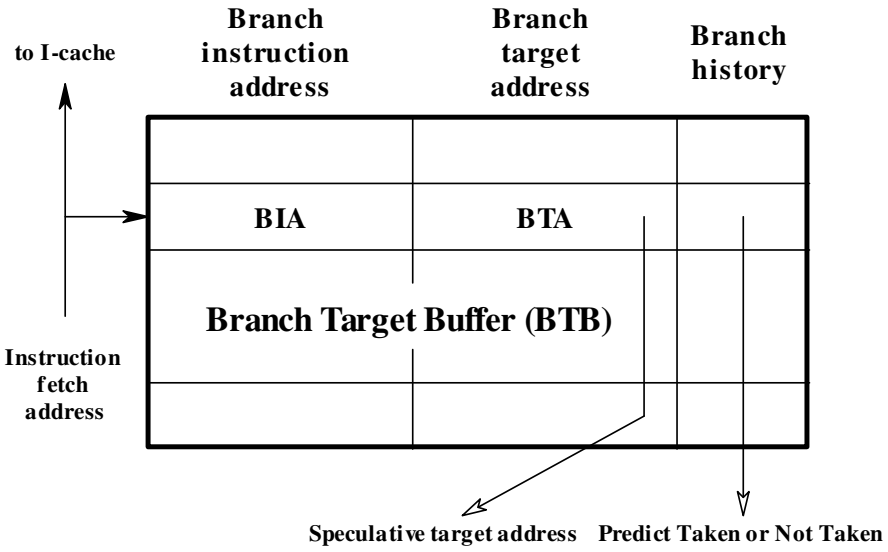
C. One method, which can improve instruction fetch performance, is instructions prefetching. The method requires that instructions can be fetched and stored in an Instruction Queue before they are needed. The fetch unit must have hardware which recognizes branch instructions and computes the branch target address.
The technique of prefetching of instructions and executing them during a pipeline stall due to instruction dependency is called branch folding.

D. Sometime is used instruction fetching prediction, which helps to determine the next instructions to be fetched. Usually this method is applied in conjunction with branch prediction. The method predicts the next instruction to fetch after fetching a conditional branch instruction.

An **efficient branch handling technique** must guarantee:

a. An early determination of the branch outcome (branch resolution);
b. Buffering of the branch target address //*hargnemise siirdeaadress*//, after its first calculation,

in a branch target buffer (BTB) or in a branch target address cache (BTAC) and an immediate reload of the program counter after a BTAC match.



BTB is a small cache memory accessed during the instruction fetch stage using the instruction fetch address. Each entry of the BTB contains two fields:

a. **Branch instruction address** (**BIA**) ;
b. **Branch target address** (**BTA**).

When a static branch instruction is executed for the first time, an entry in the BTB is allocated for it. The BTB is accessed concurrently with the accessing of the I-cache.
When the current program counter's content matches the BIA of an entry in the BTB, a hit in the BTB results. This implies that the current instruction being fetched from the I-cache has been executed before and is a branch instruction. When a hit in the BTB occurs, the BTA field of the hit entry is accessed and can be used as for the next instruction fetch address.

There are two main methods for branch prediction: static prediction and dynamic prediction.

Static branch prediction predicts always the same direction for the same branch during the whole program execution.
Dynamic branch prediction uses the special hardware for prediction.
The simplest form of static prediction is to design the fetch hardware to be biased for not taken. When a branch instruction is encountered, prior to its resolution, the fetch stage continues fetching down the fall-through path without stalling.
This is not very effective method!

The most common branch condition speculation technique is based on the history of previous branch execution.

History-based branch prediction makes a prediction of the branch direction, whether taken or not taken, based on previously observed branch directions.

Branch prediction is effective only then, if the branch is predictable.

Terminology (Branch Prediction Technique)

Acronym		Branch Address	Branch History	Target Address	Target Instruction
BHT	**Branch history table**	Yes	Yes		
BTAC	**Branch target address cache**	Yes		Yes	
BTIC	**Branch target instruction cache**	Yes			Yes
BTB	**Branch target buffer**	Yes	Yes	Yes	
BPC	**Branch prediction cache**	Yes	Yes	Yes	Yes

Amount of Parallelism Available within a Basic Block

Basic block — a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
//baas- ehk põhiplokk//

For typical programs the average dynamic branch frequency is often between 20% and 33%, which meaning that between four and seven instructions execute between a pair of branches.
The simplest way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is called **loop-level parallelism**.

Generally ILP aims at speeding up the single processors.

Widespread Methods of Increasing the ILP

Hardware techniques

a. Speculative execution instructions;
b. Register renaming;
c. Out-of-order issue with instructions lookahead.

Software techniques (usually incorporated into the compilers as optimizations)

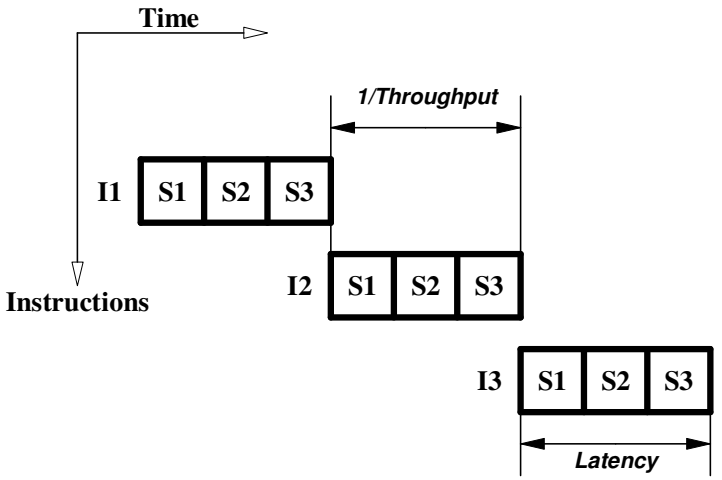
a. Trace scheduling;
b. Loop unrolling;
c. Software pipelining;
d. Optimal register allocation algorithms;
e. Static branch prediction.

Instruction-Level Parallelism Techniques

Technique	Reduces
Forwarding and **bypassing**	Potential data hazard stalls
Delayed branches and **simple branch scheduling**	Control hazard stalls
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependencies and output dependencies
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	[Ideal CPI]
Speculation	Data hazard and control hazard stalls
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Software pipelining, trace scheduling	[Ideal CPI] Data hazard stalls
Compiler speculation[4]	[Ideal CPI] Data and control stalls

INSTRUCTION PIPELINING

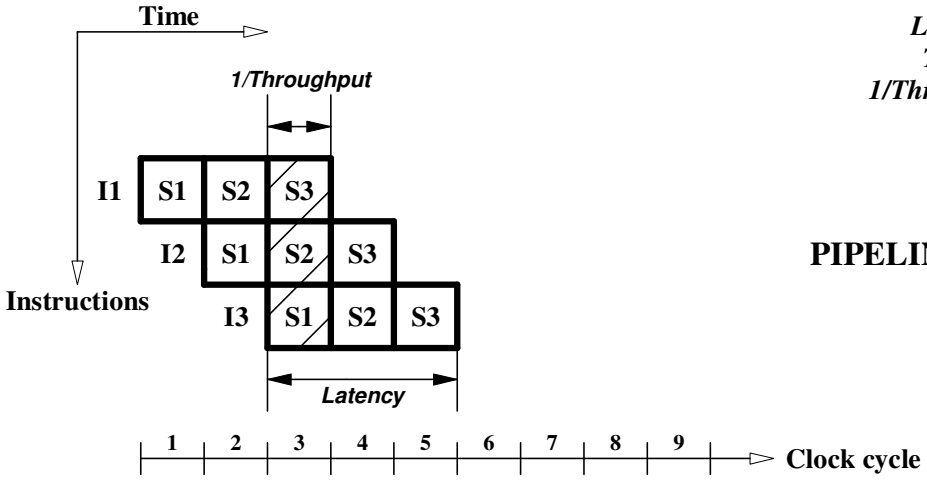
Pipelining is an instructions implementation technique in which multiple instructions are overlapped in execution.



UNPIPELINED

SYNCHRONOUS SYSTEM

***Latency* => Instruction total execution time**
***Throughput* [instructions per clock cycle]**
***1/Throughput* [clock cycles per instruction => *CPI*]**



PIPELINED

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. In the instruction pipeline multiple instructions are overlapped in exzecution.
Pipelining yields a reduction in the average execution time per instruction. The reduction can be viewed as decreasing the number of clock cycles per instruction, as decreasing the clock cycle time, or as combination.

Machine cycle – the time required between moving an instruction one step down the pipeline.

The length of a machine cycle is determined by the time required for the slowest pipeline stage.
The motivation for an **n-stage pipeline** is to achieve an **n-fold increase in throughput**.

The **ideal pipeline** is based on three idealized assumptions (**pipelining idealism**):

1. **Uniform subcomputations**
2. **Identical computations**
3. **Independent computations**

Pipelines Taxonomy

1. **Arithmetic** pipeline
   Instruction pipeline

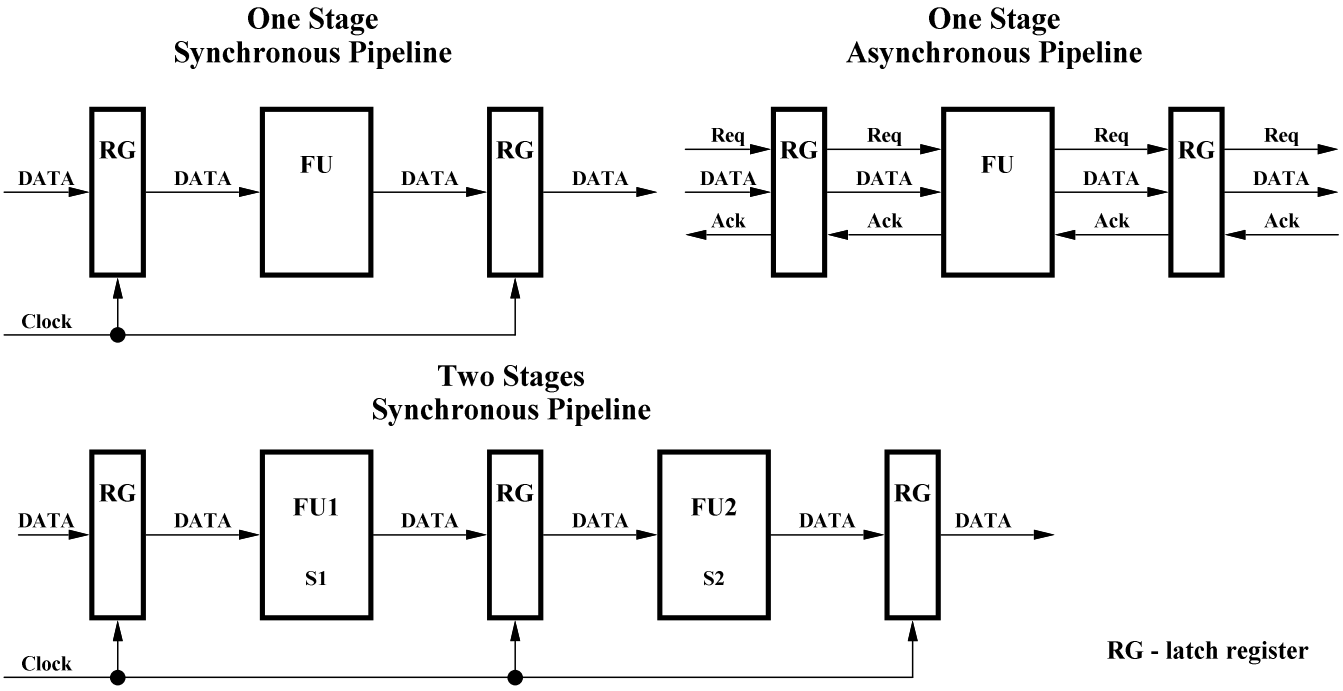
2. **Single function** (**unifunction**) pipeline
   Multiple function* pipeline
   2a. **Static** (traditional) pipeline.
   It is **unifunctional** until configuration changes
   2b. **Dynamic** pipeline.

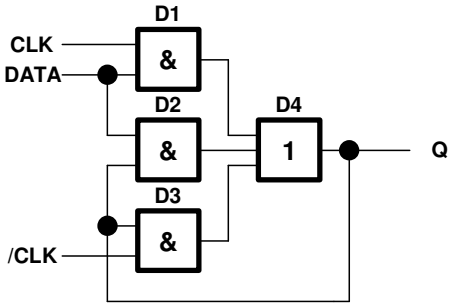
(3). Non-configurable pipeline
Configurable pipeline:
a. **Statically configurable** (2a)
(at any given moment only one active configuration exists)
b. **Dynamically configurable** (2b)
(several active configurations, *configurable on-the-fly*)

4. **Synchronous** pipeline
   ($\mathbf{t_{stage}}$ = const)
   Asynchronous pipeline
   ($\mathbf{t_{stage}}$ = var.)

5 **Scalar** pipeline
Vector pipeline

Pipeline Models





1-bit Earle latch

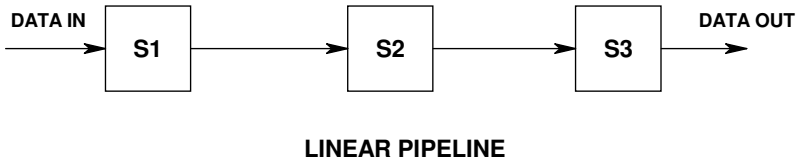
The information is stored when CLK=1 and retained as long as CLK=0. The value on the DATA line should not be altered while CLK=1.
The D1-gate is used to enter new data, the D3-gate is used to maintain old data, and D2-gate is used to eliminate logic hazards.

Synchronous pipeline model

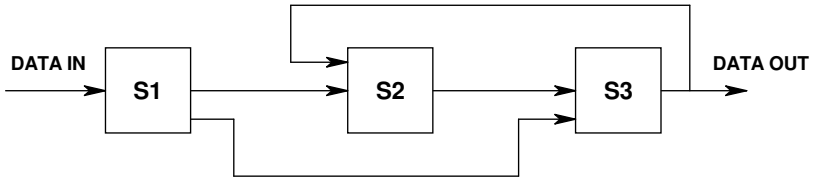
$$t_{stage} = \text{fixed}$$
$$t_{stage} = t_{FU} + t_{latch}$$
$$t_{clock} = \varphi(t_{stage})$$

Linear and non-linear (folded-back) pipelines

A folded-back pipeline is a way of avoiding synchronization overhead by using the same pipeline stage for more than one step in the processing. This strategy will only partially succeed because partitioning is imperfect, and even a perfect partition may have variance due to data-dependency.



LINEAR PIPELINE



NON-LINEAR PIPELINE

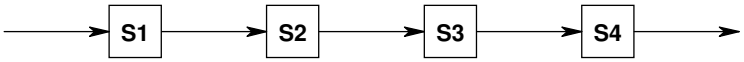
Asynchronous pipeline benefits

1. **Increased performance**
2. **Power saving**
3. **No clock problems**

Reservation Table

In a reservation table, each row corresponds to a stage in the pipeline and each column corresponds to a pipeline cycle.
The intersection of the i^{th} row (stage) and j^{th} column (clock) indicates that the stage i would be busy performing a subtask at a cycle j; where the cycle 1 corresponds to the initiation of the task in the pipeline. That is, stage i is reserved (not available for any other task) at cycle j.

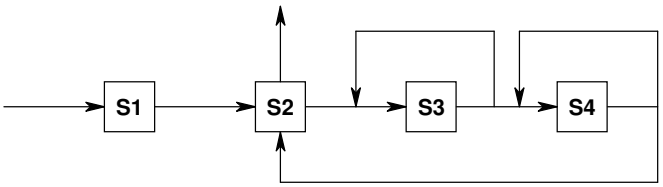
A. Reservation table for **4-stage linear pipeline**:



Stage \ Clock	1	2	3	4
S1	X			
S2		X		
S3			X	
S4				X

The reservation table shows that each stage completes its task in one clock cycle time, and hence, an instruction cycle requires four clock cycles to be completed. The number of clock cycles that elapse between two initiations is referred as latency.
The problem is to properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high (maximum) throughput.
Collision $\Rightarrow$ if two or more initations attempt to use the same pipeline stage at the same time.

B. Reservation table for **4-stage nonlinear pipeline**



Stage \ Clock	1	2	3	4	5	6
S1	X					
S2		X				X
S3			X	X		
S4				X	X	

The last two stages in the pipeline are used twice by each operand. Stage 3 (**S3**) produces a partial result and passes it on to stage 4 **(S4**). While stage 4 is processing the partial result, stage 3 produces the remaining part of the result and passes it to stage 4. Stage 4 passes the complete result to stage2 (**S2**). Because stages 3 and 4 are used in subsequent cycles by the same set of operands, the operand input rate cannot be as fast as one per cycle.

Pipelining Efficiency

We assume that:

$\mathbf{N_i}$ - the number of **i-type instructions** in the task,
$\mathbf{t_{ST}}$ - the pipeline's ***start-up*** **time**,
$\mathbf{T_i}$ - instruction **i** processing time,
$\mathbf{t_{cy}}$ - the **cycle time** in the **n-stage pipeline**.

$$\mathbf{T_i \approx t_{ST}}$$
$$\mathbf{t_{ST} = n \times t_{cy}}$$

Total un-pipelined processing time ($\mathbf{T_{totun}}$) is:

$$\mathbf{T_{totun} = N_i \times T_i = N_i \times t_{cy} \times n}$$

Total pipelined processing time ($\mathbf{T_{totp}}$) is:

$$\mathbf{t_{ST} = T_{I,}}$$
$$\mathbf{T_{totp} = \underline{(N_i\text{-}1) \times t_{cy}} + t_{ST} = t_{cy} \times (N_i\text{+}n\text{-}1)}$$
$$\downarrow$$

In each $\boldsymbol{t_{cy}}$ *pipeline produces a result for one instruction.*

The speed-up ($\mathbf{S_{Ni}}$) of pipelining, for $\mathbf{N_i}$ is given by:

$$\mathbf{S_{Ni} = T_{totun} / T_{totp} = (N_i \times t_{cy} \times n) / (t_{cy} \times (N_i\text{+}n\text{-}1))}$$

$$\mathbf{S_{Ni} = (N_i \times n) / (N_i\text{+}n\text{-}1) = n / (1\text{+}n/N_i\text{–}1/N_i)}$$

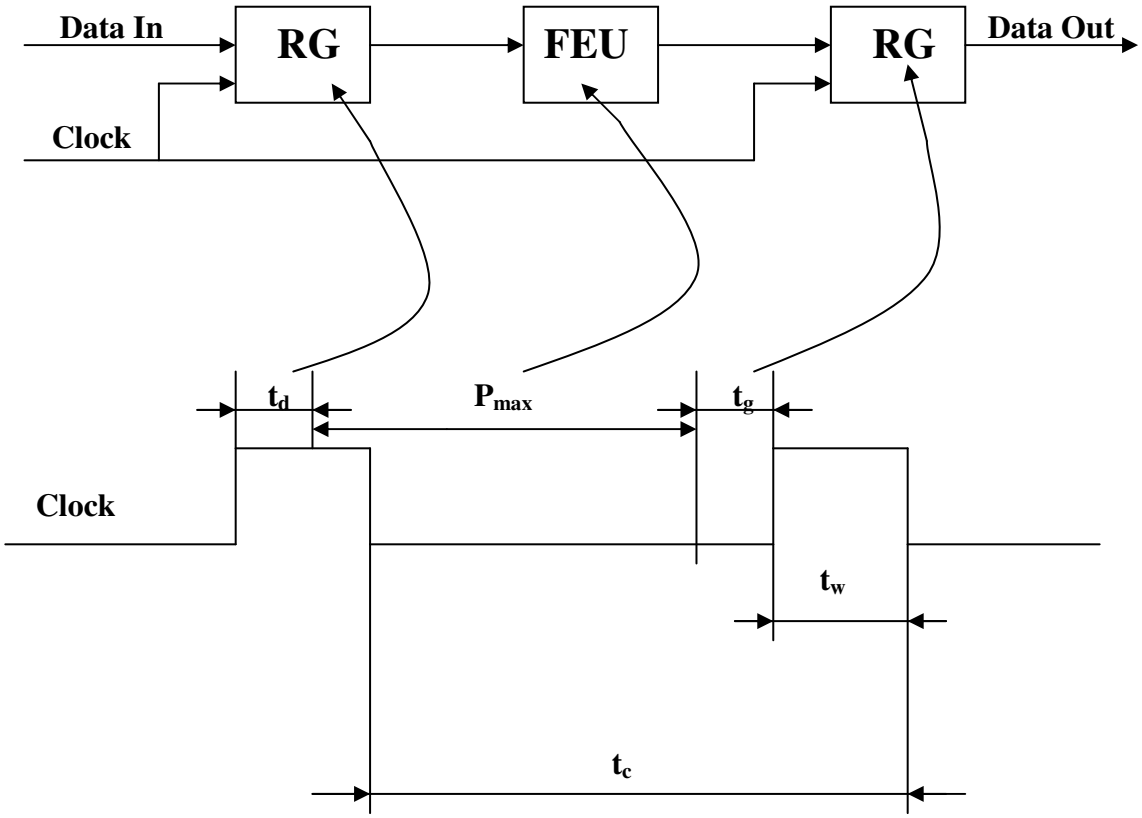
If more $\mathbf{N_i}$ are processed ($\mathbf{N_i \to \infty}$)**,** then

$$\lim_{N_i \to \infty} \mathbf{S_{Ni} = n}$$

In the limit, the pipeline of **n** stages will be **n times as fast** as the corresponding non-pipelined unit.

Pipeline Clocking

Single Phase Clocking



t_c – cycle time
t_w – clock pulse width
t_d – register output delay after data and clock are enable
t_g – register data setup time
P_{max} – maximum delay n FEU (without clock overhead **C**)
C – clock overhead

$$C = t_g + t_d$$

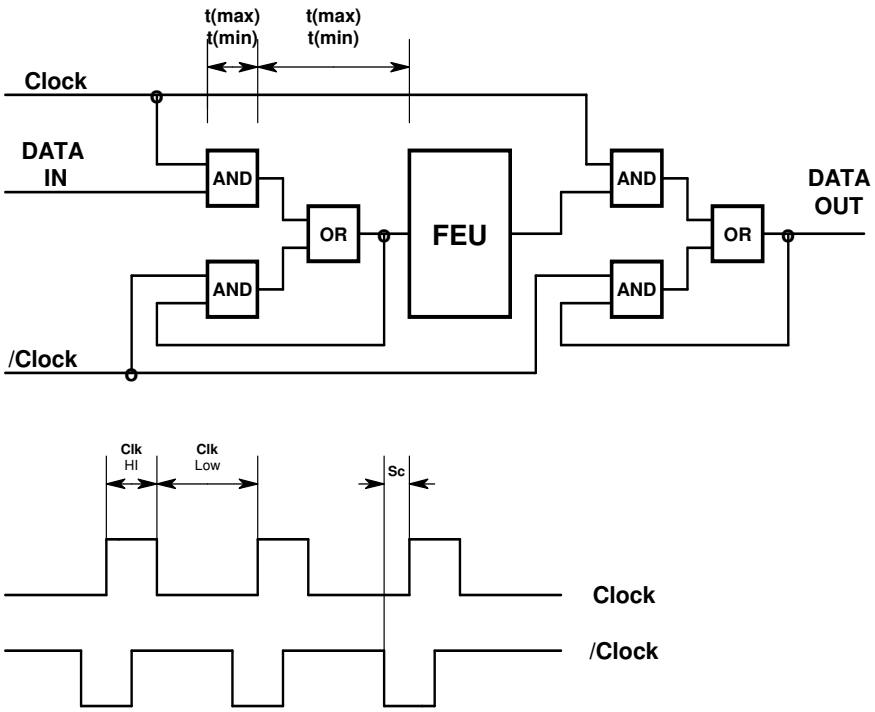
Stage cycle time — the difference between the time that an instruction enters a pipeline stage and the time that it leaves the stage.

Computation time - the difference between the time that an instruction enters a computation block and the time that all computed outputs are stable.

Idle time — the time that an instruction waits (at stage **i**), after computation has been completed, before it moves to the next stage (**i+1**).

Synchronization time - the time used to synchronize two adjacent stages which, for the clock case, includes clock skew and latching time.

Stage clocking circuit



Clock(s) skew (t_{Sc})

The **clock skew** //*taktimpulsi kiivamine*// is **determined by differences in:**

1. Clock signal lengths;
2. Differences in line parameters;
3. Differences in delays through active line elements;
4. Differences in latch threshold voltages.

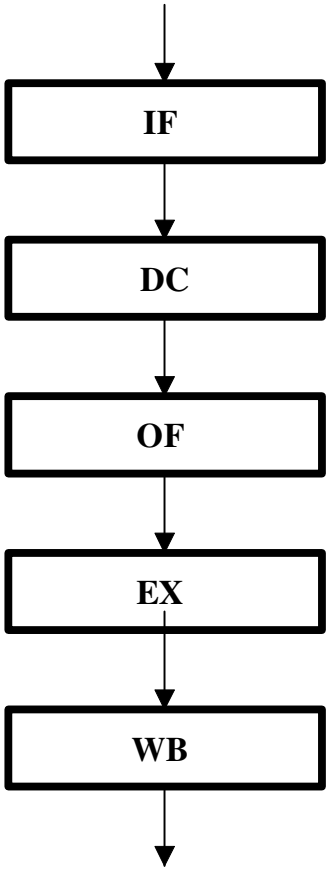
One architectural method for increasing processor performance is increasing the frequency by implementing deeper pipelines.

Clock cycle **overhead time** is a portion of the cycle used for **clock skew**, **jitter** //*taktimpulsi värin*//, **latching** //*lukustusviide*// and other pipeline overheads.

If we assume, that the **overhead per clock cycle is constant** in a given circuit technology, then we can **increase the processor frequency** by **reducing** the ***useful time*** per clock cycle.

1. Latches are faster than flip-flops.
2. Pipelines often are built with **two-phase clock** (MS flip-flop with logic in between stages).

Generic Instructions Pipeline



IF – **I**nstruction **F**etch
DC – **I**nstruction **D**ecode
OF – **O**perand **F**etch
EX – **E**xecute
WB (**OS**) – **W**rite **B**ack (**O**perand **S**tore)

Special logic is needed for each pipeline stage.

Amdahl`s Law Applied to the Pipelined Processor

$$\mathbf{S_{pip}} = \frac{1}{(1-q)+\frac{q}{n}}, \quad \text{where}$$

$\mathbf{S_{pip}}$ – speedup of the pipeline
q – fraction of the time when the pipeline is filled
(1-q) – the fraction of time when the pipeline is stalled
n – the number of the pipeline stages

It is assumed that whenever the pipeline is stalled, there is only one instruction in the pipeline or it becomes a sequential nonpipelined processor.

Deep Pipelines

Processor performance can monotonically increase with increased pipeline depth, but due to unpredictable nature of code and data streams, the pipeline cannot always be filled correctly and the flushing of the pipeline exposes the latency.
These flushes are inevitable, and pipeline exposures decrease IPC as the pipeline depth increases. The branch misprediction latency is the single largest contributor to performance degradation as pipelines are stretched.

The overall performance degrades when the decrease in IPC outweighs the increase in frequency.

The higher performance cores, implemented with longer (deeper) pipelines, will put more pressure on the memory system and require large on-chip caches.
Increasing the pipeline depth divides the computation among more cycles, allowing the time for each cycle to be less.
Ideally, doubling the pipeline depth (n) would allow twice the frequency.
In reality, some mount of delay overhead ($T_{overhead}$) is added with each new pipeline stage, and this overhead limits the amount of frequency improved by increasing pipeline depth.

$$\mathbf{f} = \frac{1}{T_{cycle}} = \frac{1}{\frac{T_{\log ic}}{n} + T_{overhead}} = \frac{n}{T_{\log ic} + n \times T_{overhed}}, \quad \text{where}$$

$\mathbf{T_{cycle} = t_{CY}}$
$\mathbf{T_{logic}}$ – delay in stage logic circuits

The equation shows how frequency is improved by dividing the logic delay of the instruction among a deeper pipeline.

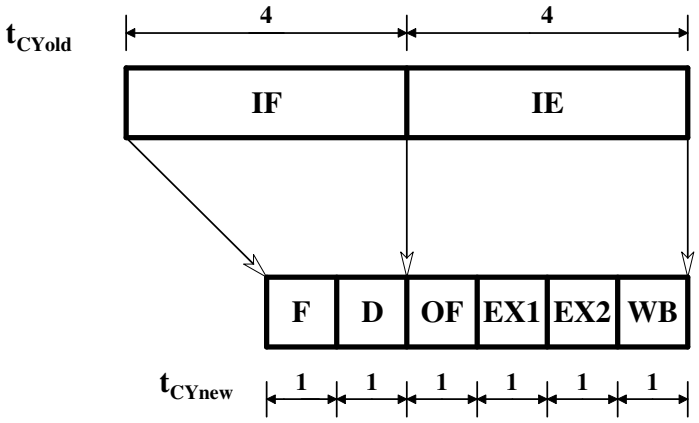
Doubling frequency requires increasing pipeline depth by more than a factor of 2.

Increasing the pipeline depth increases the number of pipeline stalls per instruction and reduces the average instructions per cycle.

Example

Pipeline depth increasing

$$T_{tot} = (N_i - 1) \times t_{CY} + t_{ST} = t_{CY} \times (N_i + n - 1)$$
$$t_{stage} = \text{fixed}$$



$n_{old} = 2$

$t_{CYold} = 4$ $f_{old} = 0{,}25$

$n_{new} = 6$

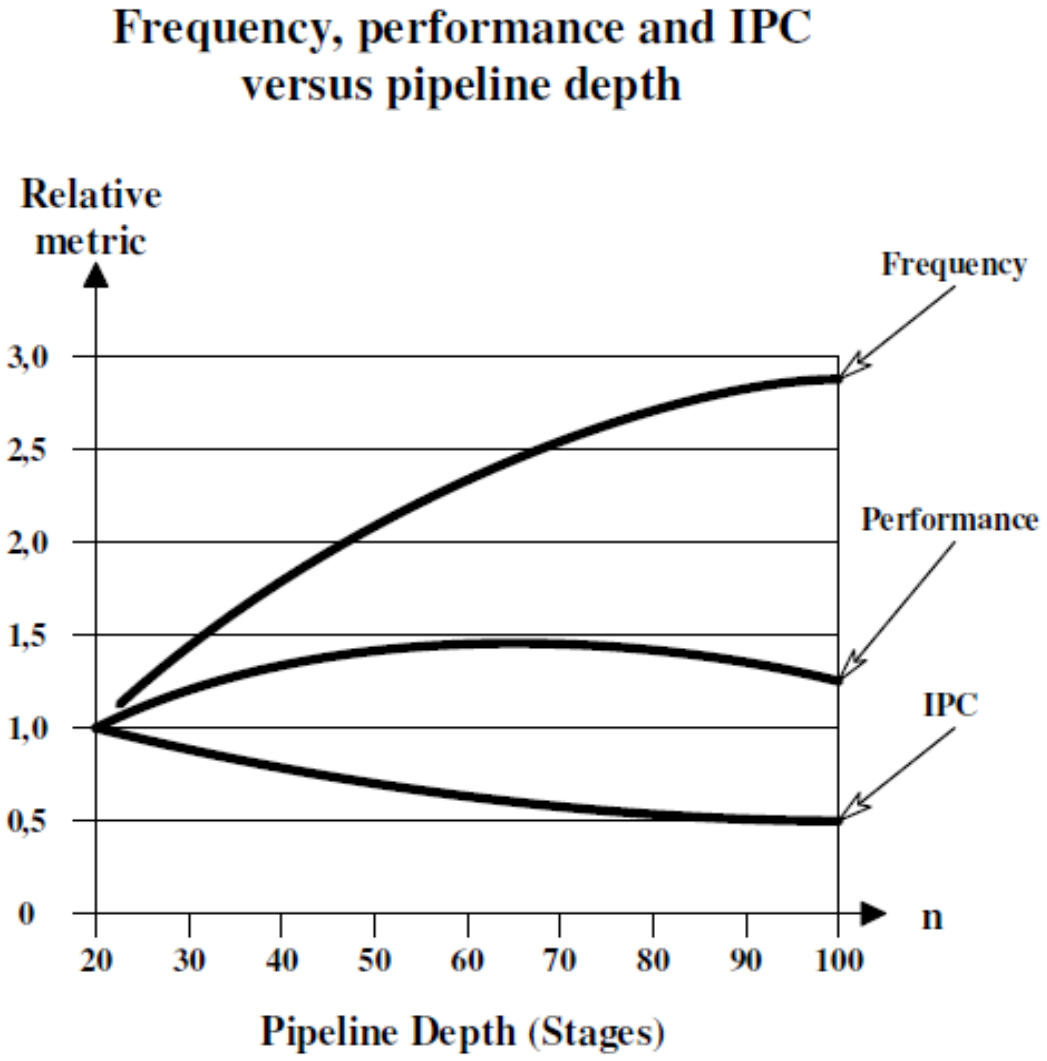
$t_{CYnew} = 1$ $f_{new} = 1$

IF - instruction fetch; IE - instruction execution
F - fetch; D - decode; OF - operand fetch; EX1, EX2 - execution; WB - write back

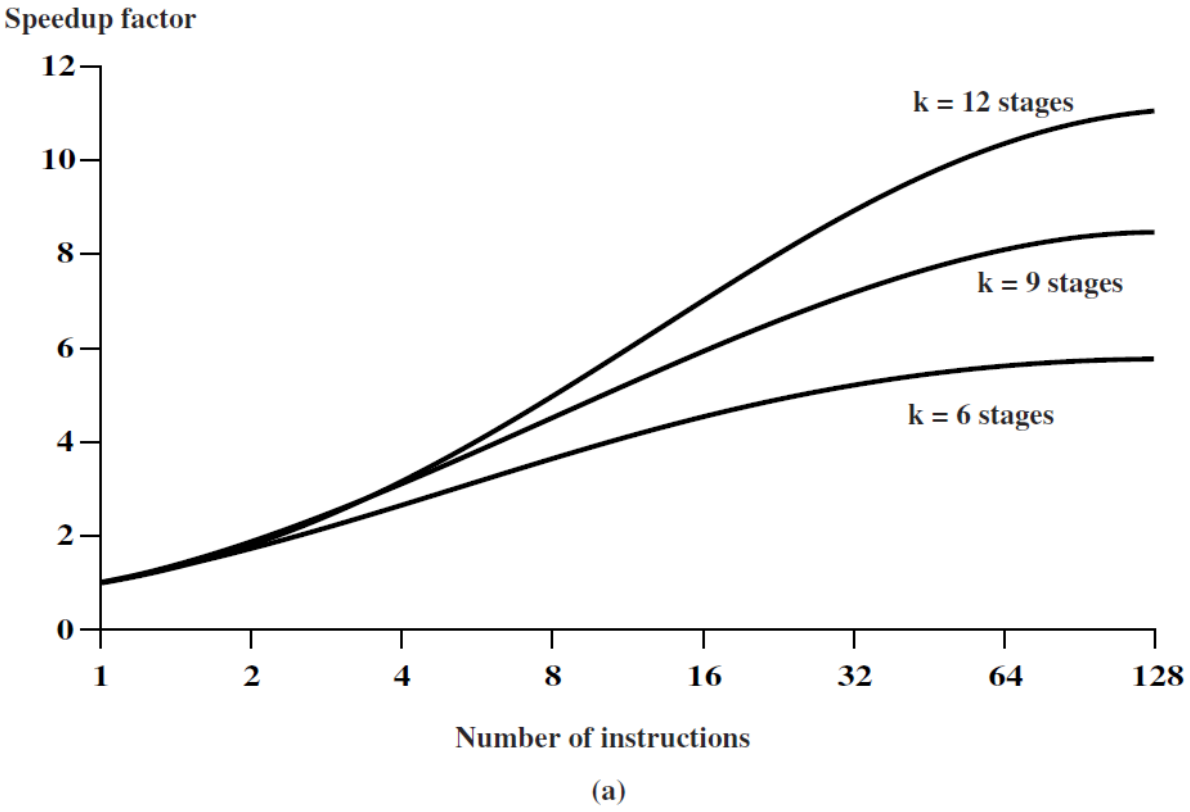
$$n_{new} = n_{old} + \underline{4}$$
$$f_{new} / f_{old} = \underline{4}$$
$$T_{totold} = t_{CYold} \times (N_i + n_{old} - 1) = 4 \times (N_i + 2 - 1) = 4N_i + 4$$
$$T_{totalnew} = t_{CYnew} \times (N_i + n_{new} - 1) = 1 \times (N_i + 6 - 1) = N_i + 5$$
$$\text{Speedup} = T_{totalold} / T_{totalnew}$$

$$\text{Speedup} = \frac{4N_i + 4}{N_i + 5} = \frac{4 + \frac{4}{N_i}}{1 + \frac{5}{N_i}}$$

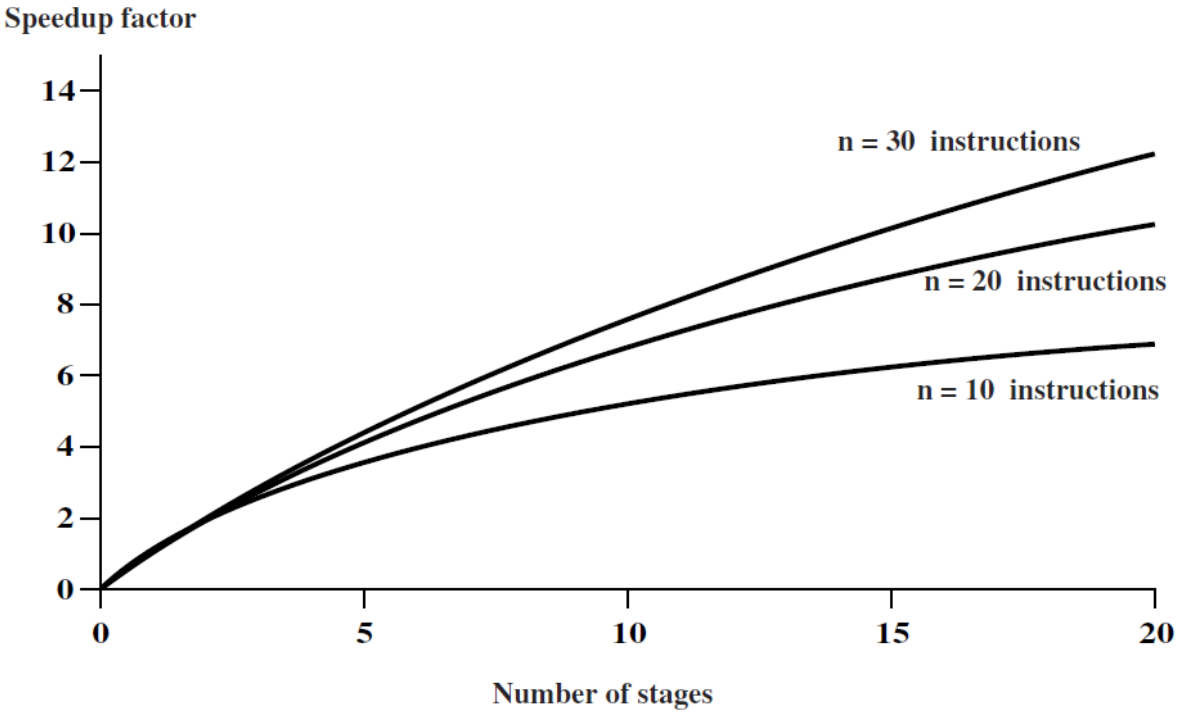
$$\text{Speedup} = \underline{4}$$
$$\lim N_i \to \infty$$



In figures relative metric is tied to ***Pentium 4*** microprocessor technical characteristics.

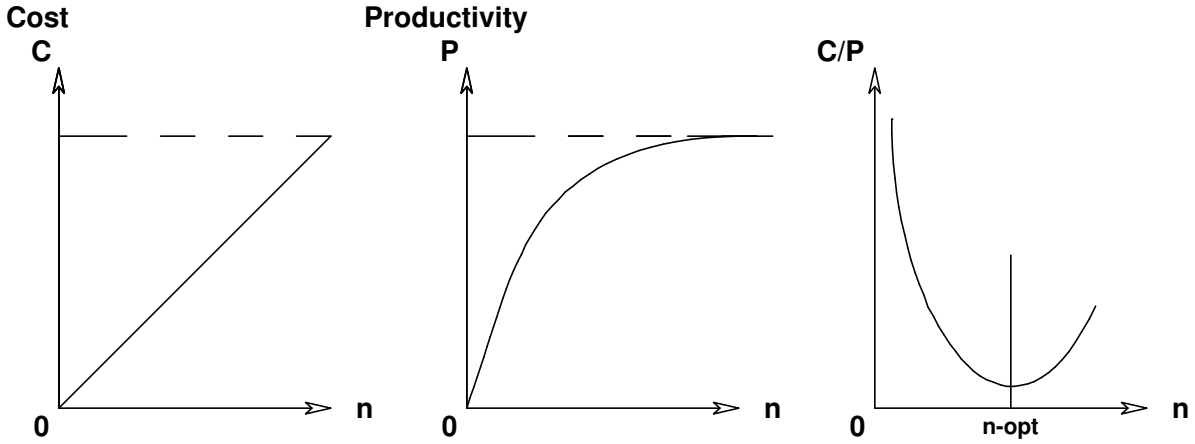


(a)



NB! It is supposed, that the instructions are processed with no branches.

The **real improvement** depends upon how much circuit design minimizes the delay overhed per pipestage and how much microarchitectural improvements offset the reduction in IPC.



The larger the number of pipeline stages, the greater the potential for speedup. Deep pipelines, which implement a fine-grained decomposition of a task, only perform well on long, uninterrupted sequences of the task iteration.

The Optimum Pipeline Depth for a Microprocessor

(by A. Hartstein and Thomas R. Puzak)

The **optimum pipeline depth** {the number of pipeline stages} ($\mathbf{n_{opt}}$) for microprocessors can be expressed as:

$$n_{opt}^2 = \frac{N_I \times t_p}{\alpha \times \gamma \times N_H \times t_o}, \quad \text{where}$$

$\mathbf{N_I}$ – the number of instructions;

$\mathbf{N_H}$ – the number of hazards (each hazard causes a full pipeline stall);

$\mathbf{t_p}$ – the total delay of the pipeline (processor);

$\mathbf{t_o}$ – the latch (between pipeline stages) overhead for the technology being used;

$\boldsymbol{\alpha}$ – an average degree of superscalar processing (whenever the execution unit is busy).
The $\boldsymbol{\alpha}$ varies with the workload running at the processor.
For <u>scalar</u> processors $\boldsymbol{\alpha = 1}$ and for <u>superscalar</u> processors $\boldsymbol{\alpha > 1}$;

$\boldsymbol{\gamma}$ – the weighted average of the fraction of the pipeline stalled by hazards.

$$\mathbf{0 \le \gamma \le 1}$$

Some observations

1. The optimum pipeline depth increases for workloads with few hazards ($\mathbf{N_H\downarrow; n_{opt}\uparrow}$).
2. As technology reduces the latch overhead (t_o), relative to the total logic path (t_p), the optimum pipeline depth increases ($\mathbf{t_o\downarrow;n_{opt}\uparrow}$).
3. As the latch overhead increases relative to the total logic delay, the optimum pipeline depth decreases ($\mathbf{t_p/t_o\downarrow; n_{opt}\downarrow}$**).**
4. As the degree of superscalar processing (α) increases, the optimum pipeline depth decreases ($\mathbf{\alpha\uparrow; n_{opt}\downarrow}$).
5. As the fraction of the pipeline that hazards stall (γ) decreases, the optimum pipeline depth increases ($\mathbf{\gamma\downarrow;n_{opt}\uparrow}$).

Shallow–and-Wide Pipeline *versus* Deep-and–Narrow Pipeline

Because the pipeline stages may take multiple cycle times, some processor architectures add additional stages to sub-divide the work performed at each stage, so the pipeline can move lock step with the processor clock speed. This results in two basic pipeline models:

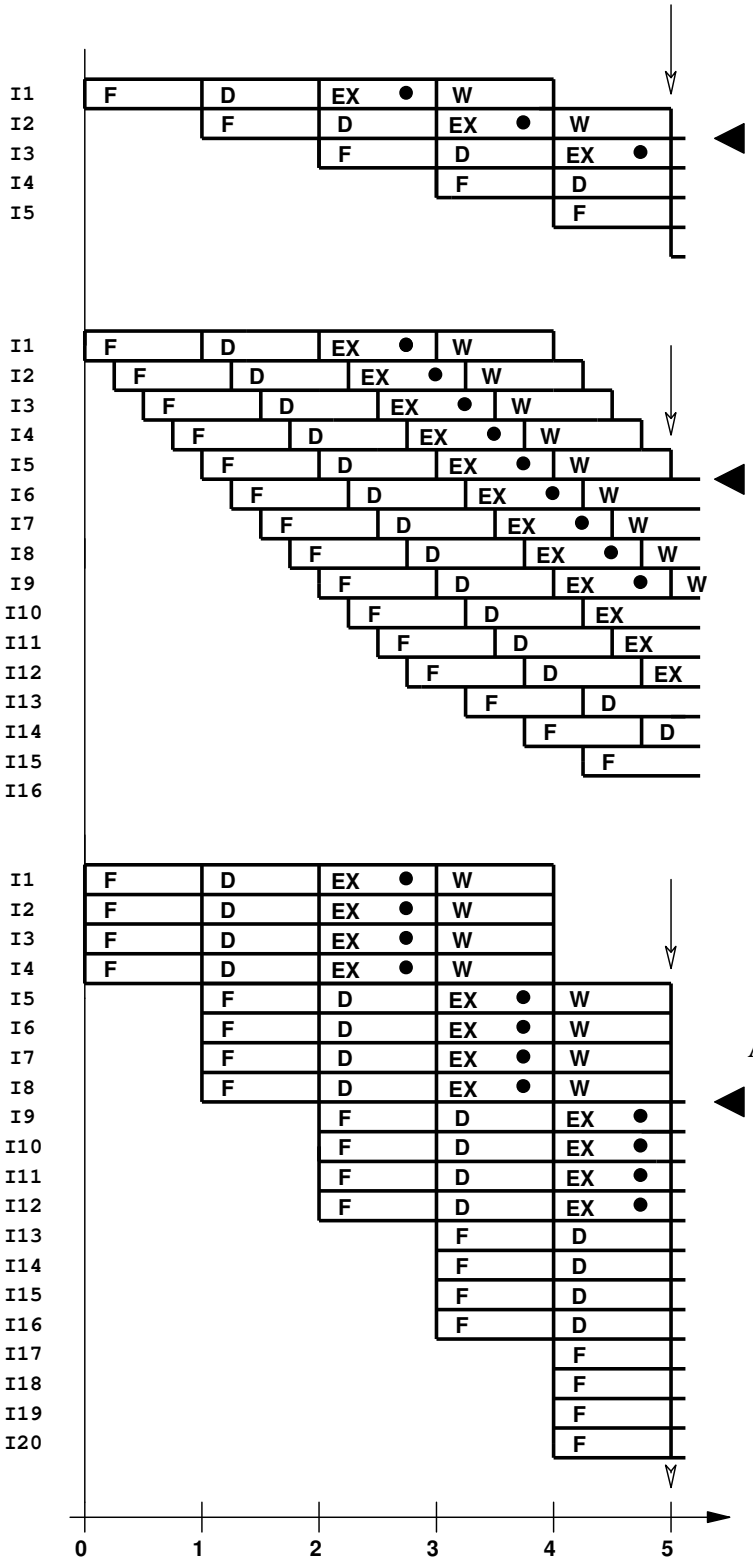
1. **Shallow–and-wide pipeline** //*lühike ja lai konveier*//;
2. **Deep-and–narrow pipeline** //*pikk ja kitsas konveier*//.

- The shallow–and–wide model is designed for energy efficiency. The shallow–and–wide approach uses a lower clock speed to the processor and a lower transistor count resulting in lower power consumption with the less thermal dissipation.
  The shallow–and–wide pipeline model is better suited for very “branchy” code.

- In the deep–and–narrow model additional pipeline stages can reduce the amount of work performed at each stage and can increase the clock speed of the processor.
  The deep-and–narrow model is well suited to linear code with few branches where are doing a lot of repetitive operations.
  The “branchy” code will perform poorly, regardless of clock speed.

Base-, Super- and Superscalar Pipelines

Operational latency is the time (in cycles)
until the result of an instruction is available for use as an operand in a subsequent instruction.



Base Pipeline

Per cycle is issued 1 instruction.

Superpipeline

In a superpipeline of degree m,

Per cycle is issued 1 instruction.
Simple instruction latency measured m cycles is m.

m=4

Superscalar Pipeline

A superscalar pipeline of degree n can issue n instructions per cycle.
Per cycle is issued 4 instructions.
Simple instruction latency measured in cycles is 1.

n=4

Pipeline Hazards

Correct operation of a pipeline requires that operation performed by a stage must not depend on the operation(s) performed by other stage(s).
Dependencies are a property of programs. Presence of dependence indicates potential for hazard (risk), but actual hazard and length of any stall is a property of the pipeline.

Hazard – the situation that prevents the next instruction in the instruction stream from executing during its designated clock cycle.

Classes of Hazards

1. Structural hazards (**resource conflicts**)

The hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

- Multiple copies of the same resource (replication).
- Instructions prefetch (forming instructions queue)

Starvation – the result of conservative allocation of resources in which a single process is prevented from execution because it's kept waiting for resources that never become available.

Critical region – the parts of program that must complete execution before other processes can have to the resources being used.

Deadlock – a problem occurring when the resources needed by some processes to finish execution are held by other processes, in turn, are waiting for other resources to become available.

Deadlock conditions:

1. **Mutual exclusion** //*vastastikune välistamine*// - only one process is allowed to have access to a dedicated resource.
2. **Resource holding** //*ressursihõive*// - it's an opposed to resource sharing.
3. **No preemption** //*puudub väljasaalimise*// **-** the lock of temporary reallocation of resources.
4. **Circular wait** //*ringootus*// - a process involved in the impasse is waiting for another to voluntarily release the resource.

All four conditions are required for the deadlock to occur and as long all four conditions are present the deadlock will continue; but if one condition can be removed the deadlock will be resolved.

2. Control hazards (**procedural conflicts**)

Instructions following a conditional branch instruction have procedural dependency on the conditional branch instructions.

Speculative execution = Branch Prediction + Dynamic Scheduling

3. Data hazards

Hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

Data hazards due to **register operands** can be determined at the **decode stage**.
Data hazards due to **memory operands** can be determined only after computing the **effective address**.

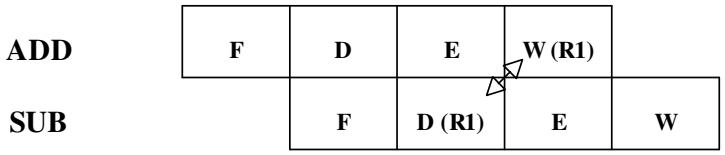
Data Hazard Types

Data hazards occur among instructions because the instructions may access (read, write) the same storage (a register or a memory) location.
There are the true dependencies as RAW hazards, because the *consuming instruction* can only read the value after the *producing instruction has written it*.
In addition are artificial dependencies (name dependencies) where two instructions use the same name but don't exchange data. These name dependencies result from WAR and WAW hazards.

Consider two **instructions A** and **B**, with **A** occurring before **B**.

1. **True data dependencies** (read after write or **RAW**)
   Instruction **B** ($\mathbf{I_B}$) tries to read a source before instruction **A** ($\mathbf{I_A}$) writes it, so **B** incorrectly gets the old value.

ADD **R1**, R2, R3	**R1**:=(R2)+(R3)	[R1/**WB**]	{$\mathbf{I_A}$}
SUB R4, R5, **R1**	R4:=(**R1**)-(R5)	[R1/**D**]	{$\mathbf{I_B}$}



2. **Output dependencies** (write after write or **WAW**)
   Instruction **B** tries to write an operand before it is written by instruction **A**.

3. **Antidependencies** (write after read or **WAR)**
   Instruction **B** tries to write a destination before it is read by instruction **A**, so instruction A incorrectly gets the new value.
   As in pipelines the values reading occurs before than writing results, such hazards are rare.

Data Hazards Summary

Instruction **b** follows instruction **a** in the program order.
WR(a) – register content is modified by instruction **a**.
RR(a) – register content is read by instruction **a**.

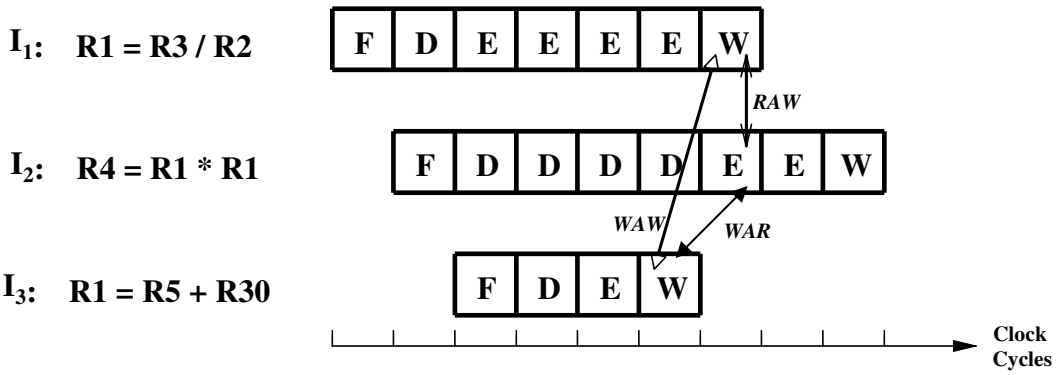
WAR: RR(a) ∩ WR(b) ≠ Ø;
RAW: WR(a) ∩ RR(b) ≠ Ø;
WAW: WR(a) ∩ WR(b) ≠ Ø.

Terms that are used for various types of data dependencies

Kogge	*Flynn*	*Johanson*
RAW	**Essential**	**Data**
WAW	**Output**	**Output**
WAR	**Ordering**	**Anti**

Example

Execution instructions (**I_1, I_2, I_3**) on an out-of-order CPU which has two execution units:



Data Hazards Involving Registers

1. Instruction **I_3** is able to enter the execution stage even before **I_2**, since **I_3** does not depend on any result of the proceeding instructions. It even terminates before **I_1** and this causes a **WAW hazard**.
2. Instruction **I_2** tries to read R1 before **I_1** writes it. There is a **RAW hazard**.
3. Since instruction **I_3** writes R1 before **I_2** reads it, there is a **WAR hazard**.

Dealing with Hazards in Pipelines in General

Issue:	**a.** *check structural hazards*;	
	b. *check for WAW hazards*	[stall issue until hazard cleared]
Read operand:	*check for RAW hazards*	[wait until data ready]
Execution:	execute operations	
Write back:	*check for WAR*	[stall write until clear]

Generally, the **CPI for a pipelined processor** is equal to:

Pipeline CPI = Ideal Pipeline CPI + Structured stalls + Data hazard stalls + Control hazard stalls

Solution Strategies for Pipeline Hazards Caused by Resource Conflicts

General resource conflicts might be avoided by **instruction scheduling** or by **resources replication**.

Solution Strategies for Pipeline Hazards Caused by Data Dependences

1. **Data speculation;**
2. **Operand forwarding** (result (register) forwarding), **load forwarding**, **store queue**;
3. **Software scheduler**;
4. **Hardware scheduler** (out-of-order execution).

Data speculation

Instructions, threads whose operands are not yet available, are executed with predicted data.
What can be speculated on?

a. **Register operand**;
b. **Memory address**;
c. **Memory operand**.

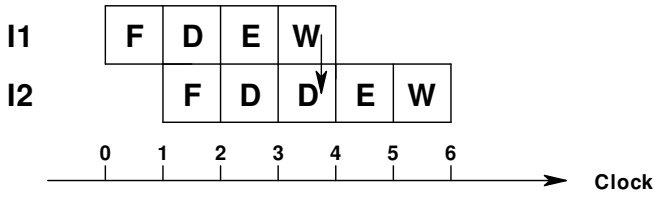
Register (Operand) Forwarding

A. Hardware operand forwarding allows the result of one ALU operands to be available to another ALU operation in the cycle that immediately follows.

To **reduce pipeline stalls** by **data hazard**, the **register forwarding** (**bypass)** is used to handle **RAW**.

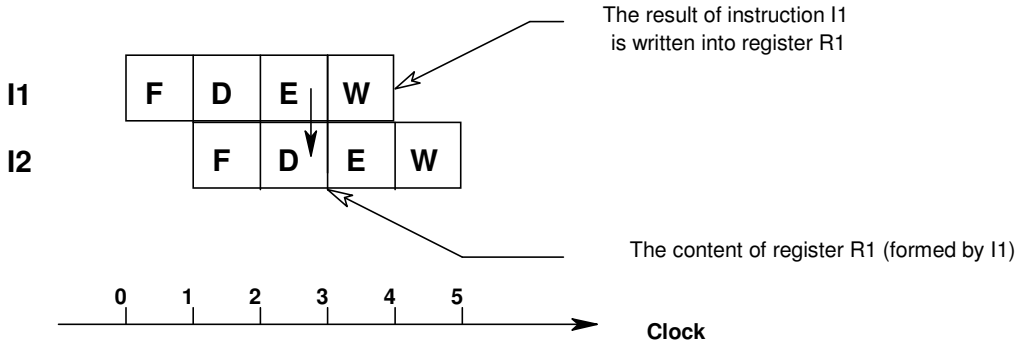
I1: **R1** := R2 + R3
I2: R4 := **R1** + R5

A **4-stage pipeline** that **does not support register forwarding**



a. **R1** must be updated before **R1** in **I2** is read.
b. **R1** in **I1** is updated at **W** stage, but **R1** in **I2** is read in **D** stage.
c. The pipeline must be stalled (**I2 bubble**) until **R1** is updated in **I1** (**RAW hazard!**).

A **4-stage pipeline** that **supports register forwarding**



There are two types of complexity regarding register forwarding:

1. **Bypass control** (determines which result in pipeline should be forwarded).
2. **Result drive** (determines which destinations the result must be forwarded).

B. **Software operand forwarding** is performed in software by **compiler**.
This feature requires the compiler to perform data dependency analysis in order to determine the operand(s) that can possibly be made available (forwarded) to subsequent instructions.

Software Operand Forwarding

	Store – Fetch	**Fetch – Fetch**	**Store – Store**
Original instruction sequence	**Store R2, (R3)** M[R3] ← R2 ***Load*** **(R3), R4** R4 ← M[R3]	**Load (R3), R2** R2 ← M[R3] ***Load*** **(R3), R4** R4 ← M[R3]	**Store R2, (R3)** M[R3] ← R2 **Store R4, (R3)** M[R3] ← R4
Modified instruction sequence	**Store R2, (R3)** ***Move*** **R2, R4** R4 ← R2	**Load (R3), R2** ***Move*** **R2, R4**	**Store R4, (R3)** M[R3] ← R4

Solution Strategies for Pipeline Hazards Caused by Control (Procedural) Dependences

1. **Branch prediction //***hargnemise prognoosimine***//**;
2. **Delayed branches //***viidatud siire***//**;
3. **Traces** (**threads**) //lõimed//;
4. **Multipath techniques //***mitmiklõime meetodid***//.**

} **Control speculation**

Branch Prediction

Branch prediction – a method whereby a processor guesses the outcome of a branch instruction so that it can prepare in advance to carry out the instructions that follow the predicted outcome.

Static branch prediction is a prediction that uses information that was gathered before the execution of the program.

The common static strategies that are used to predict whether a branch will be taken or not (predict never taken, predict always taken, predict by opcode). Overall probability a branch is taken is about 60-70%, but probability of backward branch is ~90% and forward branch is ~50%.

Dynamic branch prediction uses information about taken or not-taken branches gathered at run-time to predict the outcome of a branch.

Branch history tables (BHT) are widely used in dynamic prediction strategy**.**

Branch penalty (B_P) can be evaluated as:

$$B_P = w_c \times (1\text{-}a) \times b_r \times IPC, \quad \text{where}$$

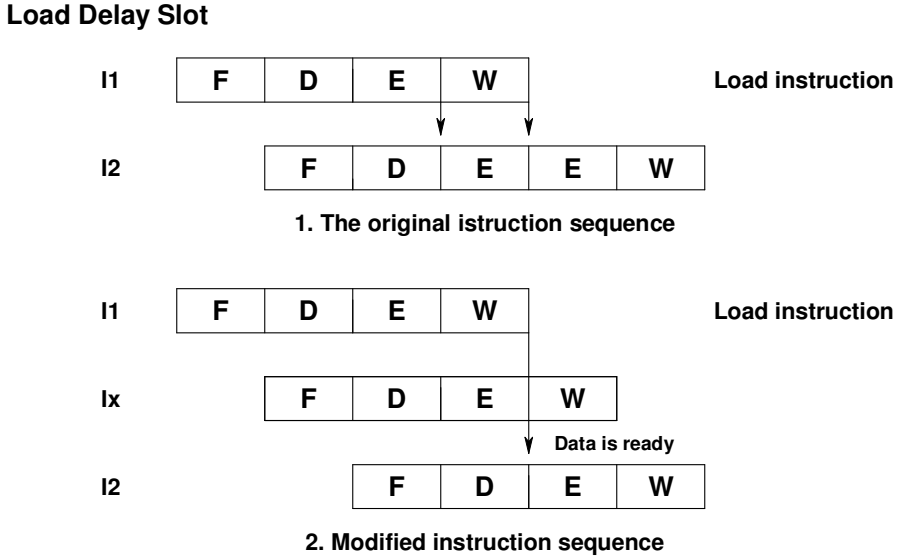
w_c – the number of clock cycles wasted due to a branch misprediction;
a – the prediction accuracy (**a = 0 ÷ 1**);
b_r – the ratio of the number of branches over the number of total instructions.

Branch Prediction and a Delay Slot

- It might appear that in the modern highly pipelined CPU the unconditional branches are not a problem. There is no ambiguity about where to go.
- The trouble lies in the nature of pipelining. As a rule, the instruction decoding occurs in the pipe's second stage. The fetch unit has to decide where to fetch from the next before it knows what kind of instruction it just got.
  Only one cycle later can it learn that it picked up an unconditional branch, but then it has already started to fetch the instruction following the unconditional branch.
- A substantial number of pipelined CPUs have the property that the instruction following an unconditional branch is executed, even though logically it should not be.
- The position after branch is called - delay slot //*viitepilu*//.

Delayed branch is a type of branch where the instruction, following the branch is always executed, independent of whether the branch condition is true or false.

The delay slot may be formed also after a load instruction (*Load Delay Slot*).



Processors with very deep pipelines could have two or more delay slots.
The more delay slots that exist after a conditional branch instruction, the more difficult it will be for the compiler to find useful, independent instructions with which to fill them.

Pipeline Performance with Delay Slots

Assume single cycle ($\mathbf{t_{cy} = 1}$) execution for all instructions except loads, stores and branches.
The **average number of cycles per instruction** ($\mathbf{CPI_{ave}}$) is given by:

$$\mathbf{CPI_{ave} = Pb \times (1+Bp) + Pm \times (1+Mp) + (1-Pb-Pm) \times (t_{cy}) =}$$
$$\mathbf{= 1 + Bp \times Pb + Mp \times Pm}, \quad (1) \quad \text{where}$$

Pb - probability that an instruction is a branch;
Bp - branch penalty;
Pm - probability that an instruction is memory reference;
Mp - memory reference penalty.

T. R. Gross and J. L. Hennessy developed an algorithm for optimizing delayed branches and have shown that the first branch delay slot can be filled with useful instructions more than half the time, while the subsequent delay slots are increasingly harder to fill.

Handling Conditional Branches in Modern Processors

The conditional branches are worse. Early pipelined CPUs stalled until it was known whether the branch would be taken or not.
There are different possibilities to handle conditional branches in modern processors.

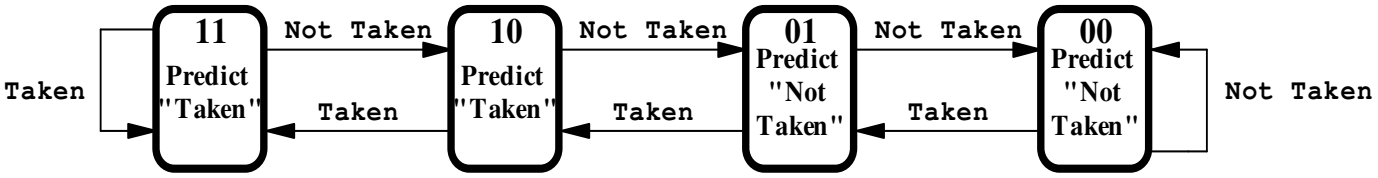
A. The Dynamic Branch Prediction

Branch history table ***//hargnemiste eelloo tabel//*** **(BHT)** logs conditional branches as they occur. The BHT is a small cache memory associated with the instruction fetch stage of pipeline. Each entry in the table contains of three elements: the branch instruction address, some history bits and information about the target instruction.
In most implementations the third field contains the address of the target instruction (in the branch target buffer BTB) or the target instruction itself.

C. 2-bit Branch Prediction

There is a BHT holding 2-bit counters for each of the recently accessed branches in processor. Each counter holds the value “00”, “01”, “10” or “11”. Whenever a branch is taken, its 2-bit counter is incremented if it is below the maximum value “11”.
Whenever a branch is not taken, its 2-bit counter is decremented. This incrementing or decrementing is "saturating” as incrementing stops at “11”, and decrementing stops at “00”. When a branch instruction is fetched, its address is used to look up the value of its counter for that branch.

If the value is a “**00**” or “**01**”, the branch is assumed to be not **taken**.
If the value is “**10**” or “**11**”, the branch is assumed to be **taken**.



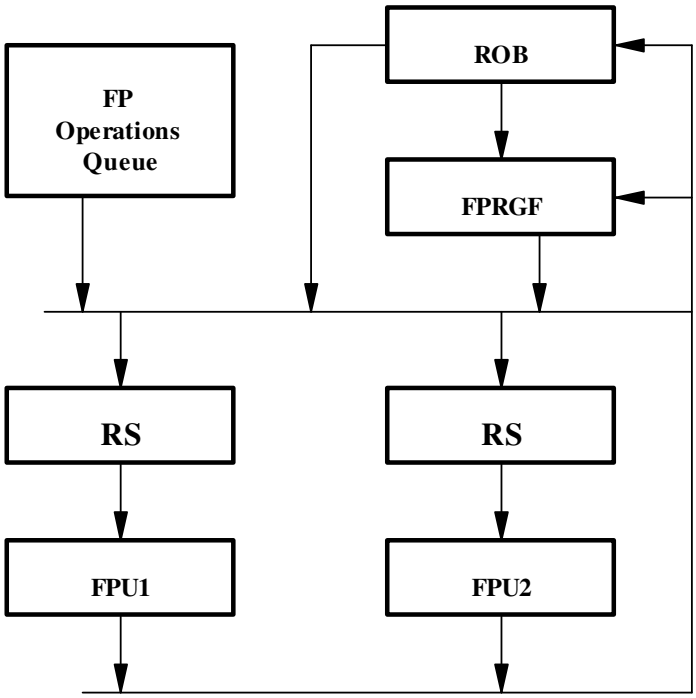
C. The Dynamic Pipeline Scheduling and Reservation Stations

Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. In processors the pipeline is divided into three major units:

1. **Instruction fetch unit** (**FU**),
2. **Execution unit** (**EU**),
3. **Commit unit** (**CMU**).

Each functional unit has buffers, so called reservation stations (RS) ***//ootejaam,ootepuhver//***.

Reservation station is a buffer within a functional unit that holds the operands and operation. As soon the buffer contains all its operands and the proper functional unit is ready to execute, the result is calculated.
When the result is completed, it is sent to any RSs waiting for this particular result as well as to the commit unit (CMU). Commit unit buffers the result until it is safe to put the result into the register file or for store into memory.
The special buffer in the CMU, the reorder buffer (ROB) //*ümberjärjestamise puhver*//, is used to supply operands.



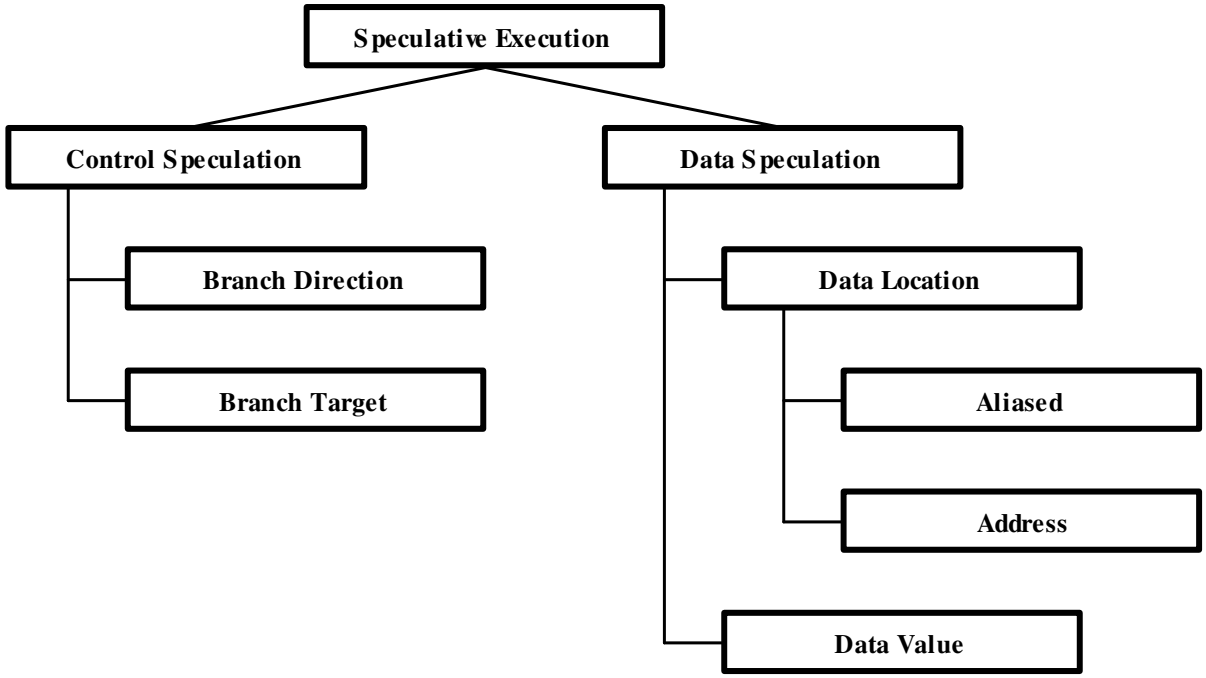
To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependencies to be tracked, and the commit unit is required to write results to registers and memory in program execution order. This mode is called in-order completion.

If an exception occurs, the processor can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception.
In this case the front end and the back end of the pipeline run in order, the execution units are free to initiate execution whenever that data they need is available.

C. Speculative Execution

Speculation – the processor (or compiler) guesses the outcome of an instruction so as to enable execution of other instruction that depend on the speculated instruction. Producing correct results during speculative execution requires result checking, recovery and restatrt hardware mechanisms:

Checking mechanisms to see if the prediction was correct;
Recovery mechanisms to cancel the effects of instructions that were issued under false assumptions (e. g., branch mispredictions);
Restart mechanisms to re-establish the correct instruction sequence.



1. **Control speculation** ***//spekulatiivne juhtimine//***
   Refers to the **execution of instructions before** it has been determined that they would be executed in the normal flow of execution.
2. **Data speculation** ***//spekulatiivsed andmed//***
   It refers to the execution of instructions on **most likely correct data values**.

Control Speculation and **Branches** in the Speculative Instructions Stream

1. Each speculative instruction is tagged with speculative bit which is carried throughout all pipelines;
2. The speculative instruction is stalled in the decode stage if second branch encountered before the previous speculative branch resolves;
3. When speculative branch resolves, then if:

   a. Prediction was correct ⇒ clear speculative bit on all speculative instructions,
   b. Prediction was incorrect ⇒ discard all instructions tagged as speculative.

To allow speculation past multiple branches multiple tagging bits are added per instruction, indicating on which outstanding branches the instruction is speculative.

Data Speculation

Data dependencies are a major limitation to the amount of ILP that processors can achieve.
Data value speculation can eliminate the ordering imposed by data dependencies.

Data dependencies cause a serialization on the execution of program instructions.

Data value speculation refers to the mechanisms that deal with data dependencies by predicting the value that flow through them and execute speculatively the instructions that consume the predicted value.

Data dependence speculation refers to the techniques that are based on predicting dependencies among instructions and executing speculatively the code by enforcing the predicted dependences.

Data value speculation is based on the observation that inputs and outputs of many instructions sometimes follow a predictable pattern.

Load value prediction is more powerful than load address prediction, since the memory access has to perform in order to obtain the predicted value, even if the memory address is correctly predicted.

Speculatively issued loads must be verified.

This is done by issuing them to the address computation unit when their source operands are available. The actual address is compared to the predicted one and in the case of a misprediction, the predicted load and those instructions dependent on it are re-executed.

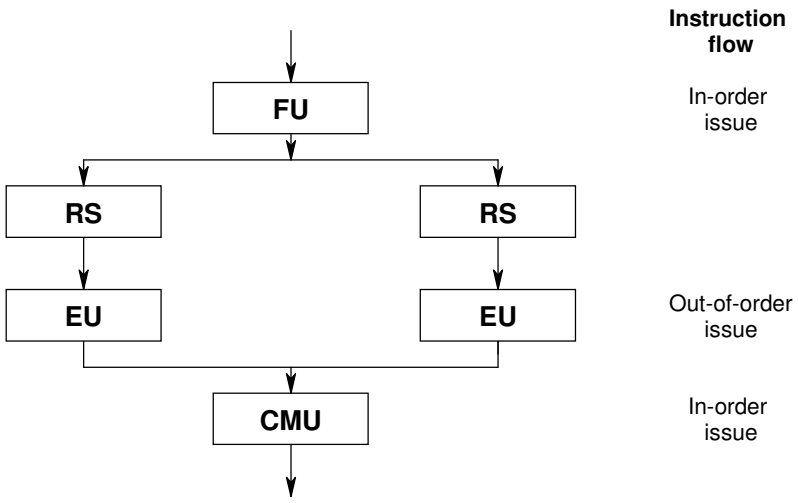
E. Out-of-order (OOO) Execution

A processor's instruction issue and completion policies are:

- **In-order issue** with **in-order completion**;
- **In-order issue** with **out-of-order completion**;
- **Out-of-order issue** with **out-of-order completion**;
- **Out-of-order issue** with **out-of-order completion** and **in-order commit**.

Out-of-order execution is a **situation in pipelined execution** when an **instruction blocked from executing does not cause the following instructions to wait**.

An **out-of-order architecture** takes code that was written and compiled to be executed in a specific order, reschedules the sequence of instructions so that they make maximum use of the processor resources, executes them, and then arranges them back in their original order so that results can be written out to memory.
An out-of-order-processors use schedulers, which scan a window of upcoming instructions for data dependencies. The scheduler determines which instructions must wait for results from others and which are ready to execute. The scheduler can take into account not just the data and instruction needs but also any execution resources.



Example

In-Order Instruction Issue with Out-Of-Order Instruction Completion

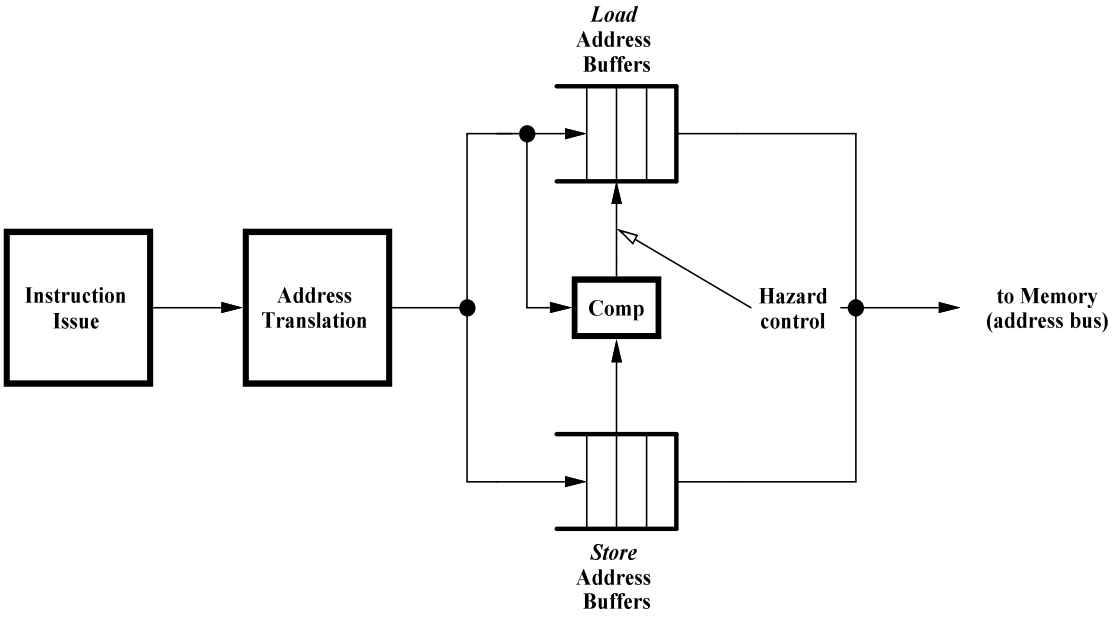
Instruction	Mnemonics	DR	SR2	SR1	Latency
I1	DIV	R6	R6	R4	4
I2	LD	R2	(R3)		1
I3	MULT	R0	R2	R4	3
I4	DIV	R8	R6	R2	4
I5	SUB	R10	R0	R6	1
I6	ADD	R6	R8	R2	1

Completion	Clock Cycle														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
In-order	I1	I2	—	—	**I1**	**I2**	I3	I4	—	**I3**	I5	**I4**	I6	**I5**	**I6**
Out-of-order	I1	I2	**I2**	I3	**I1**	I4	**I3**	I5	**I5**	**I4**	I6	**I6**			

Memory Hazards and OOO

Memory hazards can occur during store and load operations. Store (address) buffers are used to make sure memory operations don't violate hazard conditions. Store addresses are buffered in a FIFO queue. Store address buffers contain the addresses of all pending store operations.
Store addresses remain buffered until:

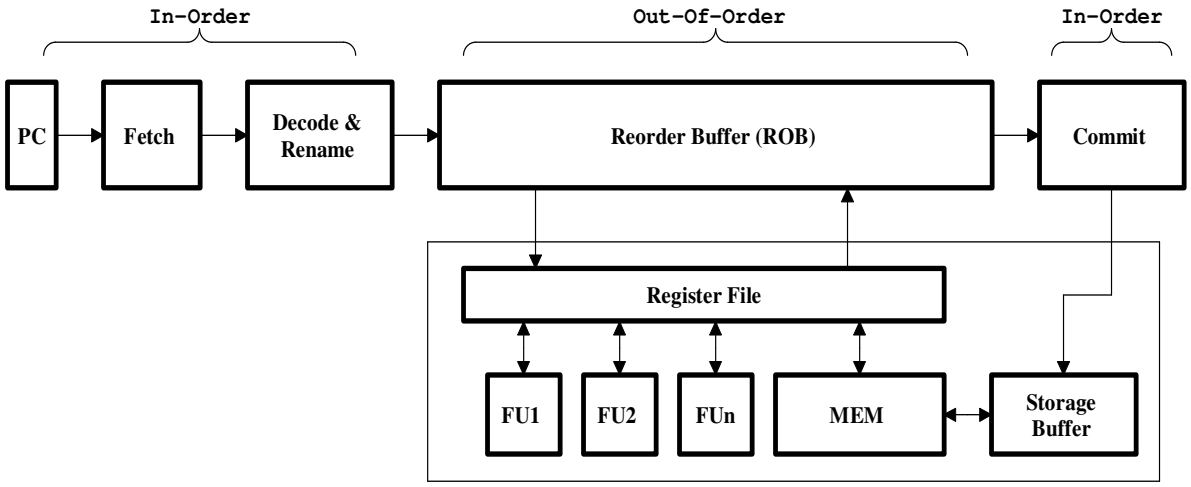
1. Their data is available;
2. The *Store* instruction is ready to be committed.



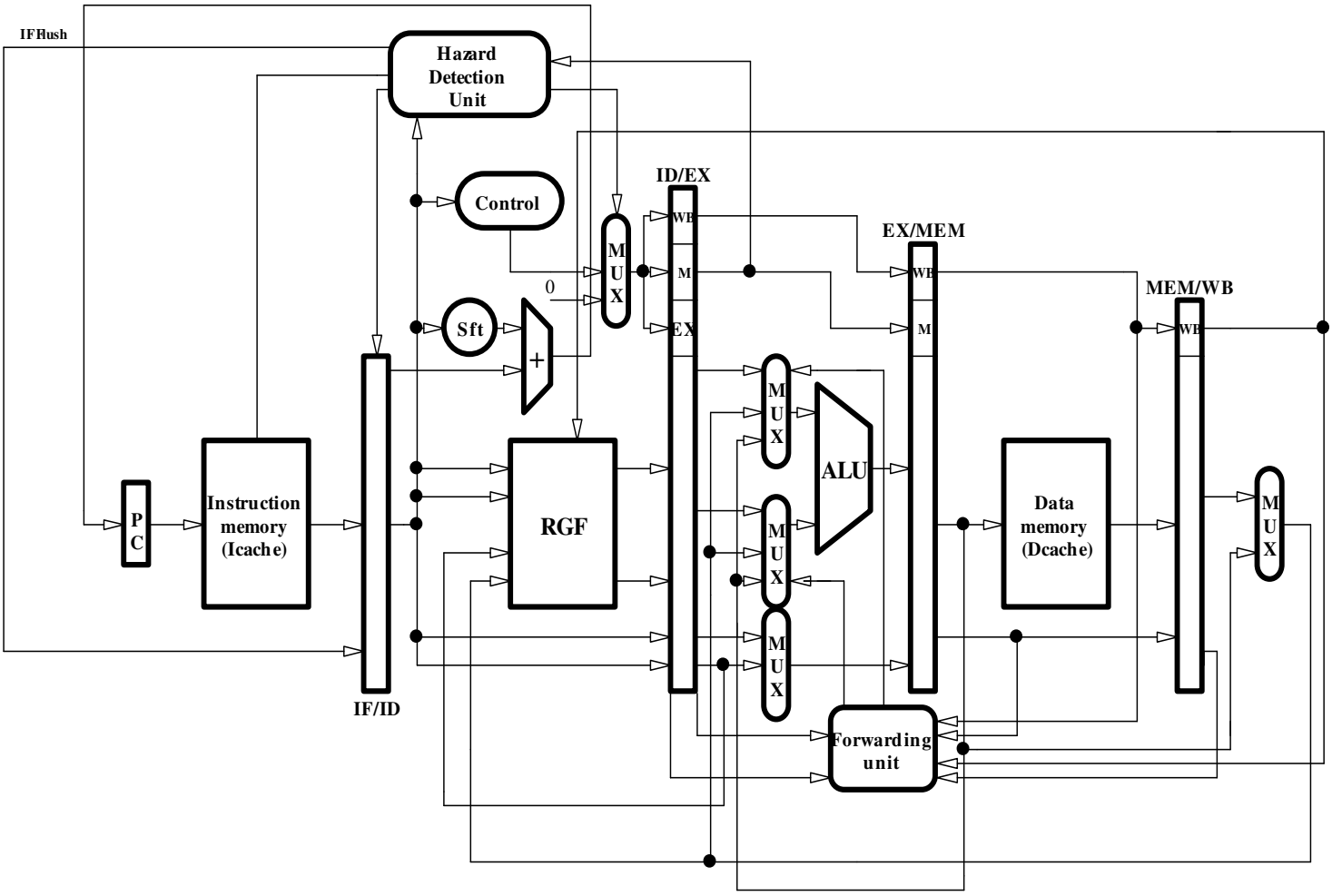
Branches in Oout-of-order Processors

To recover mispredict branches in in-order processors all instructions following a branch in pipeline are discarded. In out-of-order processors are used shadow registers and memory buffers for each speculative branch.

Superspeculative Out-Of-Order Superscalar Processor



Pipeline´s Simplified Data and Control Paths



IF/ID - pipeline register between the IF (instruction fetch) and ID (instruction decode and register file read) stages.
ID/EX - pipeline register between ID and EX (execution or address calculation) stages.
EX/MEM - pipeline register between EX and MEM (data memory stages).
MEM/WB - pipeline register between MEM and WB (write back) stages.

Pipeline Interrupts

Precise interrupt (exception) – it is required that that **the system state**, when the interrupt (exception) occurs, **is the same as that in a nonpipelined CPU that executes instructions in sequential order**.

In that case an **interrupt (exception**) occurring during the **execution of instruction Ij** is **precise** if the following conditions are met:

1. All instructions issued **prior to Ij** have completed their execution;
2. No instruction has been issued **after Ij**;
3. The program counter PC contains **Ij**'s address.

The most direct is to make all **interrupts (exceptions) precise** by forcing all instructions to complete in the order in which they are issued.

Methods for supporting speculation without introducing incorrect exception behavior:

1. **HW and OS ignore exceptions for speculative instructions**.
2. A set of bits (**poison bits**) **are attached to result registers** written by speculated instructions when the instructions cause exception.
3. HW support for speculation – **buffer results** from instructions until known that the instructions would execute (in **ROB**).

Interrupts (exceptions) in pipelined processor that are not associated with exact instruction that was the cause of the interrupt (exception) are called **imprecise interrupts** (**exceptions**).

When delayed branching is used, the instructions in the branch-delay slot are not sequentially related. If during execution an instruction in the branch-delay slot the interrupt is occurs, and the branch is taken, the instructions in the branch-delay slot and the branch-target instruction must be restarted after interrupt is processed.

In the case of OOO completion we must provide a mechanism to recover the precise state or context of the processor at the time of the interrupt.

> A small register set a history buffer (HB), is used to store temporarily the initial state of every register that is overwritten by each executing instruction Ij.
> If an interrupt occurs during Ij's execution, the corresponding precise CPU state can be recovered from the values stored in HB, even if a second conflicting interrupt is generated by a still-completing instruction.

Example

Ideal CPU Performance

Implementation	**CPU parameters** Perfect branch prediction with no Load/Store misses and stalls. **1.** Program has 1000 instructions with: **a.** 20% **B**ranch instructions; **b.** 10% **C**ontrol instructions; **c.** 20% **L**oad instructions; **d.** 10% Store instructions; **e.** 40% **A**rithmetic instructions. **2.** CPU clock frequency (F) is 1 GHz. $\mathbf{T_{CPU} = IC \times CPI/F}$
Non-pipelined [Every instruction takes 4 clock cycles to execute]	$\mathbf{T_{CPU} = (0,2\ CPI_B + 0,1\ CPI_C + 0,2\ CPI_L + 0,1\ CPI_S + + 0,4\ CPI_A) \times 1000 / 10^9}$ $T_{CPU} = (0,2 \times 4 + 0,1 \times 4 + 0,2 \times 4 + 0,1 \times 4 + 0,4 \times 4) \times 10^{-6} =$ $= \mathbf{4 \times 10^{-6}\ s}$
Pipelined **4-stage pipeline** **F, D, E, W** [In every clock cycle is retired only one instruction. Each instruction takes one clock cycle to execute.]	$\mathbf{T_{CPU} = (0,2\ CPI_B + 0,1\ CPI_C + 0,2\ CPI_L + 0,1\ CPI_S + + 0,4\ CPI_A) \times 1000 / 10^9}$ $T_{CPU} = (0,2 \times 1 + 0,1 \times 1 + 0,2 \times 1 + 0,1 \times 1 + 0,4 \times 1) \times 10^{-6} =$ $= \mathbf{1 \times 10^{-6}\ s}$
Pipelined **3-way superscalar** [In a given clock cycle we can retire one of the following instruction groups: **a. 3 A**rithmetic instructions; **b. 3 B**ranch instructions; **c. 2 C**ontrol instructions; **d. 1 L**oad instruction; **e. 1 S**tore instruction.]	$\mathbf{T_{CPU} = (0,2\ CPI_B + 0,1\ CPI_C + 0,2\ CPI_L + 0,1\ CPI_S + + 0,4\ CPI_A) \times 1000 / 10^9}$ $T_{CPU} = (0,2 \times (1/3) + 0,1 \times (1/2) + 0,2 \times (1/1) + 0,1 \times (1/1) +$ $+ 0,4 \times (1/3)) \times 10^{-6} \approx \mathbf{0,5 \times 10^{-6}\ s}$

Type	Number of pipelines	Issue order	Instructions per cycle	Scheduling
Simple scalar	**1**	in - order	**1**	static
Scalar	**>1**	in – order	**1**	static
Superscalar in-order	**>1**	in - order	**>1**	dynamic
VLIW	**>1**	in – order	**>1**	static
Superscalar out-of-order	**>1**	out-of-order	**>1**	dynamic

Comment

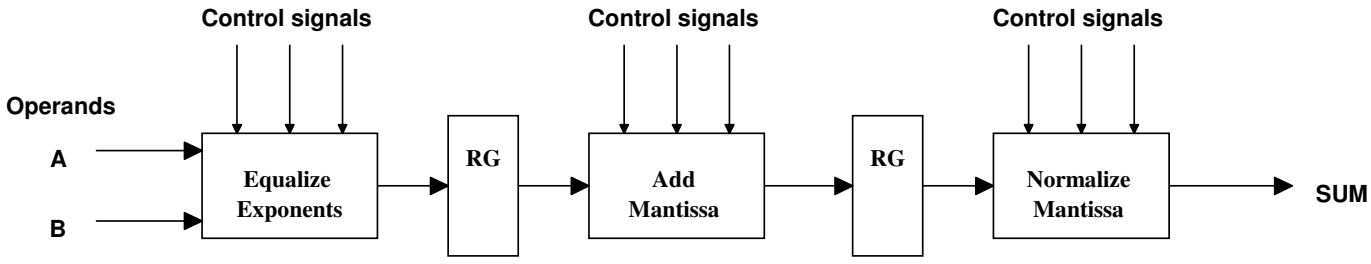
Timing Anomalies

Most powerful microprocessors suffer from *timing anomalies*.
Timing anomalies are contraintuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole task.

If the processor speculates on the outcome of conditional branches, it prefetches instructions in one of the directions of the conditional branch. When the condition is finally evaluated it may turn out that the processor speculated in the wrong direction. All the effects produced so far have to be undone.
In addition, fetching the wrong instructions has partly ruined the cache contents.
The local worst case, the I-cache miss, leads to the globally shorter execution time since it prevents a more expensive branch misprediction. This exemplifies one of the reasons for timing anomalies, *speculation-caused anomalies*.
Another type of timing anomalies are *scheduling anomalies*.
These occur when a sequence of instructions, partly depending on each other, can be scheduled differently on the hardware resources, such as pipeline units. Depending on the selected schedule, the execution of the instructions or pipeline phases takes different times.

Appendix

Operational Unit (Arithmetic) Pipeline
Floating-point ADD operation pipeline



Buffer registers (RG) needed when the processing times for pipeline stages are not equal. Some flag bits indicate the completion of processing at the output of each stage may be needed. In addition to the pipelining within the functional units, it is possible to pipeline arithmetic operations between the functional units. This is called *chaining //aheldus//*.

SUPERSCALAR PROCESSORS FUNDAMENTALS

Basic Evolution Phases of Processors

I	II	III
Traditional von Neumann processor	**Scalar ILP processor**	**Superscalar ILP processor**
Instructions sequential issue Sequential execution	***Instructions sequential issue Parallel execution***	***Instructions parallel issue Parallel execution***
[Nonpipelined processors]	[Processors with multiple nonpipelined execution units or pipelined processors]	[VLIW and superscalar processors: processors embodying multiple pipelined execution units]

Parallelism of instruction issue
Parallelism of instruction execution
Processor performance

Generally, CPU's **absolute performance P_{CPU}** can be expressed as follows:

$$P_{CPU} = f \times IPC, \text{ where}$$

f – the clock frequency,
IPC – instructions per clock cycle (per cycle throughput)

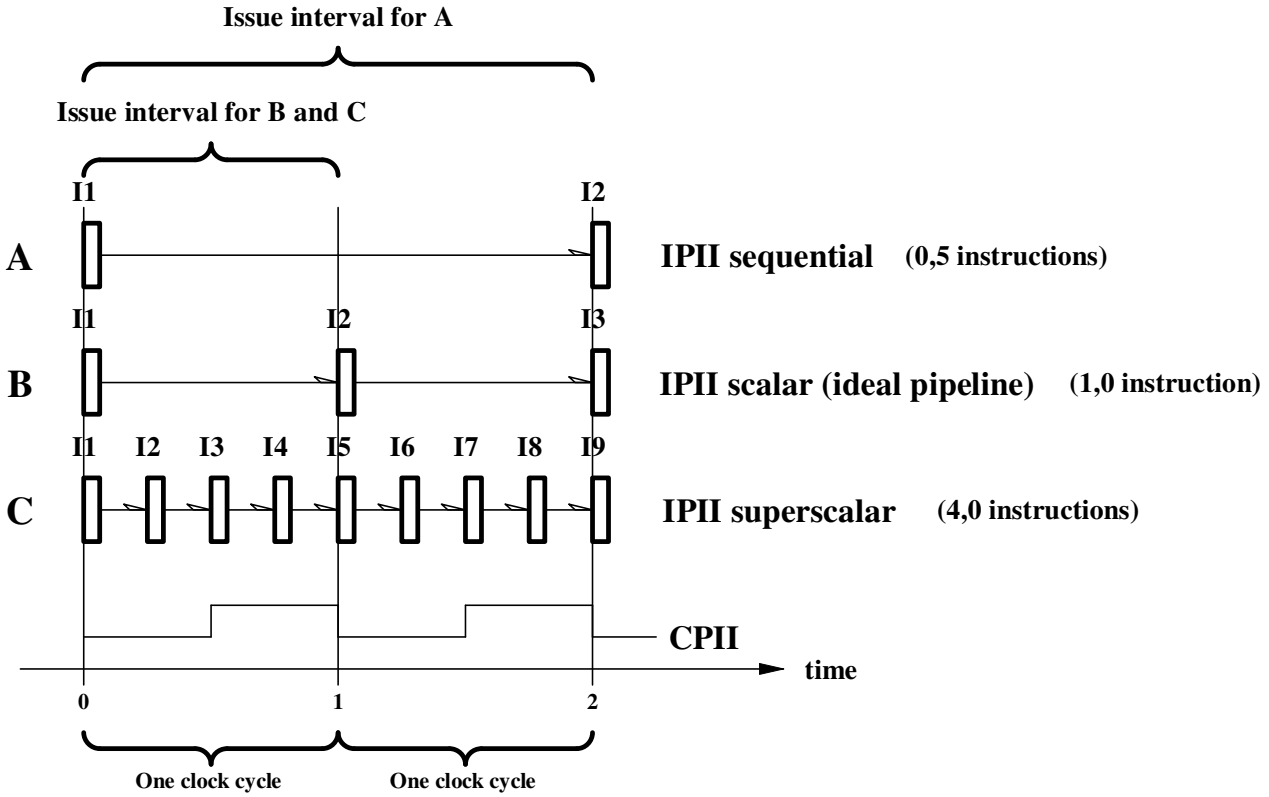
IPC can be expressed by two **CPU's internal operational parameters**:

$$IPC = IPII / CPII = 1/CPII \times IPII, \text{ where}$$

IPII – instructions per **issue interval**
CPII – cycles per issue interval

Issue intervals are subsequent segments of the execution time of a program such that each issue interval begins at a clock cycle when the processor issues at least one instruction and ends when the next issue interval begins.
In sequential processors CPII equals to the average execution time of the instructions, i. e., CPII >> 1.
The CPII for pipelined or superscalar processors is CPII = 1, and for superpipelined processors is CPII < 1.

CPII reflects the temporal parallelism of instruction processing.



For **scalar processors IPII = 1**, and for **superscalar processors**

$$1 < IPII < n_I,$$ where

n_I – the issue rate of processor.

IPII reflects the **issue parallelism**.

	Scalar processor	**Superscalar processor**	**Superpipelined processor**
CPII (temporal parallelism)	**>1**	**1**	**<1**
IPII (issue parallelism))	**1**	**>1**	**>1**

The benefit of using IPII and CPII instead of IPC or CPI is being able to identify two sources of parallelism separately.
While taking account the average number of data operations proceeds per cycle (DOPC) we get:

DOPC = IPC × OPI, where
OPI – operations per instruction (average).
OPI reveals **intrainstruction parallelism.**

Practically **OPI<1**.

For **instruction sets** including **multioperation instructions** the **OPI > 1**.

$$\mathbf{OPC = 1 / CPII \times IPII \times OPI}$$
$$\mathbf{P_{CPU} = f \times (1/CPII) \times IPII \times OPI}$$

The key possibilities for boosting processor performance in the processor microarchitecture:

1. Increasing the clock frequency;
2. Increasing the temporal parallelism;
3. Increasing the issue parallelism;
4. Increasing intrainstructions parallelism.

Increasing Instruction Executing Rate

There are two fundamental ways of increasing a processor's instruction executing rate:

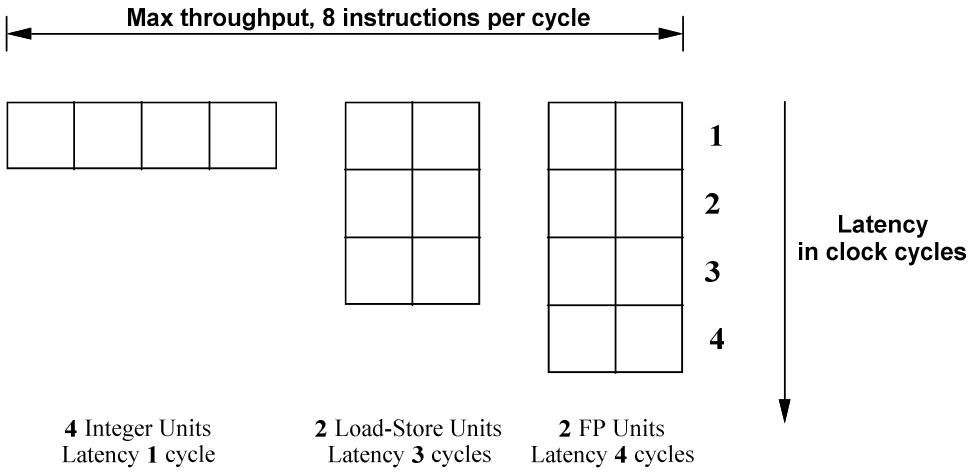
1. By increasing clock speed.
   This approach relies on improvements in IC *technology*.
2. By increasing the number of instructions that are executed simultaneously.
   This approach relies on improvements in processor's *architecture*.

Both factors, increasing issue width and pipeline depth, place increasing pressure on the instruction fetch unit. Increasing the superscalar issue width requires a corresponding increase in the instruction fetch bandwidth.

Recall Little's Law

Parallelism (P) = Throughput (T) × Latency (L)

Pipelined ILP Machine



Question: *How much instruction-level parallelism (ILP) required to keep processor pipelines busy?*

T = 8; $\mathbf{L_{tot}} = [(4 \times 1) + (2 \times 3) + (2 \times 4)] / 8 =$ **9/4**; **P** $= 8 \times 9/4 =$ **18 instructions**

Contrary to pipeline techniques, ILP is based on the idea of **multiple issue processor** (**MIP**). An MIP has multiple pipelined datapaths for instruction executing.

Resource Replication and Widening

In order to increase the number of operations performed per cycle the resources of the processor must be increased. There are used two alternatives – **resource replication** //*replikatsioon e dubleerimine*// and **widening** //*laiendamine*//.

Resource replication consists of increase the number of resources available by adding more independent functional units.
Resource widening consists of increasing the number of operations that each functional unit can simultaneously perform per cycle.

The replication is more versatile than the widening technique.

- Applying a replication degree of n means that the processor can access n independent words in memory or perform n independent operations per clock cycle.
- Applying the same degree of widening requires a study at compile time to detect compactable operations.

On the other hand, the replication technique has, in general, higher costs than the widening technique.

Buses

- Widening increases the width of the data bus, but not the control and address bus.
- Replication increases the number of buses between the register file and the L1 cache.
- Widening requires no additional address translations, while replication requires several address translated per cycle.

Register File

- Applying replication increases the number of ports per bit.
- Both techniques increase the register file area on chip and clock cycle time.

Code size

- Widening can reduce the total number of instructions.
- The reduction of the code size can reduce the miss rate of the instruction cache and improve performance.

For a given technology, the best performance is obtained when combining a certain degree of replication and widening in the hardware resources.

It is more effective than applying only the replication of resources.

Instruction Fetch

In the instruction fetch stage s instructions are getting out of the instruction cache during each clock cycle. These instructions are called *fetch group //käsuvõtu grupp//.*
Let p be that probability that instruction is a taken branch or unconditional branch (jump).
Let assumes that the fetch stage can predict whether the branch is taken.
If the first instruction in the fetch group is a taken branch or unconditional jump, then the rest of the instructions in the fetch group should not be executed.
The probability of P_1 that an only one executable instruction (the branch or jump) is fetched:

$$\mathbf{P_1 = p}$$

The probability of fetching two executable instructions $\mathbf{P_2}$ is:

$$\mathbf{P_2 = (1\text{-}p) \times p}$$

The last instruction in the fetch group is executable whether it is a taken branch or unconditional branch

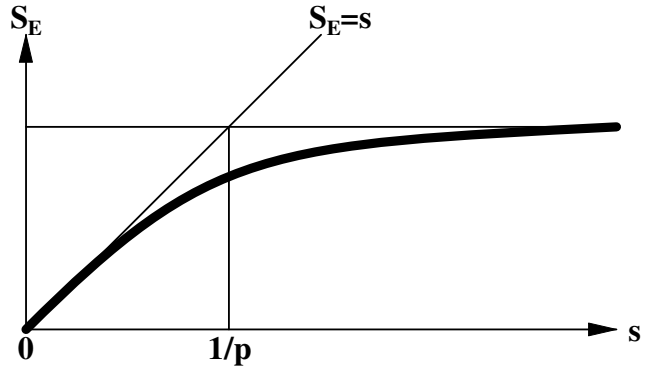
$$\mathbf{P_S = (1\text{-}p)^{s-1}}$$

The average number of fetched instructions that are executable S_E is:

$$\mathbf{S_E = 1 \times P_1 + 2 \times P_2 + \ldots + (s\text{-}1) \times P_{s-1} + s \times P_s} \quad \text{or}$$

$$\mathbf{S_E} = \frac{1-(1-p)^s}{p}$$

When the fetch group is small (s<1/p), almost all of the instructions are executable (S_E = s).
For large fetch groups (s>1/p), the average number of executable instructions can never be any large than (1/p), i.e., $S_E < 1/p$.
General purpose instruction streams typically have at least 10% taken branch and unconditional branch instructions (p=0,1).



The limit on the number of executable instructions in fetch group would seem to be a serious limitation for large superscalar processors.

Instruction Window

Instruction window – the set of instructions that is examined for simultaneous executing.
Since each instruction in the window must be kept in processor and the number of comparisons required to execute any instruction in the window grows quadratically in the window size, real window size are likely to be small.
The number of comparisons required every clock cycle is equal to:

(maximum instruction completion rate) × (window size) × (number of operands per instruction)

The window size limits the maximum number of instructions that may issue.
The notion of the instruction window comprises all the waiting stations between the decode (rename) and execute states (FUs).

The instruction window isolate the decode/rename stage from the execution stages of the instruction pipeline. The decode stage continues to decode instructions regardless of whether they can be executed or not. The decode stage places the decoded instructions in the instruction window as long as there is room in the window.

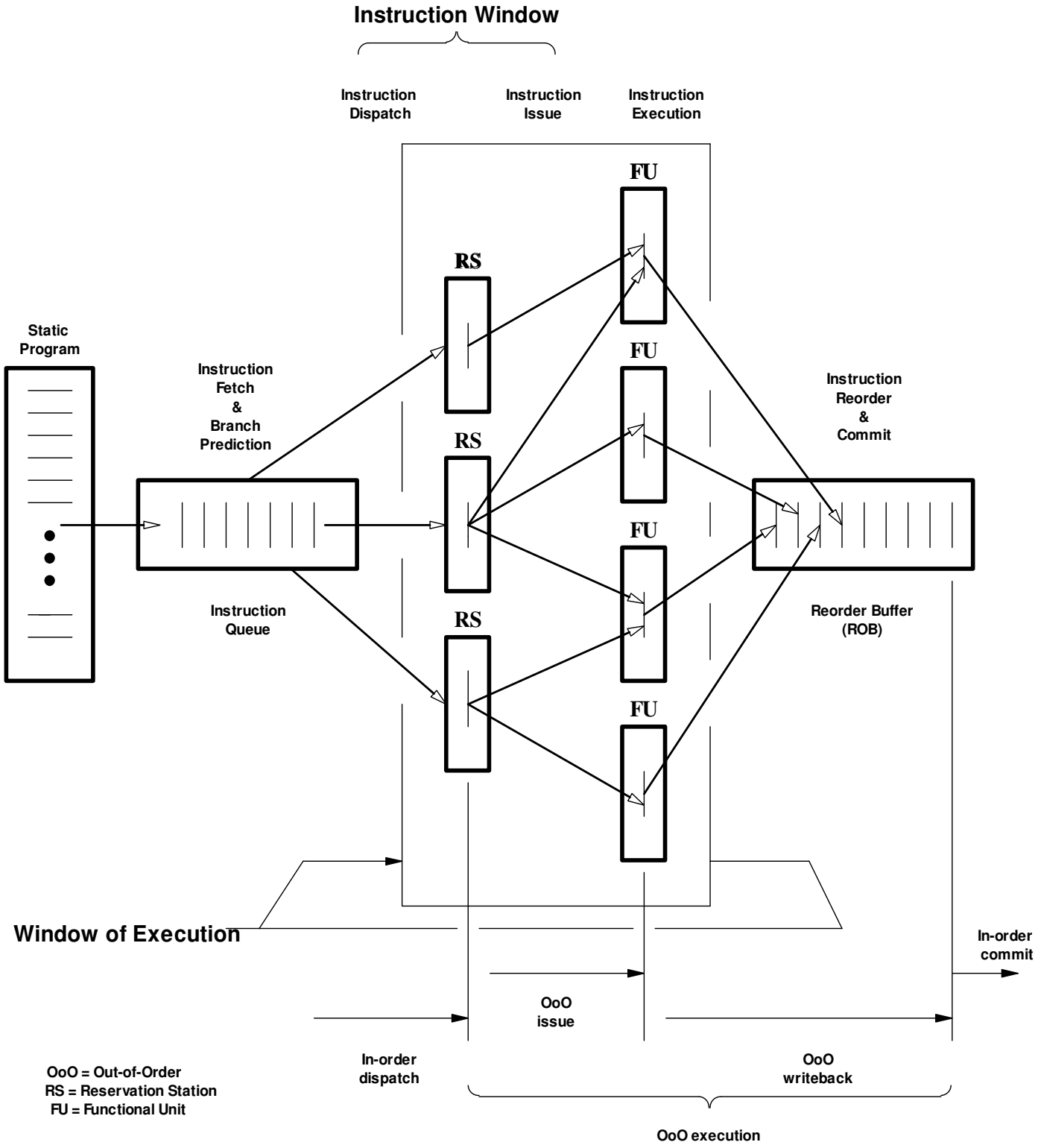
The instructions in the instruction window are free from control dependences, which are removed by branch prediction and free from antidependences or output dependences, which are removed by renaming. Only data dependences (RAW) and resource dependences remain to be taken into consideration.

Instruction dispatch – assigning an execution unit to an instruction.
Instruction issue – beginning execution of an instruction on an execution unit.

Aligned issue – until all the **n** instructions have not completed their execution, the following group is not decoded/issued.

Alignment-free issue – it is possible to decode/issue instructions following the group without completing the execution of the **n** instructions in the current group.

Superscalar Processor's General Model



The total window size is limited by the required storage, the comparison, and a limited issue rate, which means large windows less helpful.

The instruction window size (WS) directly limits the number of instructions that begin execution in a given cycle.

In practice, real processors will have a more limited number of FUs, as well as limited number of buses, and register access ports, which serve as limits on the number of instructions initiated per clock. The maximum numbers of instructions that may issue begin execution, or commit in the same clock cycle is usually much smaller than the window size.

Instruction Shelving

Shelving //*käsuladustus*// decouples instruction issue and data dependency checking. This technique presumes special instruction buffers in front of the execution units.
With shelving, instructions are issued first to the RS with essentially no need for dependency checks. Shelving delays dependency checking to later step of processing. During dispatching processor checks the instructions held in the RSs for data dependencies, and forwards dependency-free instructions to available execution units.

Deep Pipelines in Superscalar Processors

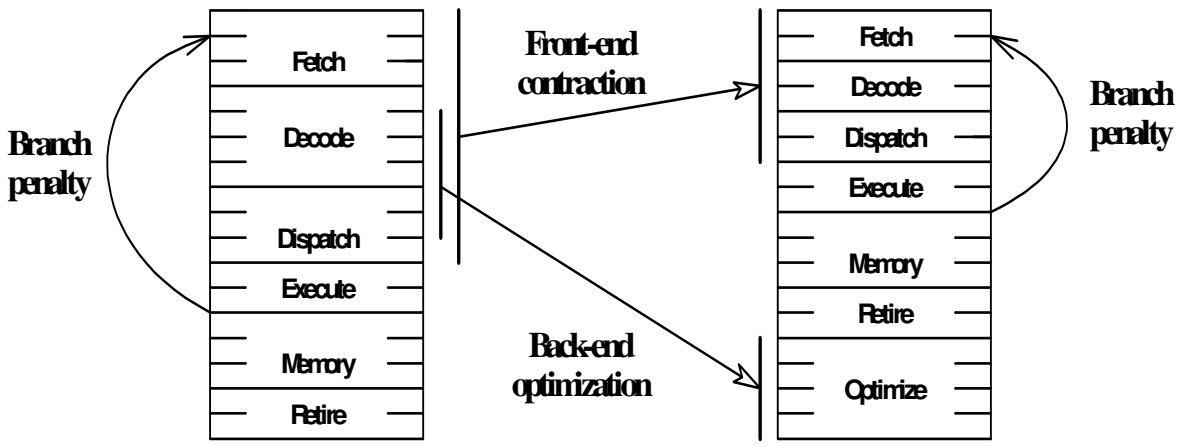
A deeper pipeline increases the number of pipeline stages and reduces the number of logic gate levels in each pipeline stage. The benefit of deeper pipelines is the ability to reduce the machine cycle time and hence increase the clocking frequency.

As pipelines get wider, there is increased complexity in each pipeline stage, which increases the delay of each pipeline stage. To maintain the same clocking frequency, a wider pipeline will need to be made even deeper.
With a deeper pipeline the penalties incurred for pipeline hazard resolution can become larger.

To ensure overall performance improvement with deeper pipeline, the increase in clocking frequency must exceed the increase in CPI.

There are two approaches that can be used to mitigate the negative impact of their increased branch penalty in deep pipelines:

1. With a mispredicted branch, all the instructions in the front-end pipeline stages must be flushed.
2. The second approach is to move some of the front-end complexity to the back end of the pipeline => back-end optimizations (see figure and table).



n=17 stages
BP = 12 stages

n=17 stages
BP = 8 stages

There is no longer the centralized control performed by the control path. Instead, a form of distributed control via propagation of the control signals through the pipeline stages is used.

Not only the data path is pipelined, but also the control path is pipelined.

The traditional data path and the control path are integrated into the same pipeline.

Optimizations Types

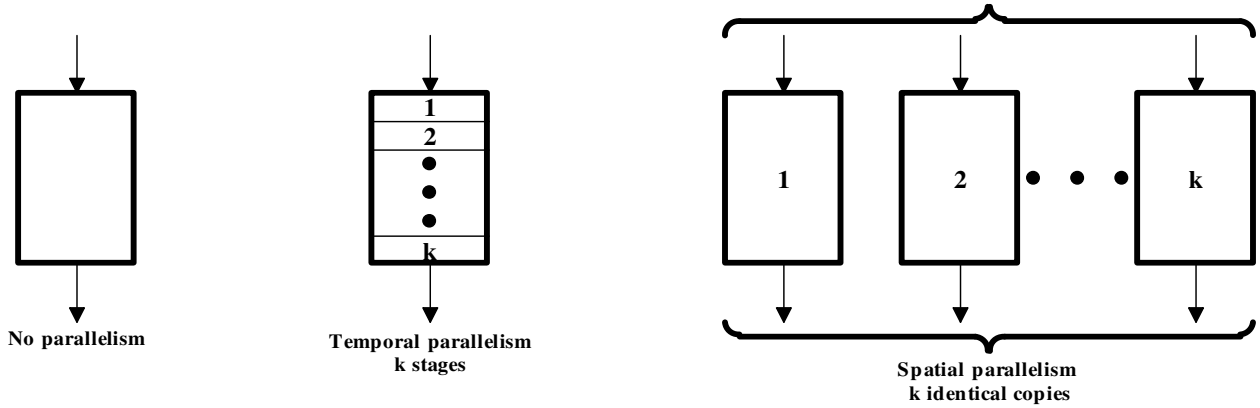
Optimization type	Explanation	Percentage of the total number of optimizing transforms
High-level Procedure integration	**At the source level: processor independent** Replace procedure call by procedure body	
Local Common sub-expression elimination Constant propagation Stack height reduction	**Within straight-line code*** Replace two instances of the same computation by single copy Replace all instances of a variable that is assigned a constant with the constant Rearrange expression tree to minimize recourses needed for expression evaluation	**18%** **22%**
Global Global common sub-expression elimination Copy propagation Code motion Induction variable elimination	**Across a branch** Same as local, but this version crosses branches Replace all instances of a variable A that has been assigned X (i.e. A = X) with X Remove code from a loop that computes same value each iteration of the loop Simplify/eliminate array addressing calculations within loops	**13%** **11%** **16%** **2%**
Processor-dependent Strength reduction Pipeline scheduling Branch offset optimization	**Depends on processor knowledge** Many examples, such as replace multiply by a constant with adds and shifts Reorder instructions to improve pipeline performance Choose the shortest branch displacement that reaches target	

- **Straight-line coding** //*sirg- e lineaarprogramm*// - a program (basic block) that is written to avoid the use of loops and branches, providing a faster execution time.

Superscalar Pipeline Models

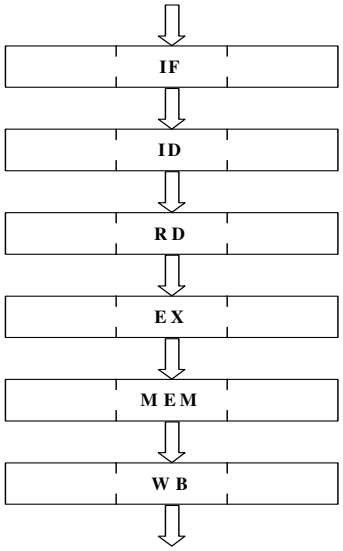
- **Superscalar pipelines** are **parallel pipelines** [**A**], in being able initiate the processing of multiple instructions in every machine cycle.
- **Superscalar pipelines** are **diversified pipelines** [**B**] employing **multiple** and **heterogeneous functional units in their execution stage**.
- **Superscalar pipelines** can be implemented as **dynamic pipelines** [**C**] in order to achieve the best possible performance without requiring reordering of instructions by the compiler.

A. Parallel Pipeline (Superscalar Pipeline)



The degree of parallelism of a machine is measured by the maximum number of instructions that can be concurrently in process at one time.
Temporal parallelism via pipelining requires less hardware than spatial parallelism, which requires replication of the entire processing unit.

An example of the 6-stage parallel pipeline of width wp=3

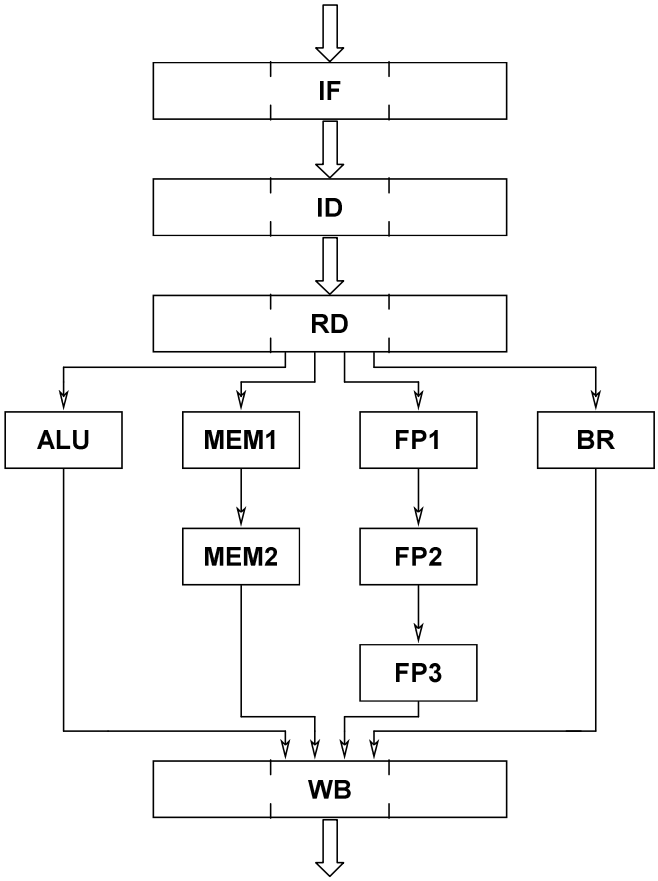


Parallel pipeline is a combination of pipelining and parallel processing.
For parallel pipelines or superscalar pipelines the speedup is measured with respect to a scalar pipeline and is primarily determined by the **width** (**wp)** of the parallel pipeline.
A parallel pipeline with width **wp** can concurrently process up to **wp** instructions in each of its pipeline stages.

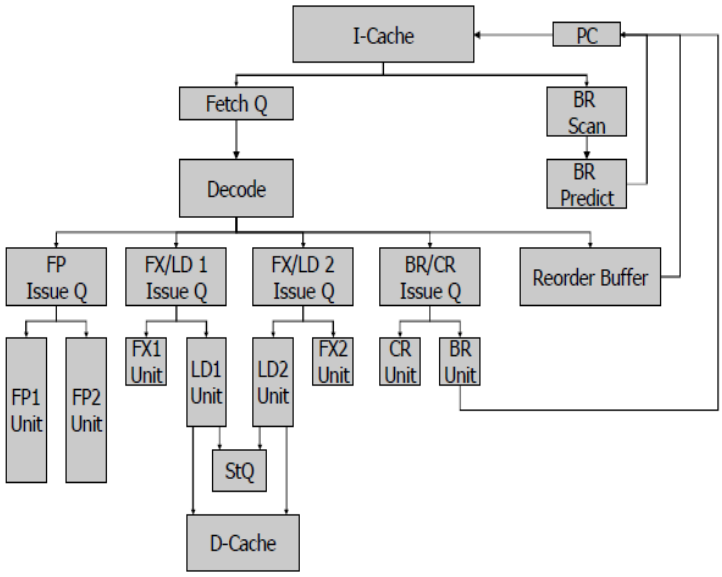
B. Diversified Pipeline

Diversified pipelines are parallel pipelines that have been specialized to process different instructions types.
In parallel pipelines multiple different functional units (FU) are used in the execution portion stage. Instead of implementing p identical pipes in an p-wide parallel pipeline, in the execution portion of the parallel pipeline, diversified execution pipes can be implemented.

Diversified Parallel Pipeline (wp=3) with 4 Execution Units (Pipes)



Power4 Diversified Pipelines



The advantages in implementing diversified execution pipelines:

1. Each pipe can be customized for a particular instruction type;

2. Each instruction type incurs only the necessary latency and makes use of all stages of an execution pipe;
3. If all instruction dependencies between different instruction types are resolved prior to dispatching, the once instructions are issued into the individual execution pipes, no further stalling can occur due to instructions in other pipes.

The idea of diversified pipelines is not new. The supercomputer *CDC 6600* (1965) had 10 parallel diversified execution pipelines.

C. Dynamic Pipeline

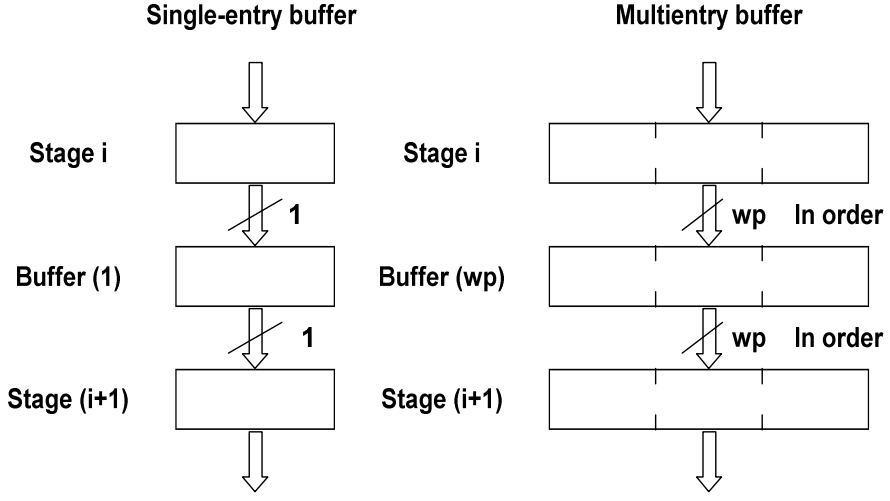
In any pipelined design, buffers are required between pipeline stages. The buffers hold all essential control and data bits for the instruction that has just traversed stage i of the pipeline and is ready to traverse stage (i+1) in the next machine cycle. Single-entry buffers are quite easy to control.
In every machine cycle, the buffer's current content is used as input to stage (i+1), and at the end of cycle, the buffer latches in the result produced by stage i. The buffer clocked each machine cycle. The exception occurs when the instruction in the buffer must be held back and prevented from traversing stage (i+1). In that case, the clocking on the buffer is disabled, and the instruction is stalled in the buffer.
Instructions dynamic scheduling – the hardware rearranges the instruction execution to reduce the pipeline stalls. Dynamic scheduling offers several advantages:

1. It enables handling some cases when dependencies are unknown at compile time;
2. It simplifies the compiler.

These advantages are gained at a cost of a significant increase in hardware complexity.
In introducing out-of-order execution, it must be split the instruction decode pipeline stage into two stages:

1. Issue – decode instructions, check for structural hazards;
2. Read operands – wait until no data hazards, and then read operands.

In parallel pipeline multiple buffers are needed between two consecutive pipeline stages.
Multiple instructions can be latched into each multientry buffer in every machine cycle.
In the next cycle, these instructions can traverse the next pipeline stage.



If all of the instructions are in a multientry buffer are required to advance simultaneously in a lockstep fashion, then the control of the multientry buffer is similar to that of the single-entry buffer. Each entry of the simple multientry buffer is hardwired to one write port and one read port of this buffer.
The entire multientry buffer is either clocked or stalled in each machine cycle. Such operation of the parallel pipeline may induce unnecessary stalling of some of the instructions in a multientry buffer. Simple multientry buffer enhancements could be:

- Adding special connectivity between entries to facilitate movement of data between entries.
- Providing a mechanism for independent accessing of each entry in the buffer.

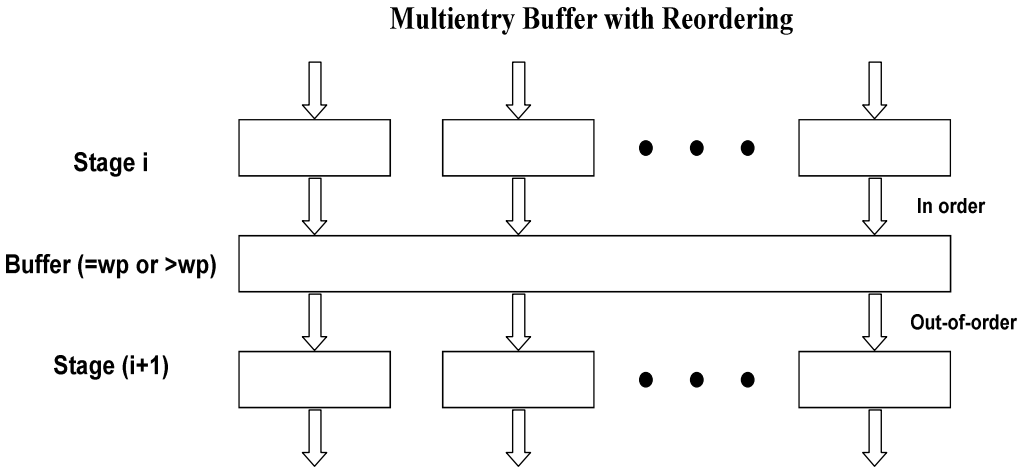
Superscalar pipelines differ from scalar pipelines in one key aspect, which is the use of complex multientry buffers for buffering instructions in flight.

In order to minimize unnecessary stalling of instructions in parallel pipeline, trailing instructions must be allowed to bypass stalled leading instructions.

Such bypassing can change the order of execution of instructions from the original sequential order of the static code. With out-of-order execution of instructions, there is the potential of approaching the data flow limit of instruction execution.

*A **parallel pipeline** that **supports out-of-order execution of instructions** is called a **dynamic pipeline**.*

A dynamic pipeline achieves out-of-order execution via the use of complex multientry buffers that allow instructions to enter and leave the buffer in different orders.

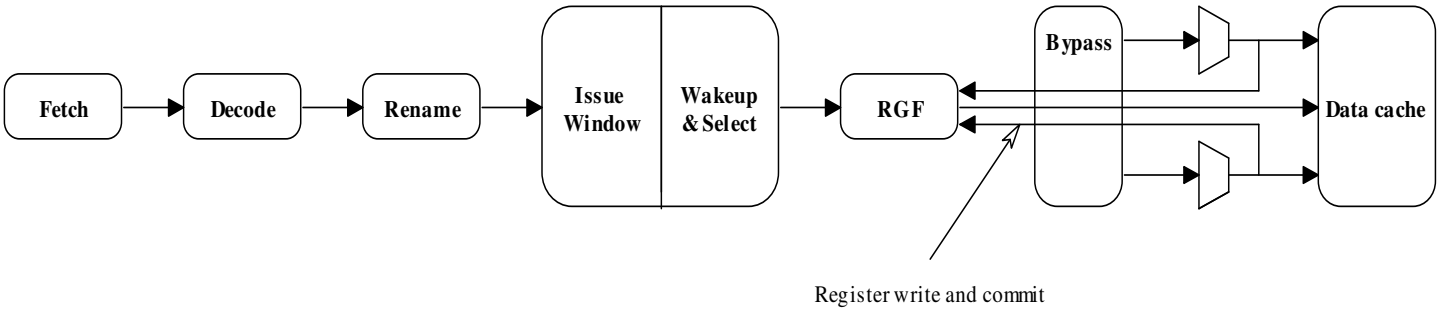


The execution portion of the dynamic diversified pipelines is usually bracketed by two reordering multientry buffers:

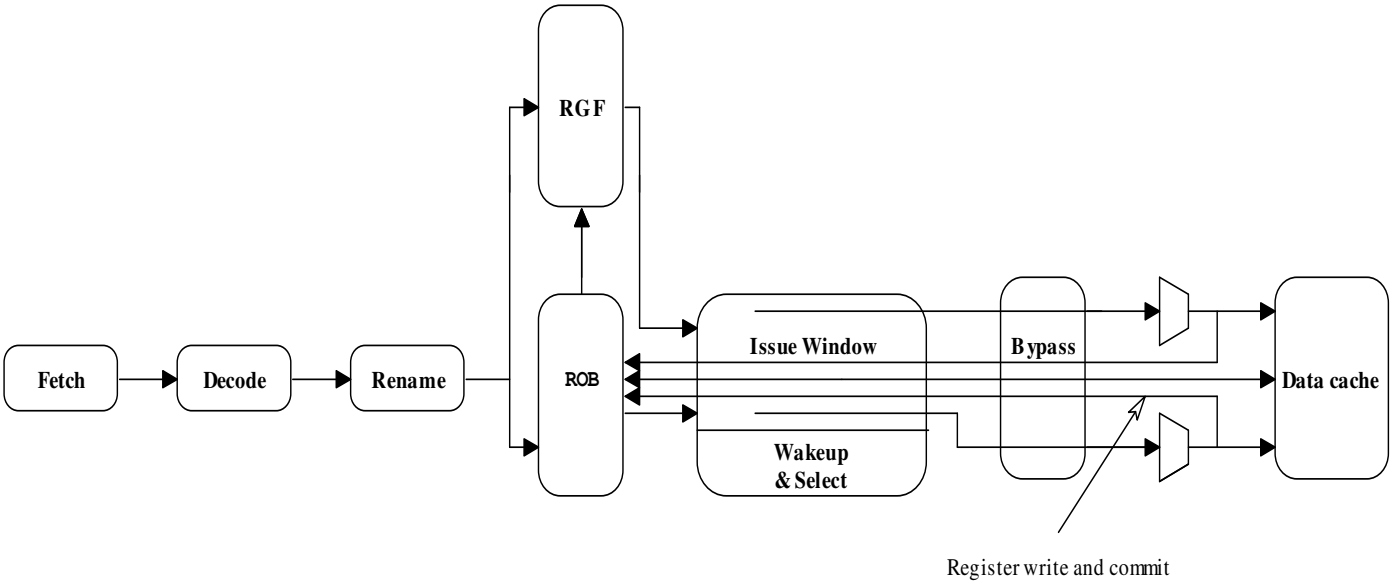
1. The first buffer is the dispatch buffer.
   It is loaded with decoded instructions according to program order and then dispatches instructions to the functional units potentially in an order different from the program order.

2. The second buffer is the completion buffer (reorder buffer - ROB).
   It buffers the instructions that may have finished execution out of order and retires the instructions in order by outputting instructions to the final write-back stage in program order.
   The operation of the ROB is as follows:

1. When an instruction is decoded, it is allocated an entry at the top of the ROB so that result value of this instruction can be written into the allocated entry it completes.
2. When the value reaches the bottom of the ROB, it is written into the register file.
   If the instruction is not complete when its entry reaches the bottom of the ROB, the ROB does not advance until the instruction completes.



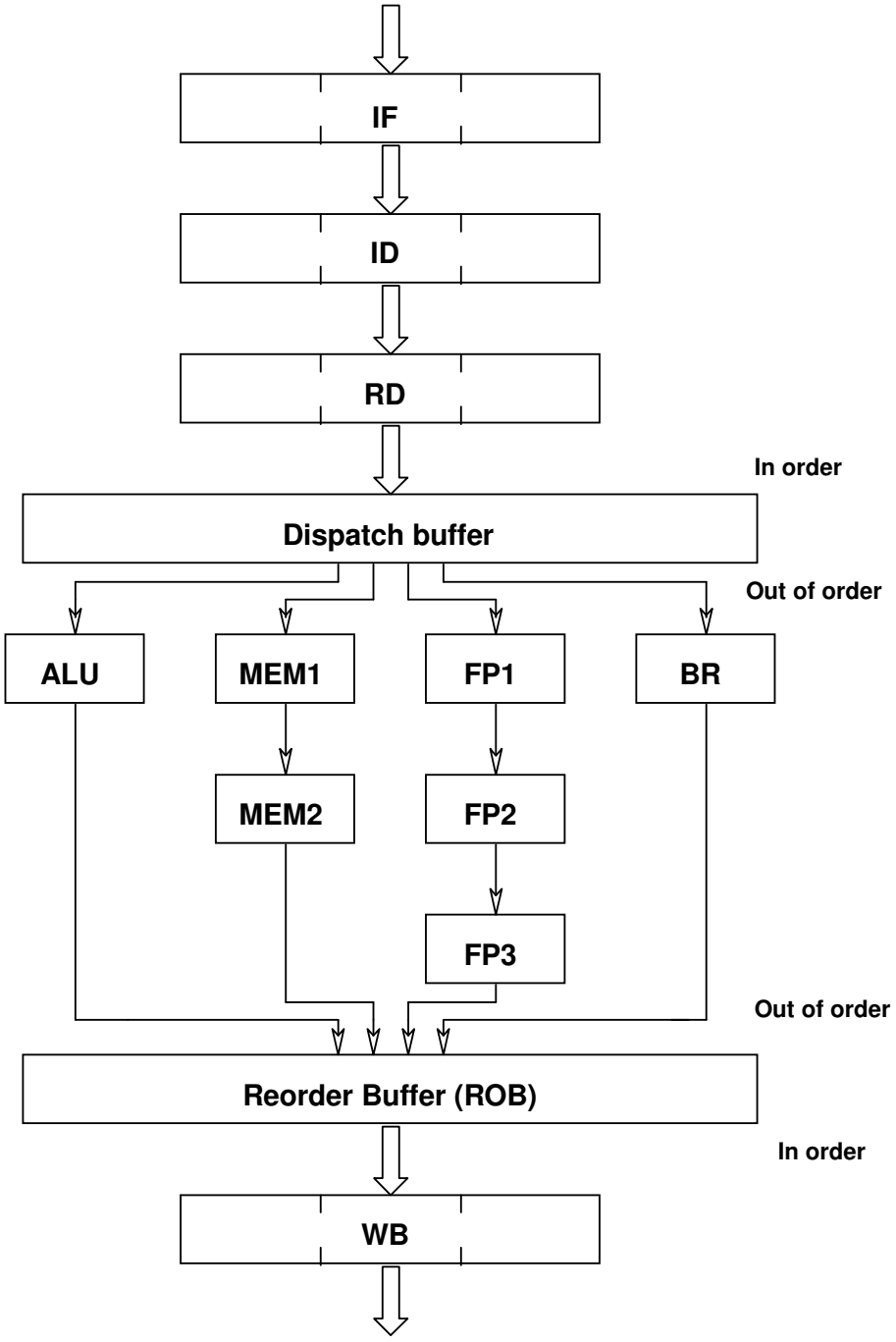
Superscalar processor model without ROB



Superscalar processor model with ROB

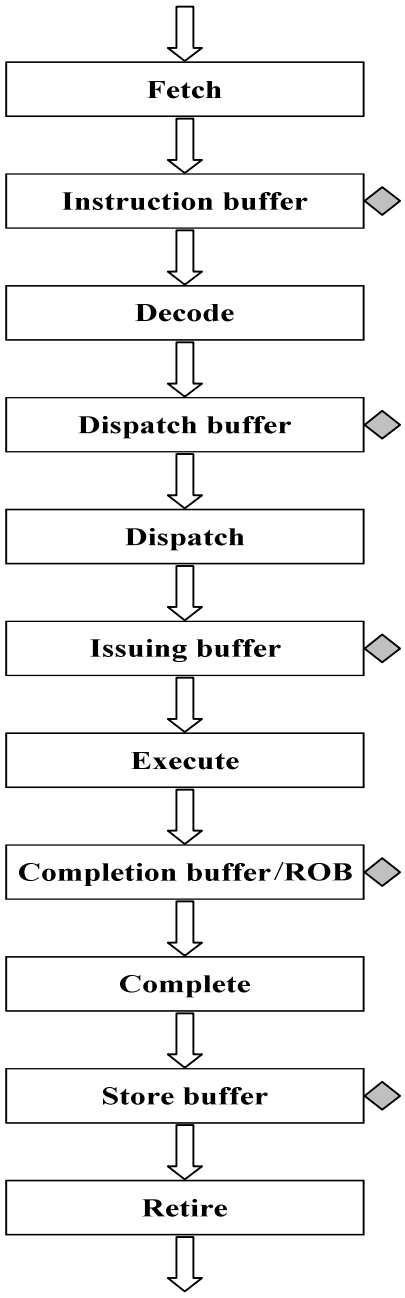
Example

The 5-stage dynamic pipeline (wp=3) which has 4 execution pipes in execution stages which are bracketed by two reordering multientry buffers (Dispatch Buffer and Reorder Buffer)



Example 2

The typical (template for) superscalar 6-stage pipeline



There are **multientry buffers** (◇) separating the six pipeline's stages.
The complexity of these buffers can vary depending on their functionality and location in the superscalar pipeline.

Dynamic Instruction Scheduling in Superscalar Processors

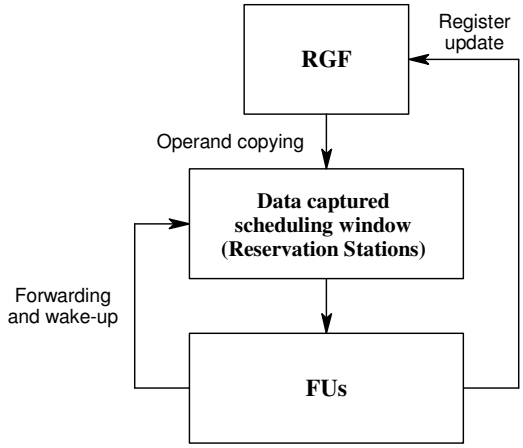
The most straightforward way to improve the IPC of a modern superscalar processor is to increase:

1. The instruction issue width (IW)
Not all types of instructions can be issued together.
2. The instruction window size (WS), [WS ≥ IW].
Restrictions

a. The IW is restricted mainly by the clock speed.
b. The WS is restricted by the number of permitted transistors.
c. The delays of almost all components of a superscalar processor except execution units are increasing functions of IW and WS.

Dynamic instruction schedulers can be:

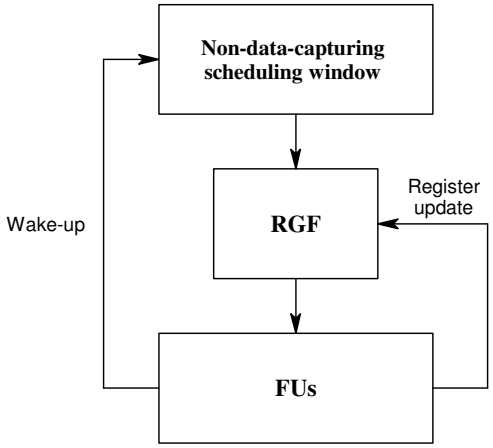
1. With data capture;
2. Without data capture.



Instruction scheduler with data capture

When instruction is dispatched, the operands that are ready are copied from the RGF into the instruction window.
For the operands that are not ready, tags are copied into the instruction window and used to latch in the operands when they are forwarded by the FUs.

Result forwarding and **instruction wake up** are **combined**.

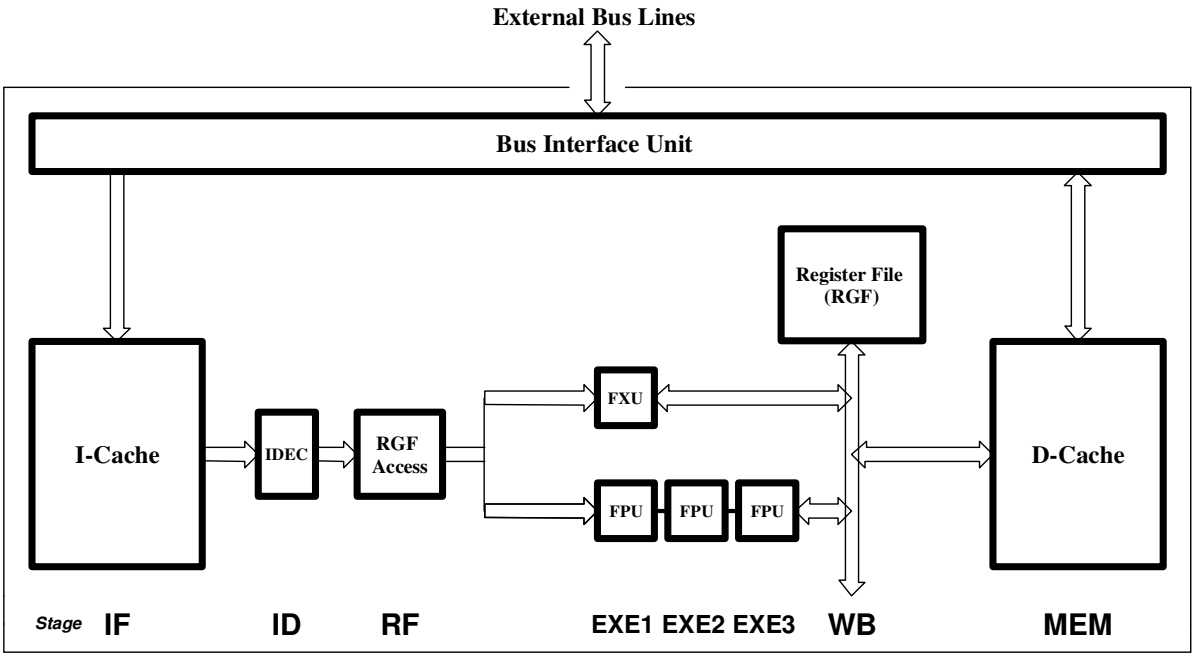


Instruction scheduler without data capture

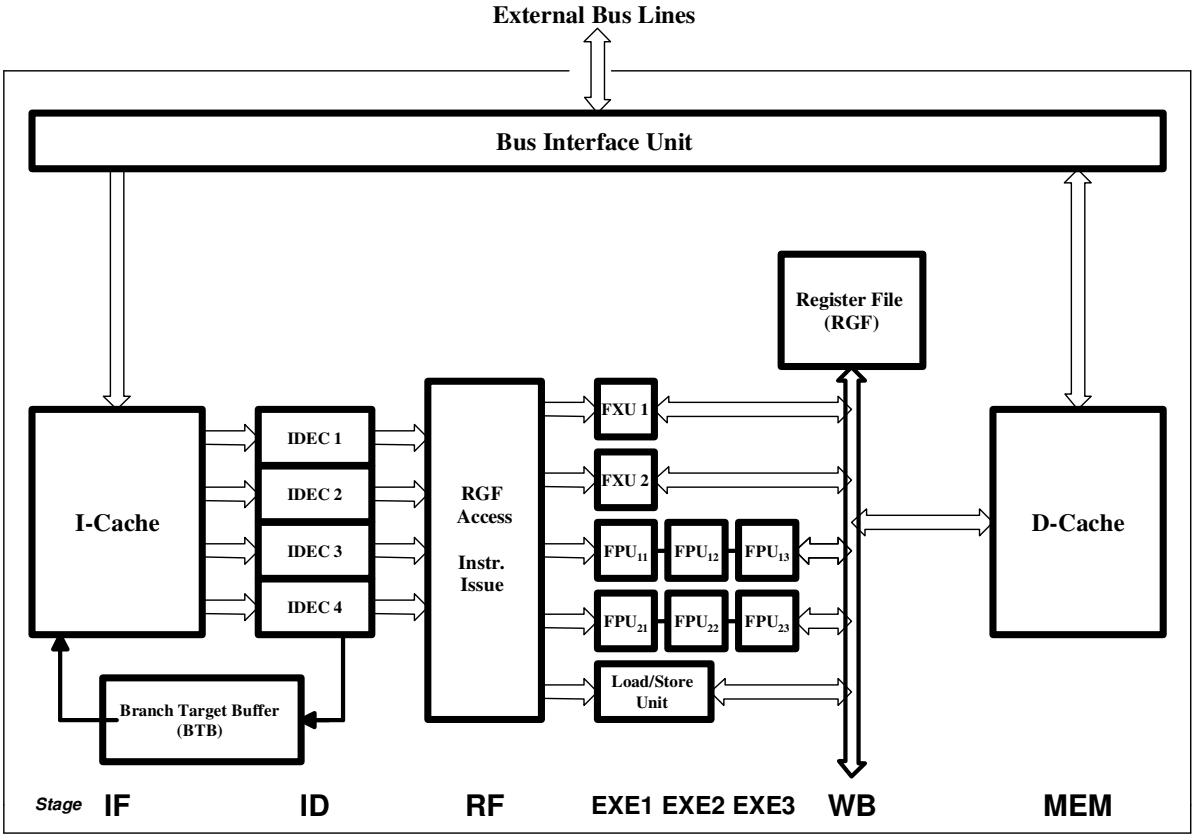
The register read is performed after the instructions are being issued to the FUs.
At instruction dispatch there is no copying operands into the register window, only tags for operands are loaded into the window.
The scheduler performs tag match to wake up ready instructions. All ready instructions that are issued obtain their operands directly from the RGF just prior to execution.

Result forwarding and **instruction wake up** are **decoupled**.

Example



The model of 8-stage pipelined *scalar* processor

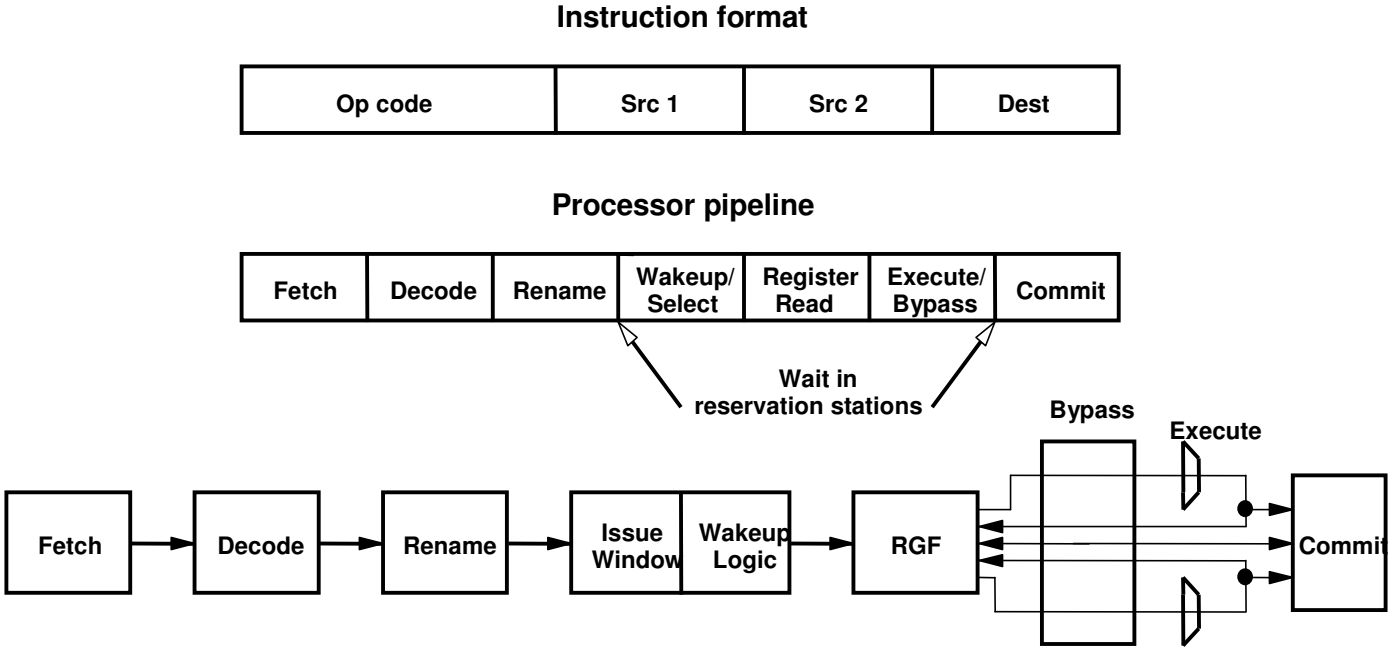


The model of 8-stage pipelined 4-way *superscalar* processor

Register Renaming and Reservation Stations

The dynamic instructions scheduling scheme was invented by Robert Tomasulo. Superscalar processors include huge register sets, so register renaming and initial dependency checks are performed before instructions are issued to the FUs.

Consider the 7-stage superscalar pipeline's model for dynamically scheduled processor. Instructions are scheduled without data capturing, in pipeline the stages are connected in order: "Rename" (RS) => "Wakeup/Select" => "Register Read" => "Execute/Bypass" (RS).



Comment

1. In the fetch stage, instructions are fetched from the instruction cache.
2. Then instructions are decoded and their register operands renamed.
3. Instructions in instruction window are free from control dependencies, and free from name dependencies.
4. Next instructions are written into the reservation stations where they wait for their source operands and a functional unit to become available.

Register renaming is a process whereby each new instruction is assigned a unique destination register from a pool of physical registers.

Register renaming is used to ensure each instruction is given correct operands and is done by dynamically mapping the logical (architectural) register numbers (names) used in program to processor's internal physical registers.
During register renaming processor removes false data dependencies (WAR, WAW) by writing the results of the instruction first into dynamically allocated rename buffers (ROB), rather than into the specified destination registers.

	Before renaming	After renaming	
I1	ADD **BX**, AX	ADD **R2**, R1	**RAW!**
I2	MUL CX. **BX**	MUL R3, **R2**	
I3	MOV **BX**, DX	MOV **R4**, R5	
I4	ADD **BX**, #5	ADD **R6**, #5	

The architectural destination register *BX* is renamed in instructions I1, I3 and I4.

The number of physical registers can be greater than the number of architectural registers, allowing registers to be reused less often. The mapping from architectural to physical registers is very similar to the mapping of virtual to physical memory addresses.
There are two main ways to implement the register renaming scheme in the superscalar processor:

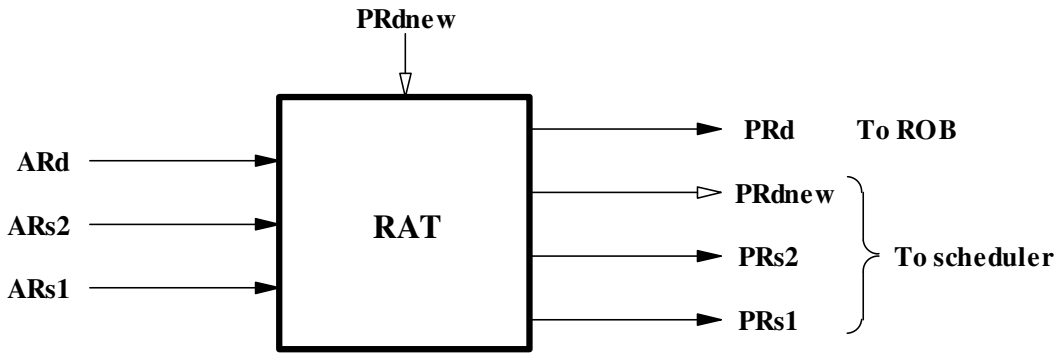
1. The first method provides a larger number of physical registers than the logical registers.
2. The second method uses a reorder buffer (ROB).

A common method to implement register renaming is to use a separate register rename file (RRF) or physical register file in addition to the architectural register file (ARF).
In a typical load/store ISA an instruction may have up to one destination register (ARd or Dest) and up to two source registers (ARs1 and ARs2 or Src1 and Src2). The register names as used by the instruction refer to the architectural register name space.

A straightforward way to implement the register renaming is to duplicate the ARF and use the RRF as a shadow version of the ARF. This will allow each architectural register to be renamed once, but this is not an efficient way to use the registers in RRF.
An alternative is to use a mapping table to perform renaming of the ARF. The core renaming activity takes place in the register alias table (RAT). The RAT holds the current mapping of architectural to physical registers.
Register renaming involves looking up a map table to pick up current mappings for architectural registers and, in parallel, true-dependence checking, followed by a modification of the mappings for source registers and that were produced by earlier instructions in the current rename group.

Generally renaming proceeds into two phases (A and **B):**

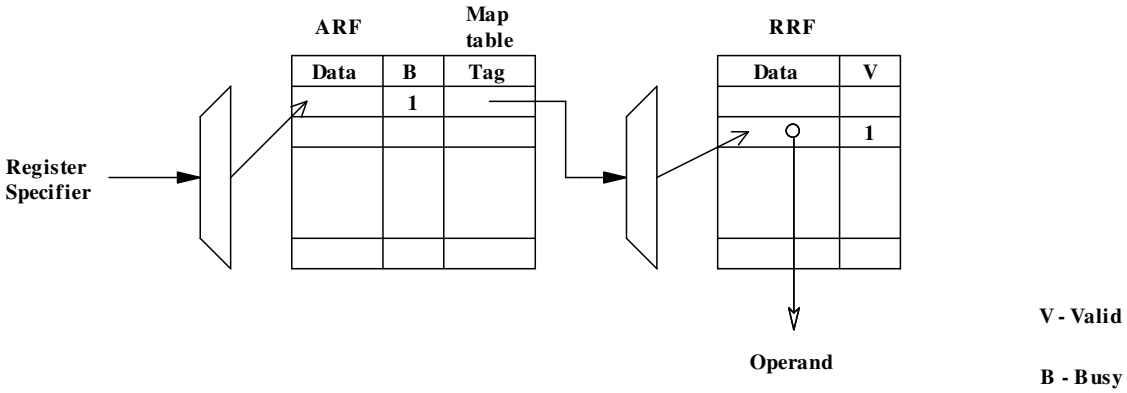
A1. Finding the current physical names **PRs1** and **PRs2** for the source (architectural or logical) registers **ARs1** and **ARs2**.
A2. Finding the current physical name **PRd** for the destination register **ARd**.
B. Assigning a new physical register for the destination register **ARd**.



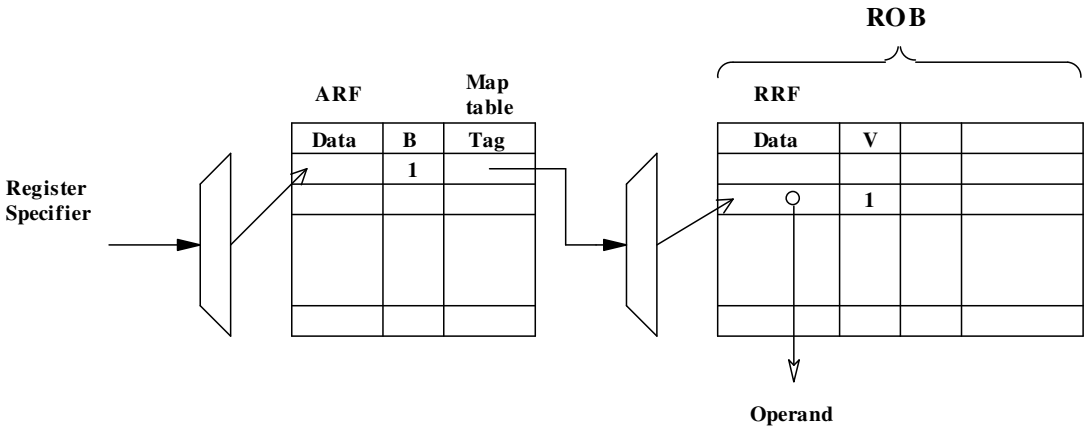
Renaming of a Single Instruction

After registers are renamed, the instruction is dispatched into out-of-order processing core. This is accomplished by establishing the entries for the instruction in two queues: the issue queue (IQ) and the ROB. The IQ is also known as the dispatch buffer or the reservation stations in distributed implementation.

Register Renaming Process Implementation



Stand-alone RRF



RRF attached to the ROB

When a separate RRF is used for register renaming, there are implementation choices where to place the RRF – stand-alone RRF or incorporated into ROB RRF.
In both options a busy field (B) is added to the ARF along with mapping table.
If the busy field of selected entry of the ARF is set (B:=1), indicates that the architectural register has been renamed, the corresponding entry of the map table is associated to obtain tag or pointer to the RRF entry.
If the RRF is in the ROB, then the tag specifies entry in a reorder buffer (ROB).
If the RRF is incorporated as part of the ROB, then every entry of the ROB contains an additional field that functions as a rename register.

Register operand fetch possibilities are:

1. The architected register contains specified operand (B=0).
The operand is fetched from the ARF.

2. The operand is result from some previous instruction and the content of the architected register is stale (B=1). The corresponding entry of the map table is accessed to retrieve the rename tag. The rename tag is specifies a rename register and used to index the RRF.

RRF indexing possibilities are:

1. The register updating instruction has finished execution (V=1). It is waiting to be completed. The operand is available in the rename register and is retrieved from the indexed RRF entry.

2. The register updating instruction has not been executed (V=0). The rename register has a pending update. The map table tag is forwarded to the reservation station instead of source operand. The tag is used later by the RS to obtain operand when it becomes available.

Destination allocation has three subtasks:

1. Set a busy bit (B) ;
2. Assign tag;
3. Update map table.

The task of register update takes place in the back end of the machine. It does not have direct impact on the operation of the RRF.

- When a register-updating instruction finishes execution, its result is written into the entry of the RRF indicated by the tag.
- When this instruction is completed its result is copied from the RRF into the ARF.

Once rename register is copied to its corresponding architectural register, its busy bit is reset (B=0) and it can be used again to rename another architectural register.
The ROB allows instructions to complete only in program order by permitting an instruction to complete only if it has finished its execution and all preceding instructions are already completed.

An advantage of the register renaming approach is that instructions commitment is simplified, since requires only two actions:

a. Record that the mapping between an architectural register number and physical register number is no longer a speculative;
b. Free up any physical registers being used to hold the "older" volume of the architectural register.

A physical register corresponds to an architectural register until the architectural register is rewritten.
Most high-end superscalar processors have chosen to use register renaming, adding from 20 to 80 extra registers. These extra registers replace a primary function of the ROB.

REM

1. The **physical registers** contain values of completed but not yet retired instructions.
2. The **architectural registers** store the committed values.
3. After committing of an instruction, copying its result from the physical register to the architectural register is required.

The Different Ways to Implement Register Renaming

The main renaming choices are:

1. Using a merged architectural and rename registers;
2. Employing a stand alone rename register file;
3. Keeping renamed values in the reorder buffer;
4. Keeping renamed values in the shelving buffers.

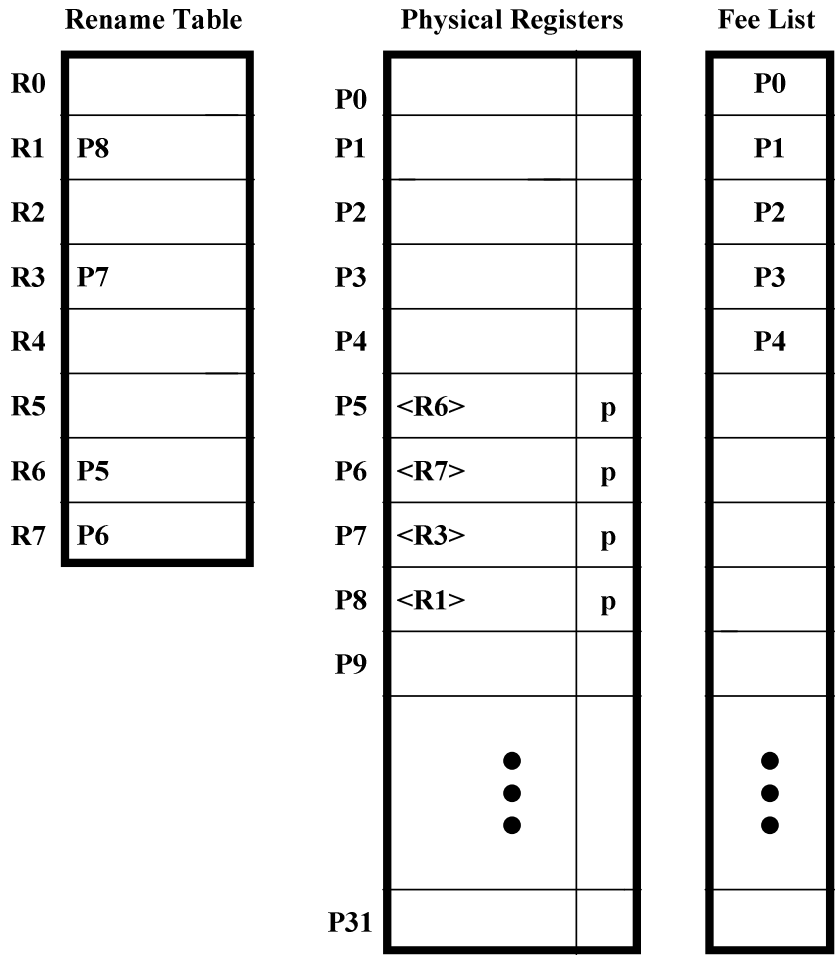
Example

Register Renaming

Initial State

Program Fragment

LD **R1**, Op(**R3**)
ADD **R3**, **R1**, #7f
SUB R6, R7, R6



ROB

Use	Operation	p1	OP1	p2	OP2	Dest	Old Dest	New Dest

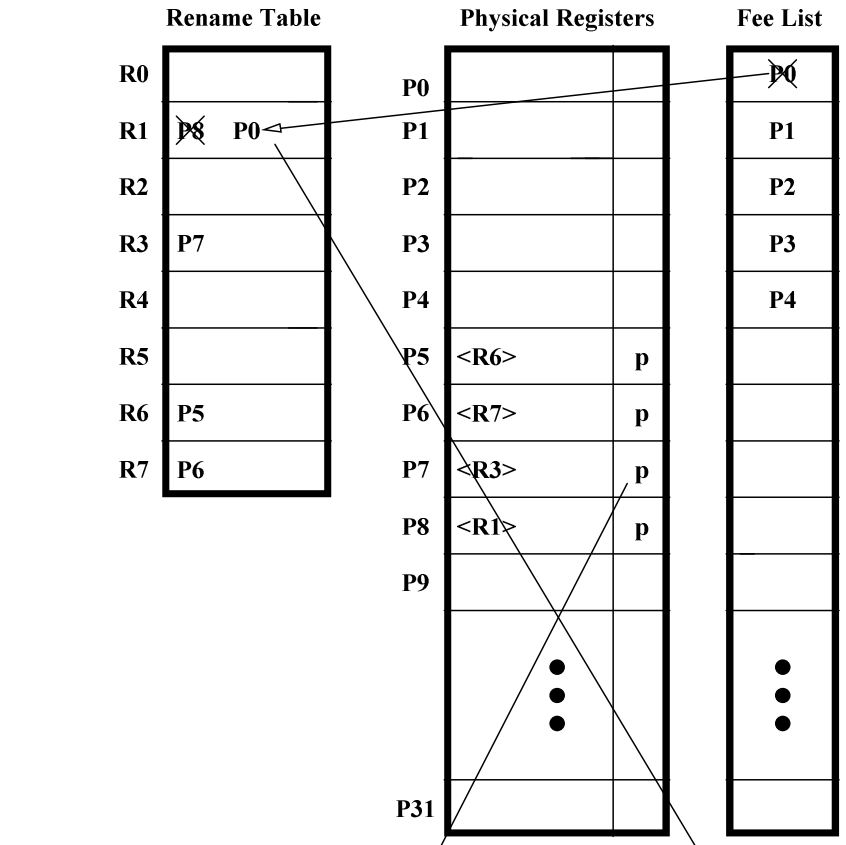
Example

Register Renaming

Step 1

Program Fragment

LD R1, Op(R3) (**Step1**)
ADD R3, R1, #7f
SUB R6, R7, R6



ROB

Use	Operation	p1	OP1	p2	OP2	Dest	Old Dest	New Dest
X	LD	p	P7			R1	P8	P0

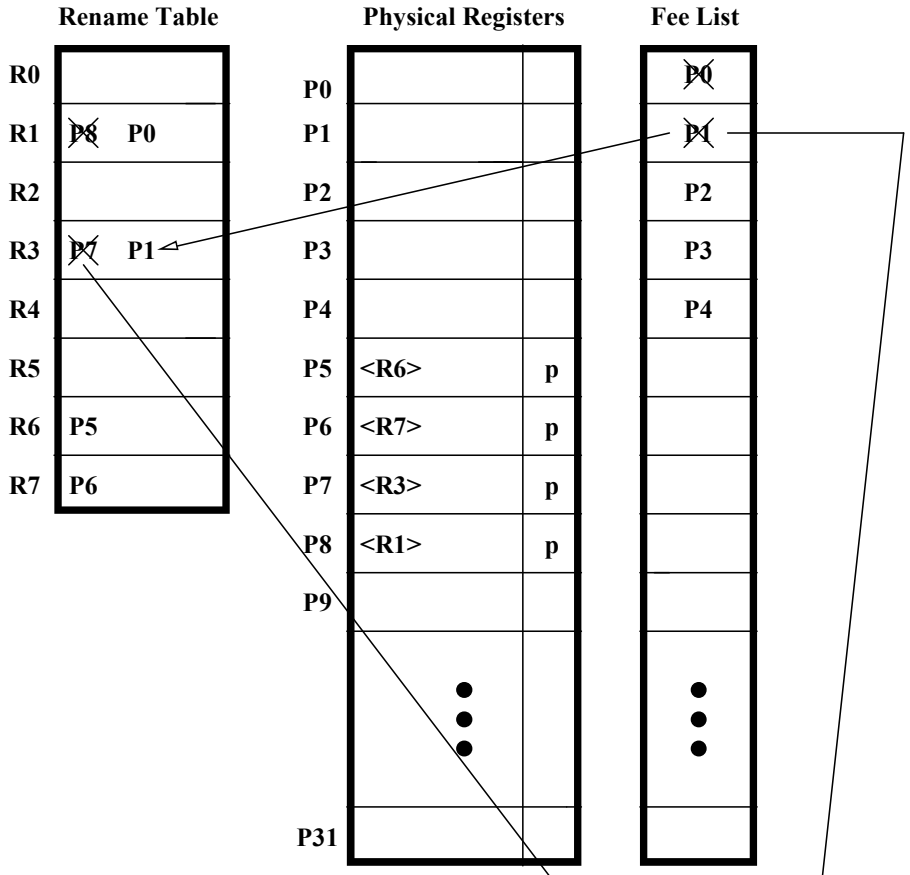
Example

Register Renaming

Step 2

Program Fragment

LD R1, Op(R3)
ADD R3, R1, #7f (**Step 2**)
SUB R6, R7, R6



ROB

Use	Operation	p1	OP1	p2	OP2	Dest	Old Dest	New Dest
X	LD	p	P7			R1	P8	P0
X	ADD		P0			R3	P7	P1

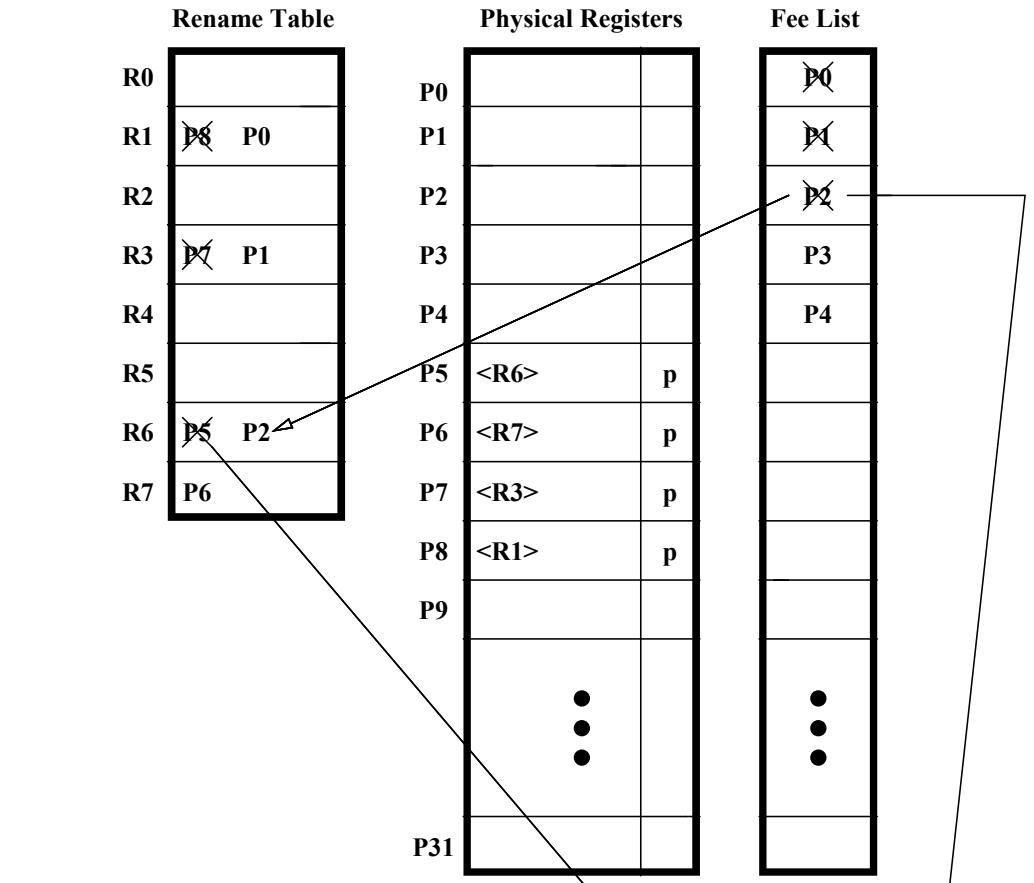
Example

Register Renaming

Step 3

Program Fragment

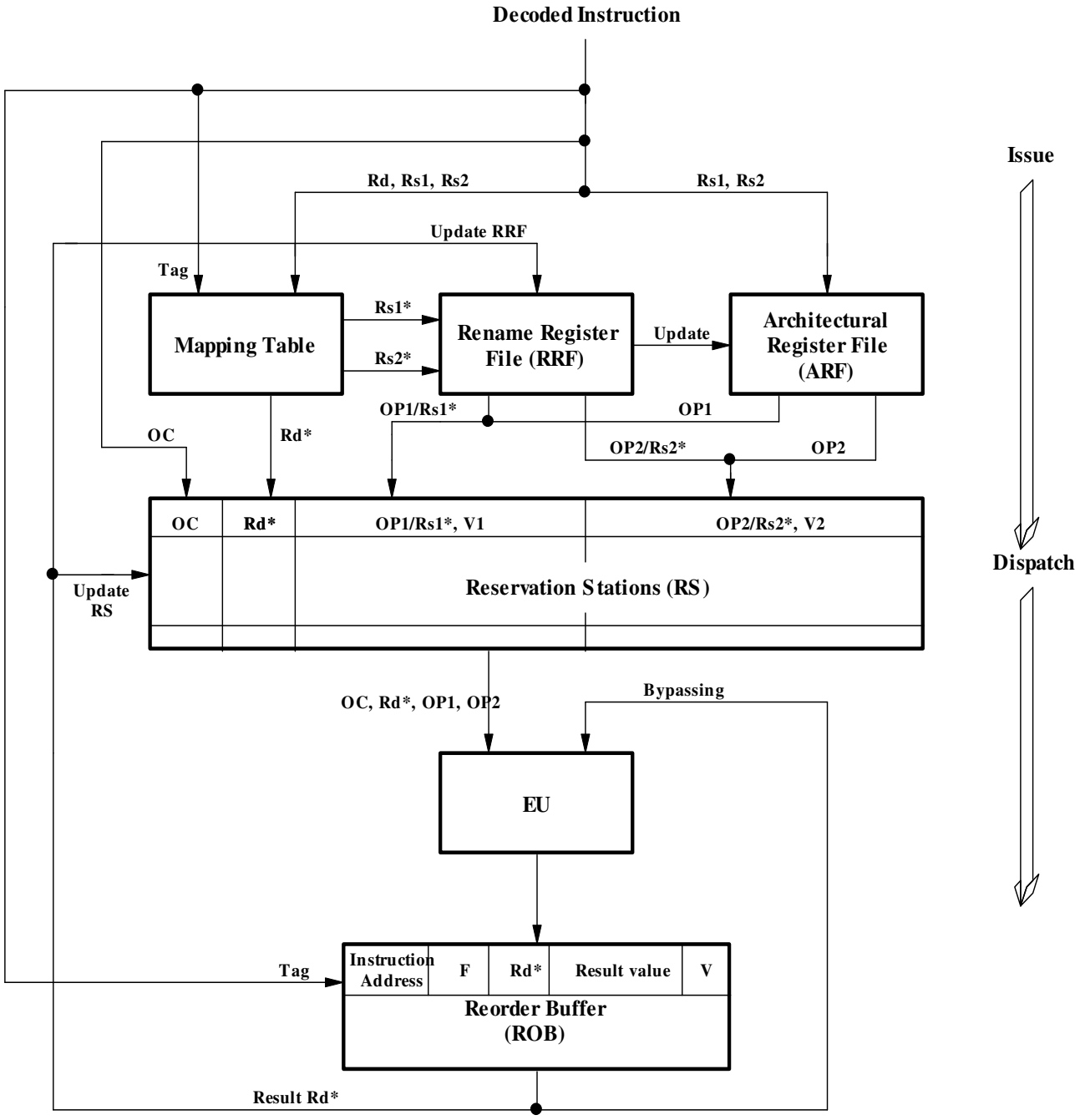
LD R1, Op(R3)
ADD R3, R1, #7f
SUB R6, R7, R6 (**Step 3**)



ROB

Use	Operation	p1	OP1	p2	OP2	Dest	Old Dest	New Dest
X	LD	p	P7			R1	P8	P0
X	ADD		P0			R3	P7	P1
X	SUB	p	P6	p	P5		P5	P2

Operand Fetching and Register Renaming in Superscalar Processor



V1, V2 - valid, operand OP1 or OP2 is valid;
OC - opcode of instruction;
Tag - the pointer to the entry in the ROB in which the result of instruction will be written.
V - valid, when true then it signifies that the Result Value field holds a valid result;
F - the flag, when true then it signifies that the this ROB entry is valid;
Rd* - the identifier of the allocated rename register. Corresponds to the entry in the mapping
Rs1, Rs2 - the source registers (Rs1, Rs2) which should be renamed;
Rs1*, Rs2* - the identifiers of the allocated rename registers (Rs1*, Rs2*).

Reservation Stations

The basic idea is that a reservation station fetches and buffers an operand as soon as it available, eliminating the need to get the operand from a register.
A RS can potentially support multiple instruction issues per cycle.

- When an instruction completes execution and produces a result, the result's name is compared to the operands names in the RS.
- If an instruction is waiting for this result in the RS, the data is written into the corresponding position and its availability indicator is set (V=1).
- When during dispatch an instruction is placed into the RS, all available operands from the register file are copied into this instruction's field.
- When all operands are ready, the instruction is executed.

After instruction completes execution, it waits in the reservation stations until all earlier instructions have completed execution. After this condition is satisfied, instruction is committed.

Instruction Scheduling Apparatus

The dynamic instruction scheduler consists of:

A. Instruction Window (Reservation Stations) and Reorder Buffer
B. Instructions Wake-up Logic
C. Instructions Select Logic

Instruction Window (Reservation Stations) and Reorder Buffer

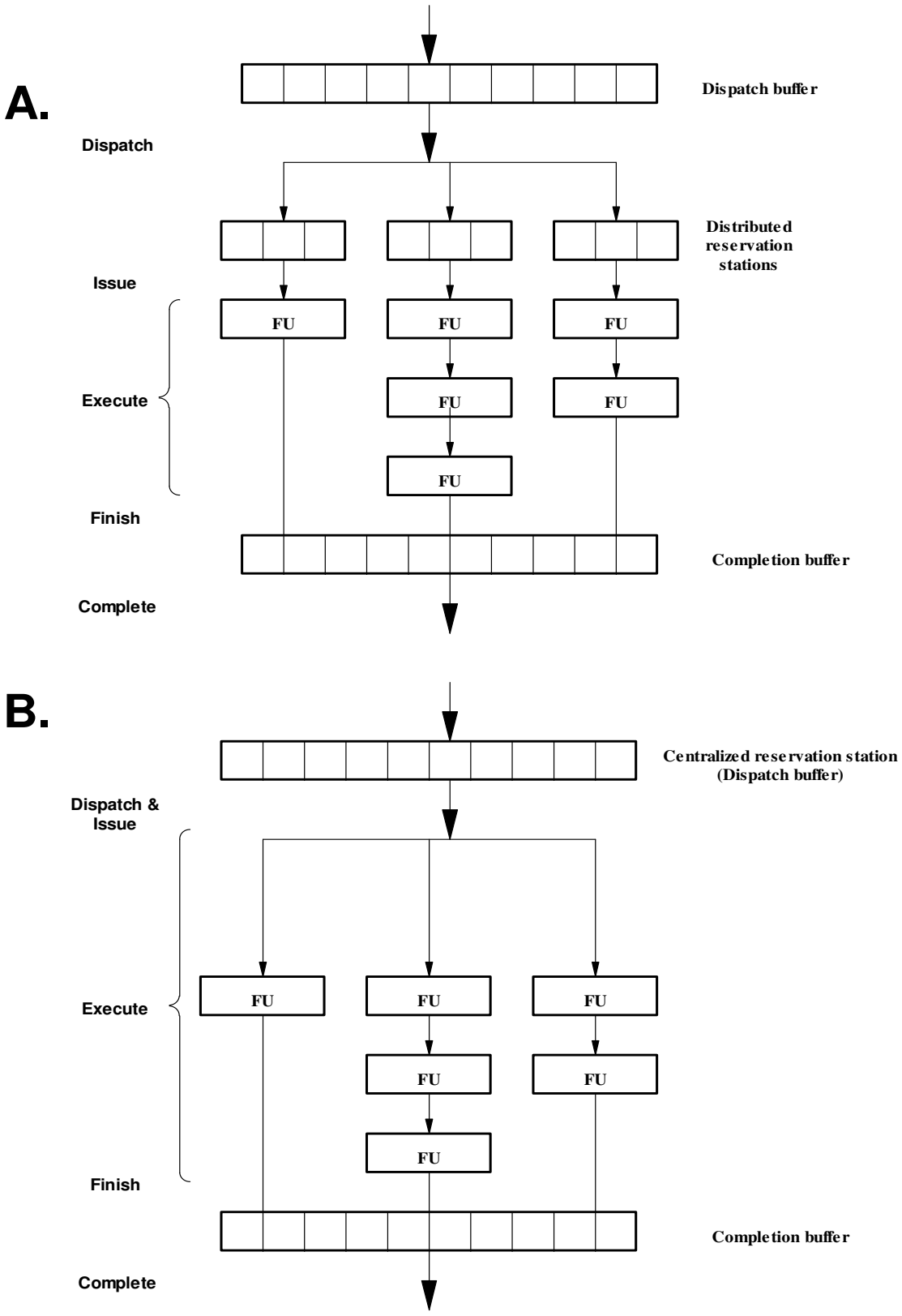
After an instruction is fetched, decoded and reordered, it is written into a reservation station entry (RSE). There are two types of reservation stations (RS):

1. Centralized reservation stations (dispatch buffer) (fig. B);
   A single buffer is used at the source side of dispatching.
2. Distributed reservation stations (fig. A, C).
   Multiple buffers are placed at the destination side of dispatching.

A centralized RS allows all instruction types to share the same reservation station.

Distributed RSs can be single ported buffers, each with only small number of entries.
Reservation stations idling entries cannot be used by instructions destined for execution in other functional unit.

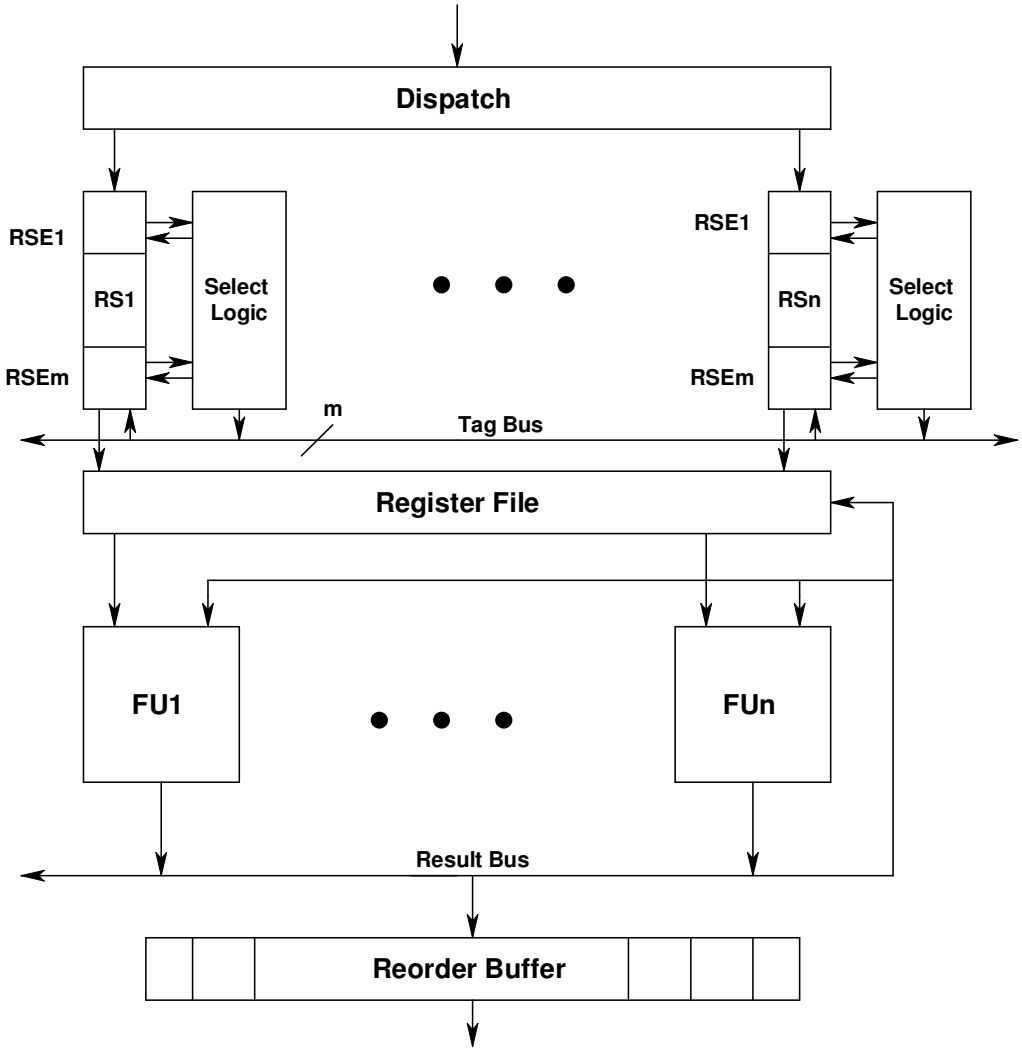
Distributed and Centralized Reservation Stations



Dispatching implies the associating of instruction types with functional unit types after instructions have been decoded.
Issuing means the initiation of execution in functional units.

- With a **centralized reservation station**, instruction dispatching and instruction issuing occur at the same time.
  With a **centralized reservation station**, the dispatching of instructions from the centralized reservation station does not occur until all their operands are ready.
- In a **distributed reservation station** these two events occur separately. Instructions are dispatched from the centralized decode/dispatch buffer to the individual RSs first, and when all their operands are available, then they are issued into the individual functional units for execution.

C.

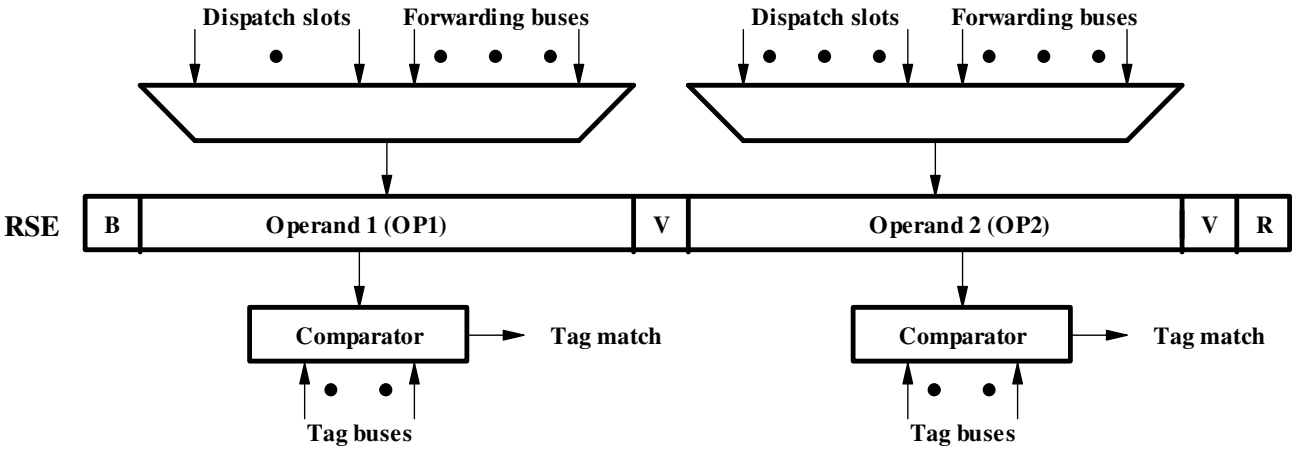


Reservation Station Entries

Typically the dispatching of an instruction requires three steps:

1. Select a free RS entry;
2. Load operands and/or tags into the selected entry;
3. Set the busy bit of that entry.

Each reservation station entry contains information about each of the instruction's sources, whether the source is ready and the number of cycles it takes producer of the source's value to execute.



Reservation Station Entry (RSE)

SRC tag	M	Shift	R	Delay	SRC tag	M	Shift	R	Delay	DEST tag

SRC tag – source operand tag;
M (match) – match when the destination tag of parent is broadcast and the tag comparator indicates that a match occurred;
Shift – on a tag match , the Shift field is loaded with the value contained in the Delay field.
R (ready) – this bit for each source is set if the data for that source is available in the register file or is available for bypass from functional unit;
Delay – because not all instructions have the same execution latency, the number of cycles between the time their tags are broadcast and the time their results are available is not constant. DEST tag – the destination register tag.

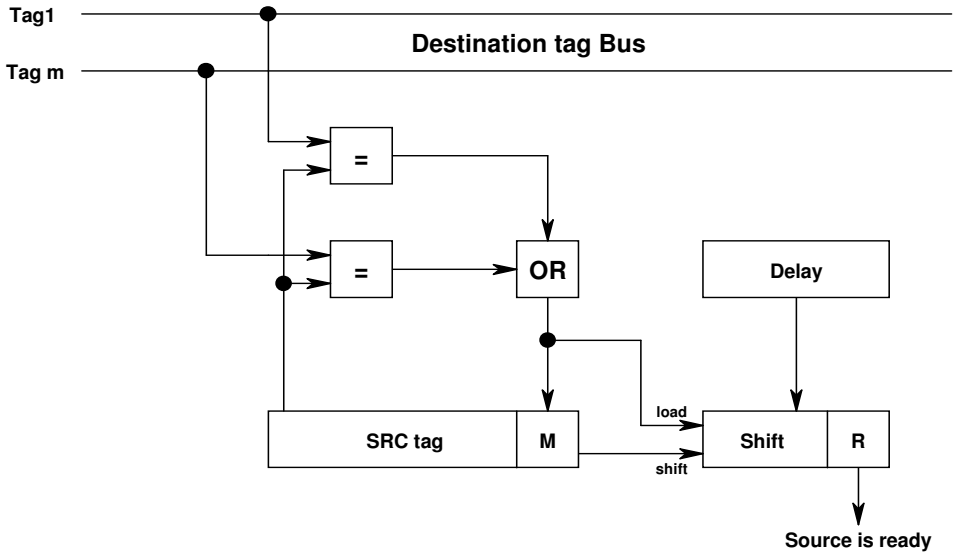
When a RSE is waiting for a pending operand, it must continuously monitor the tag bus.
When a both operand fields (M, R) are valid, then it is referred to as instruction wake up.

Instruction Wakeup Logic

The wakeup logic is responsible for waking up the instructions that are waiting for their source operands to become available. The wakeup logic sends the select logic a vector indicating which instructions are ready for execution.

For pipelined scheduling with speculative wakeup, it is accomplished by monitoring each instruction's parents and grandparents. The wakeup logic is a part of reservation stations.

Conventional Wakeup Logic



Instructions Select Logic

The select logic chooses instructions from the pool of ready instructions marked in a Request Vector. The select logic outputs a vector indicating the selected instructions which in turn becomes the input to the wakeup logic in the next clock.

Scheduling (wakeup+select) logic loop:

Wakeup and select logic form a critical loop. If this loop is stretched over more than one cycle, dependent instructions cannot execute in consequtive cycles.
Instructions that are ready to be scheduled in the current clock cycle produce results which are fed to dependent instructions that must be scheduled in the following clock cycle.
Each functional unit (FU) has a set of dedicated RSEs. Select logic associated with each FU selects the instruction that the FU will execute next.
After an instruction is selected for execution, several cycles pass before it completes execution. During this time, instructions dependent on it may be scheduled for execution.

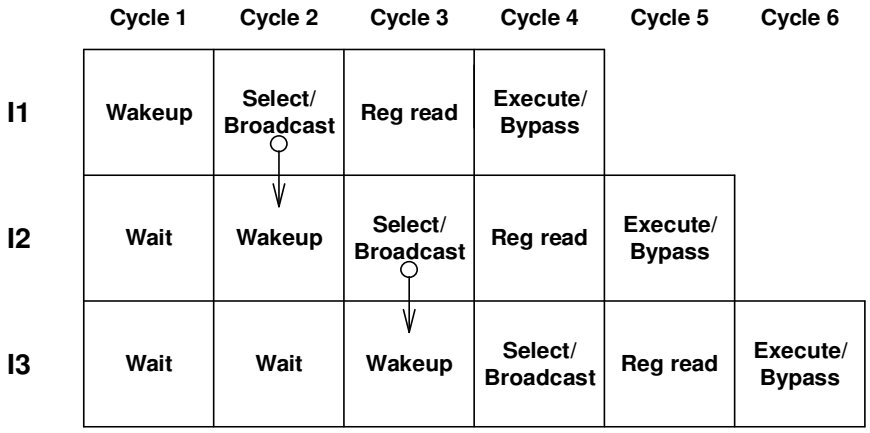
In a superscalar processor, a group of instructions must be reordered at the same time.
All the ready instructions are identified by their ready (R) bits. If more than one instruction requests execution, heuristics may be used for choosing which instruction receives grant.
The heuristics can be based on program order or how long each ready instruction has been waiting in the reservation station (RS).

The inputs to the select logic are the request signals from each of the FU's RSEs, plus additional information needed for scheduling heuristics such as priority information.
Because instructions cannot wakeup until all instructions they are dependent on have been selected, the wakeup and select form a critical loop. Generally, wakeup and select together constitute an atomic operation that is if they divided into multiple pipeline stages, dependent instructions cannot execute in consecutive cycles.

I Scheduling with Dependent Instruction Execution

An instruction wakes up in the last half of a clock cycle, and is potentially selected in the first half of the next clock cycle.
If the instruction is selected, the Grant Vector from the select logic gates information's destination tag onto tag bus, which is then fed to the tag comparators of the wakeup logic.
The tasks selection, tag broadcast, and wakeup must all occur within in one cycle.

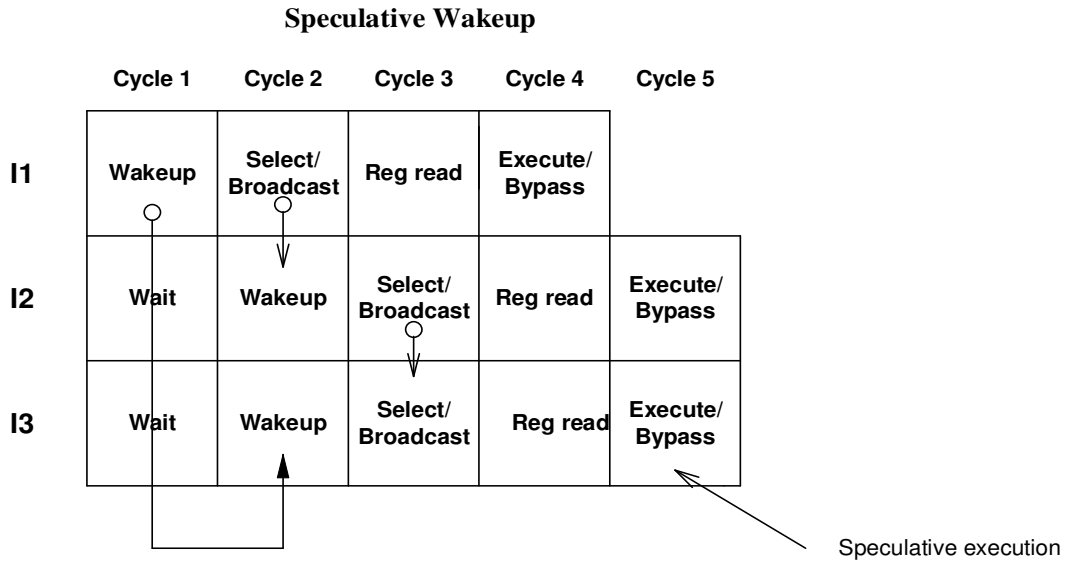
Conventional Wakeup



The schedule scheme assumes that each instruction has one-cycle latency.

II Scheduling with Speculative Wakeup

Instructions could be speculatively executeded with the aid of data value prediction mechanism while the operands of the instruction are not ready.
If the parents of an instruction's parent (grandparents) have been selected, then it is likely that the parent will be selected in the following cycle.

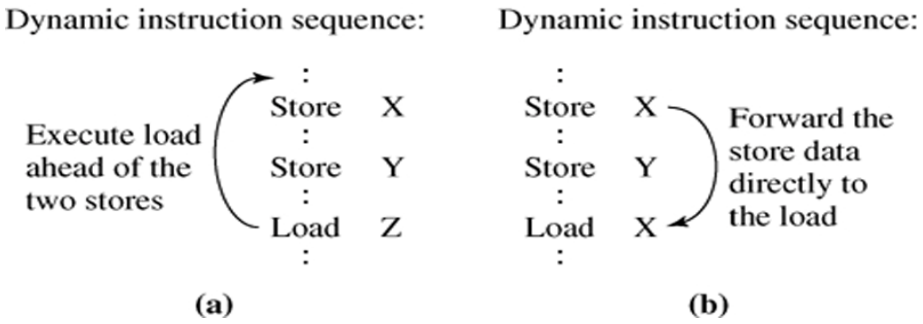


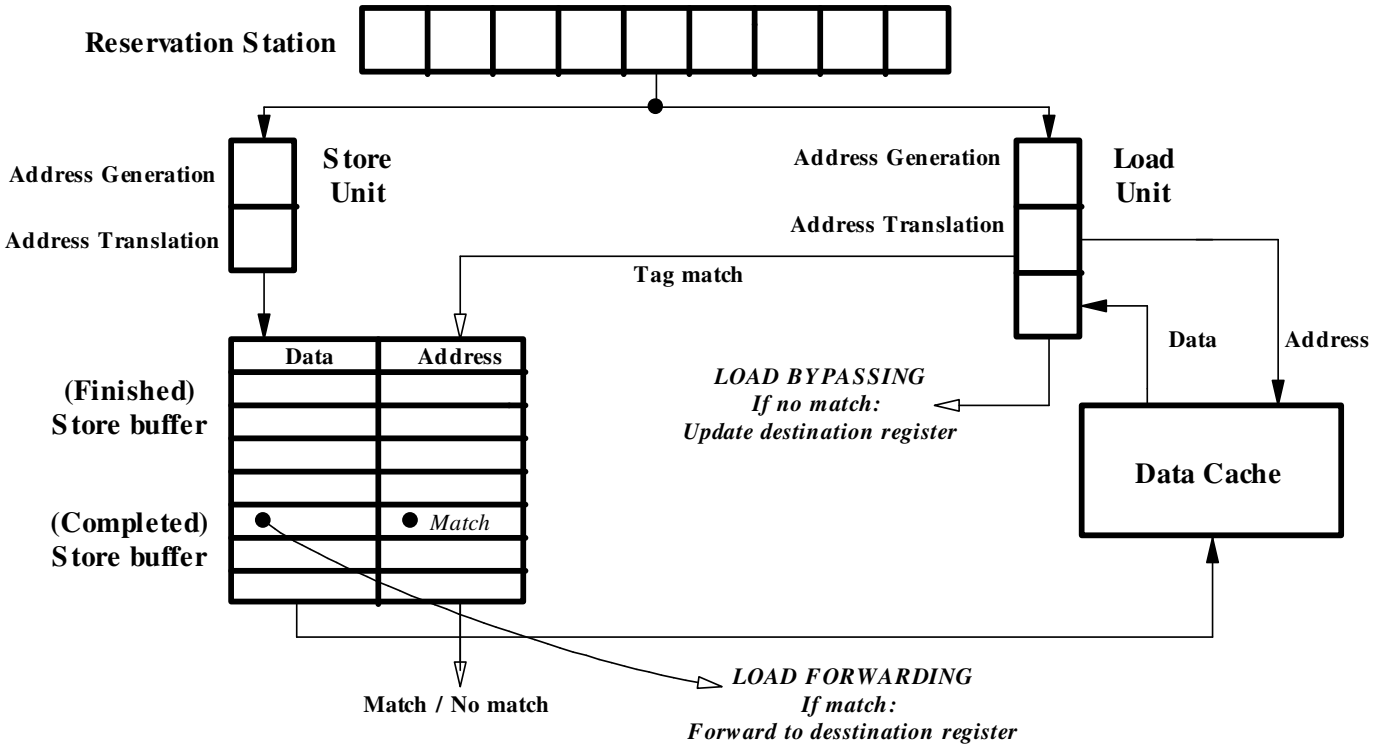
For scheduling logic pipelined over 2 cycles, the child can assume that when the tags of the grand parent pair have been received, the parent (I2) is probably being selected for execution and will broadcast its tag in the following cycle.
The child can (I3) then speculatively wakeup and be selected the cycle after its grand parent (I1) is selected.

Load Bypassing and Load Forwarding

There are two techniques for early out-of-order execution of loads – load bypassing //*laade möödumine*// and load forwarding //*laade ettesuunamine*//.

Load bypassing (**a**) allows a load to be executed earlier than proceeding stores if the load address does not alias //samanimesus// with the preceding stores.
Load forwarding (**b**) allows the load to receive its data directly from the store without having access the data memory.





The execution core responsible for processing load/store instructions has one store unit and one load unit. Both are fed by a common reservation station.
We assume that load and store instructions are issued from the RS in program order.
The store buffer operates as a queue and has two portions – finished and completed.

- The finished portion contains those stores that have finished execution but are not yet architecturally completed.
- The completed portion of the store buffer contains those stores that are completed architecturally but waiting to update the memory.

When a finished store is completed by the ROB, it changes from the finished state to the completed state. A store (instruction) does not really finish its execution until it is retired.
Key issue in implementing load bypass is the need to check for possible aliasing with preceding stores.

The load forwarding enhances and complements the load bypassing technique.

To support load forwarding added complexity to the store buffer is required.

Instruction Execution Phase

Execution Units Mix

The instruction frequencies of each type of instruction determine the optimal number of execution units (FU) of each type:

p_i – probability that any instruction is of type **i** (depends on what program is running**);**
IW – superscalar width;
$IW \times p_i$ – average number of instructions of type **i** in fetch group;
n_i – the number of execution units (FU) for instruction type **i.**

The total number of execution units (FU) is equal to:

$$\sum_i n_i \succ IW \times \sum_i p_i \cong IW \; ; \quad n_i > IW_i$$

The total number of execution units must be larger than the superscalar width.

Functional units' latencies

Issue latency – minimum number of clock cycles between the issuing of an instruction to a FU and the issuing of the next instruction to the same FU.
Result latency – number of clock cycles taken by a FU to produce a result.

Typical Functional Unit Latencies

Functional Unit	Issue latency	Result latency
Integer ALU	1	1
Integer Multiply	1	2
Load (on hit)	1	1
Load (on miss)	1	40
Store	1	—
FP Add	1	2
FP Multiply	1	4
FP Divide	12	12
FP Convert ($X_{10} \Leftrightarrow X_{IEEE\ 754}$)	1	2

Superscalar Processor Performance Degradation due to Dependencies

	Scalar Processor	Superscalar Processor without dependencies	Superscalar Processor with *false data dependencies* (WAR, WAW)	Superscalar Processor with *all data dependencies* (RAW, WAR, WAW)	Superscalar Processor with *all data* and *control dependencies*
Instructions executed per 10000 Cycles	***10000***	***40000***	***32459***	***26797***	***20765***

Superscalar Processor Performance Improvement with Dependencies Resolution

	Register renaming	Branch Prediction	Shelving
Instruction Window size – 8	***21885***	***24500***	***28958***
Instruction Window size – 16	***22736***	***25723***	***30393***

The renaming eliminates completely the false data dependencies and improves the performance with about 25%.
The branch prediction provides 90% accuracy and improves the performance with about 28%.
The shelving provides full speed of the processor despite of the presence of true data dependencies and improves its performance with about 25%.

Misprediction Recovery in a Superscalar Processor

A single branch misprediction may flush upwards of 100 in-flight instructions, causing extended retirement stalls as the pipeline gradually fills. Because of the large per-instruction penalty, branch misprediction rates of 5-10% cause a disproportionate performance loss. Methods for dealing with branch misprediction vary, but all must prevent the predicted instructions from permanently modifying the register file until the outcome of the branch is known.

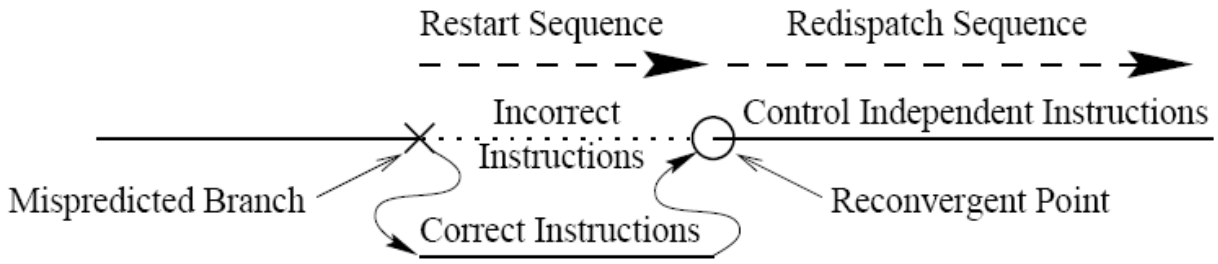
There are two main approaches to implement the correct behavior control speculation.

1. The first mechanism that relay on taking snapshots, or checkpoints, of the processor state at appropriate points and reverting back to the corresponding snapshot when a control mis-speculation is detected.
2. The second approach involves mechanism that reconstruct the desired state by sequentially processing the in-flight instructions in program order until a deviation from the program order is reached.

When a branch misprediction is detected in a superscalar processor, the processor performs a series of steps to ensure correct execution.
Typically, branch misprediction recovery requires stalling the front-end processor, repairing the architectural state, and restarting the process of fetching and renaming instructions from the correct path.
Instructions after the mispredicted branch are squashed and all resources they hold are freed. In addition, the mapping of physical registers is backed up to the point of the mispredicted branch. The instruction fetch unit is also backed up to the point of the mispredicted branch and the processor begins sequencing on the correct path.

Recovery may proceed as follows:

1. After a branch misprediction is discovered, the first control independent instruction must be found in the instruction window. It is the reconvergent point //*taastepunkt*//.
2. Instructions are selectively squashed //*kõrvaldatavad*//, depending on whether they are incorrect control dependent instructions or control independent instructions.
   Squashed instructions are removed from the window and any resources they hold are released.
3. Instruction fetching is redirected to the correct control dependent instructions, and new instructions are inserted into the instruction window which may already hold subsequent control independent instructions.
4. Based on the new, correct control dependent instructions, data dependences must be established with the control independent instructions already in the window.
   Any modified data dependences cause already-executed control independent instructions to be reissued with new data.
   This step is called the *redispatch sequence* //*uuesti jaotatud järjestus*//.



Finalizing Instruction Execution
Instruction Completion, Commitment and Retirement

In a distributed RS machine, an instruction can go through the following phases:

Fetch => Decode => Dispatch => Issue => Execute => Finish => Complete => Retire
In centralized RS machine one phase

- **Instruction finishes execution** //*käsk on töödeldud*// when it exits the FU and enters completion buffer.
- An **instruction is completed** //*lõpetatud*// when the functional unit is finished the execution of the instruction and the result is made available for forwarding or buffering.
- **Instruction committing** //*püsitatud*// means that the instruction **results** have been made **permanent** and the instruction retired from the instruction scheduler.

A **result is made permanent** //*püsitulem*// **by**:

a. Making the mapping of architectural to physical register permanent
b. Copying the result value from the rename register to the architectural register

Only the **architectural registers are renamed**. The RRF can be a separate stand-alone structure similar to the ARF or it is incorporated into the reorder buffer.]

- **Instruction retiring** //*erustatatud*// means instruction removal from the instruction scheduler with or without the commitment of operation results.

Reorder Buffer and Instruction Retirement

The ROB keeps the original program order of the instructions after instructions issued and allow result serialization during the retire stage.
The ROB contains all the instructions that have been dispatched, but not yet completed.
The typical implementation of an ROB is in the form of a multiported register file.
In m-way superscalar machine, where physical registers are implemented as slots within the ROB entries, the ROB has following ports:

1. At least 2m read ports for reading source operands for each of the m instructions dispatched/issued per cycle.
2. At least m write ports to allow to s functional units to write their result into the ROB slots acting as physical registers in one cycle.
3. At least m read ports allow up to s results to be retired into the ARF per cycle.

The status of each instruction in the ROB is tracked by using instruction state bits in every entry of the ROB.

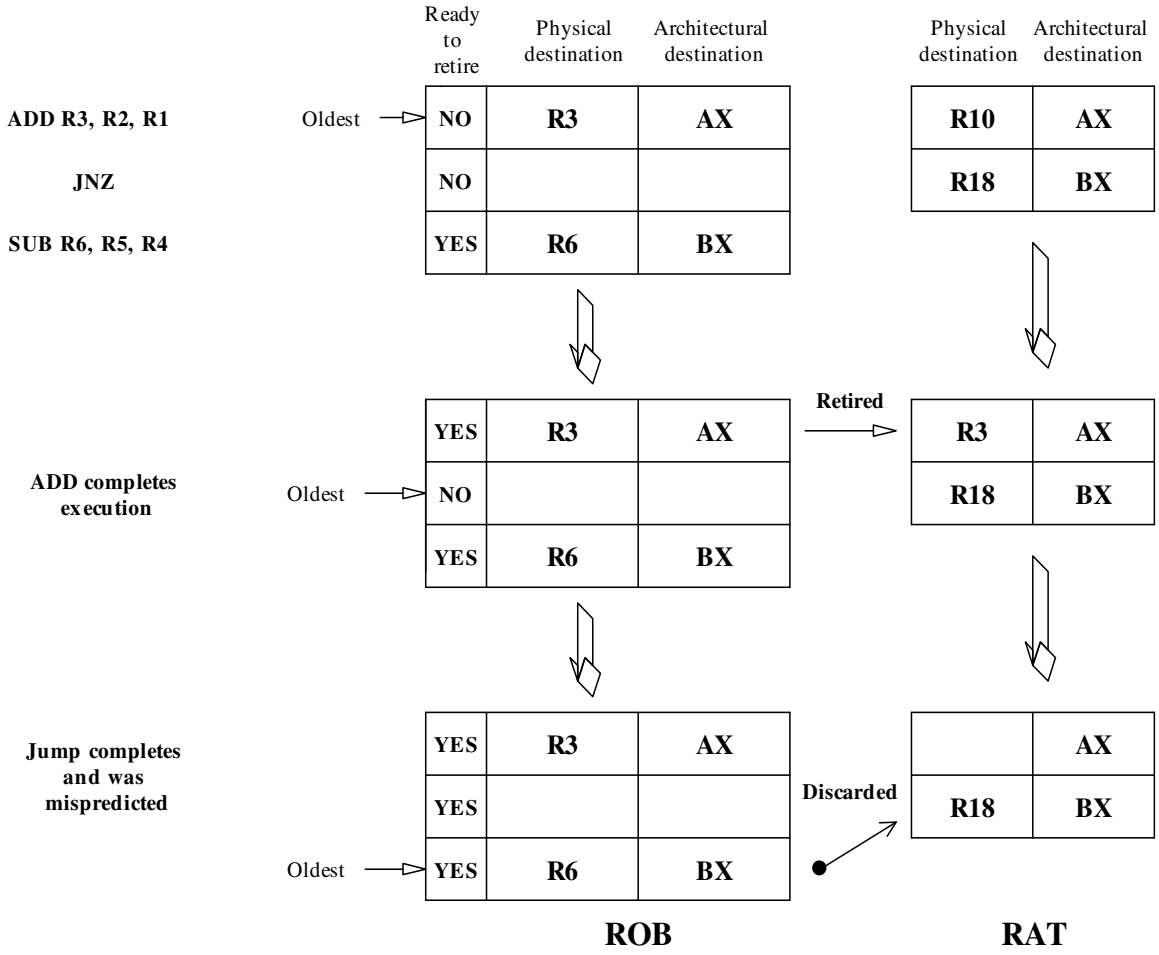
Typical ROB entry fields

Busy (B)	Issued (I)	Finished (F)	Instruction Address (IA)	Rename Register (RR)	Speculative (S)	Valid (V)

An instruction can be in one of several states: waiting execution, in execution, and finished execution. The status bits are updated as an instruction traverses from one state to the next. A special bit (S) is used to indicate whether an instruction is speculative or not.
If speculation can cross multiple branches, additional bits can be employed to identify which speculative basic block an instruction belongs to. When a branch is resolved, a speculative instruction can become nonspeculative or invalid.

Only finished and nonspeculative instructions can be completed.

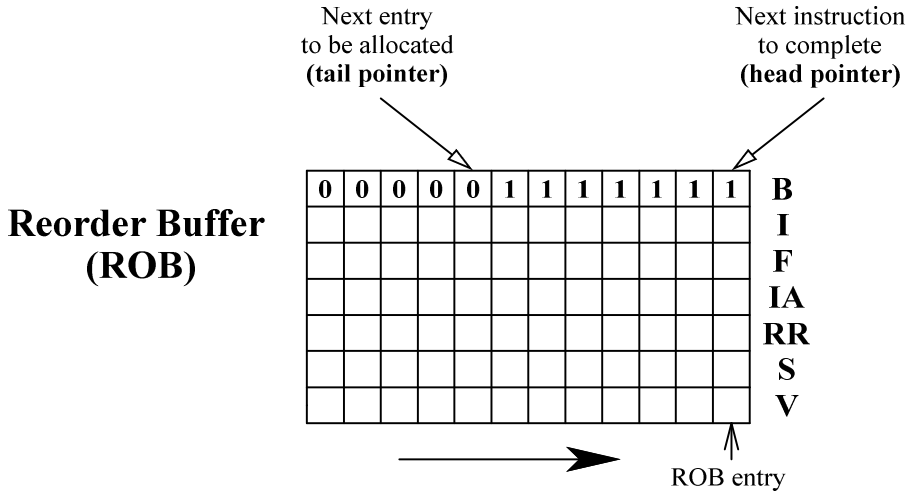
When an instruction is completed, the state is marked in its ROB entry. When a program interrupt occurs, the exception is marked in the ROB entry (not depicted on the figure).



The **ADD** instruction writes **result** to register **R3**, which has been mapped to architectural register **AX**. The subtract instruction **SUB** writes result to register **R6**, which has been mapped to architectural register **BX**.
The conditional jump instruction JNZ does not write to register.
At the right the RAT maintains a list of the physical registers holding the most recent committed in-order results for each architectural register.
When the ADD instruction completes execution, as the oldest instruction, it is allowed to retire.
The ROB oldest pointer is incremented to show that the most recent committed results for AX are stored in register R3.
Upon execution of the conditional jump (JNZ), it is discovered to have been mispredicted.
The jump is now the oldest entry in the ROB, and it is allowed to retire.
When SUB retires its result is discarded because the early retired jump instruction was mispredicted. The RAT is not updated and the latest committed results for BX are still in register R18.
The registers used by the subtract instruction will be recycled for use by other instructions.
The result the subtract instruction will be overwritten without ever having been read.

There are several different implementations of the ROB.

Often the ROB is implemented as a circular FIFO buffer with a head pointer and a tail pointer.
The tail pointer is advanced when ROB entries are allocated at instruction dispatch.
The number of entries that can be allocated per cycle is limited by the dispatch bandwidth.
Instructions are completed from the head of the queue.
Reorder buffer entries are allocated in the first issue stage and deallocated serially when the instruction retires.



During the retire stage a number of instructions at the head of the FIFO queue are scanned and an instruction committed if all previous instructions are committed or can be committed in the same cycle.
In the case of instructions that are on a misspeculated path, the instructions are removed from ROB and the physical registers freed without making the result permanent or copying back results.
When an instruction is completed its rename register and its reorder buffer entry are deallocated.

Typically the processors`s retire bandwidth is the same as the issue bandwidth.

The reorder buffer (ROB) accomplishes the following operations:

1. Allocate: The dispatch stage reserves space in the reorder buffer for instructions in program order.
2. Wait: The complete stage must wait for instructions to finish execution.
3. Complete: Finished instructions are allowed to write results in order into the architectural registers.

Store Buffer

Store instructions are processed differently than *Load* instructions.
Unlike a *Load* instruction, the data for *Store* instruction are kept in the ROB. At the time when the *Store* is being completed, these data are then written out to memory. The reason for this delayed access to memory is to prevent the premature and potentially erroneous update of the memory in case the *Store* instruction may have to be flushed.
For *Store* instruction it is possible to move the data to the Store buffer at completion.
The purpose of the Store buffer is to allow stores to be retired when the memory bus is not busy. This is giving priority to *Load* instruction(s) that needs to access to the memory.

Comparison of 5-stage Scalar and Superscalar Pipelines

Pipeline Stage	Scalar Processor	Superscalar Processor
Fetch	Fetch one instruction	Fetch multiple instructions
Decode	**1**. Decode instruction **2**. Access operand from register file **3**. Copy operands to functional unit input latch	**1**. Decode instructions **2**. Access operands from register file and reorder buffer (ROB) **3**. Copy operands to functional unit RSs
Execute	Execute instruction	**1.** Execute instructions **2.** Arbitrate for result buses
Write-back	**1**. Write result to register file **2**. Forward results to functional unit input latches	**1.** Write results to ROB **2.** Forward results to functional unit reservation stations
Commitment	---	Write results to register file

Performance (IPC) for Superscalar Architecture

Superscalar processors with parallel pipelines **complete many instructions every clock cycle**.
If there are s parallel pipelines which do not stall, then s instructions can complete every clock cycle.
When stalls occur, the superscalar processor may still be able to complete a smaller number of instructions (from (s-1) to zero).

IPC(c) – the number of instructions completed during **clock cycle c.**

$$\mathbf{s \geq IPC(c) \geq 0}$$

The average IPC can be determined from

$$\mathbf{IPC} = \frac{1}{N_c}\sum_c IPC(c), \quad \text{where}$$

$\mathbf{N_c}$ - is the total number of clock cycles.

The complexity of superscalar processor usually makes it very difficult to calculate **IPC(c)** with simple formulas. Simulation of the hardware running a set of benchmark programs is usually used.

Superscalar Processor Summary

Superscalar processors are distinguished by their ability to issue multiple instructions each clock cycle from a conventional instruction stream.

Superscalar Pocessor's Main Features

1. Instructions are issued from a sequential stream of normal instructions.
2. The instructions that are issued are scheduled dynamically by the HW
3. More than one instruction can be issued each instruction cycle.
4. The number of issued instructions is determined dynamically by HW.
5. The dynamic instruction issue complicates the HW scheduler of a superscalar processor.
6. It is a presumption that multiple functional units are available.
7. The superscalar technique is a microarchitecture technique, not an architecture technique.
8. Instruction pipelining and superscalar techniques both exploit fine-grain parallelism.

Comment

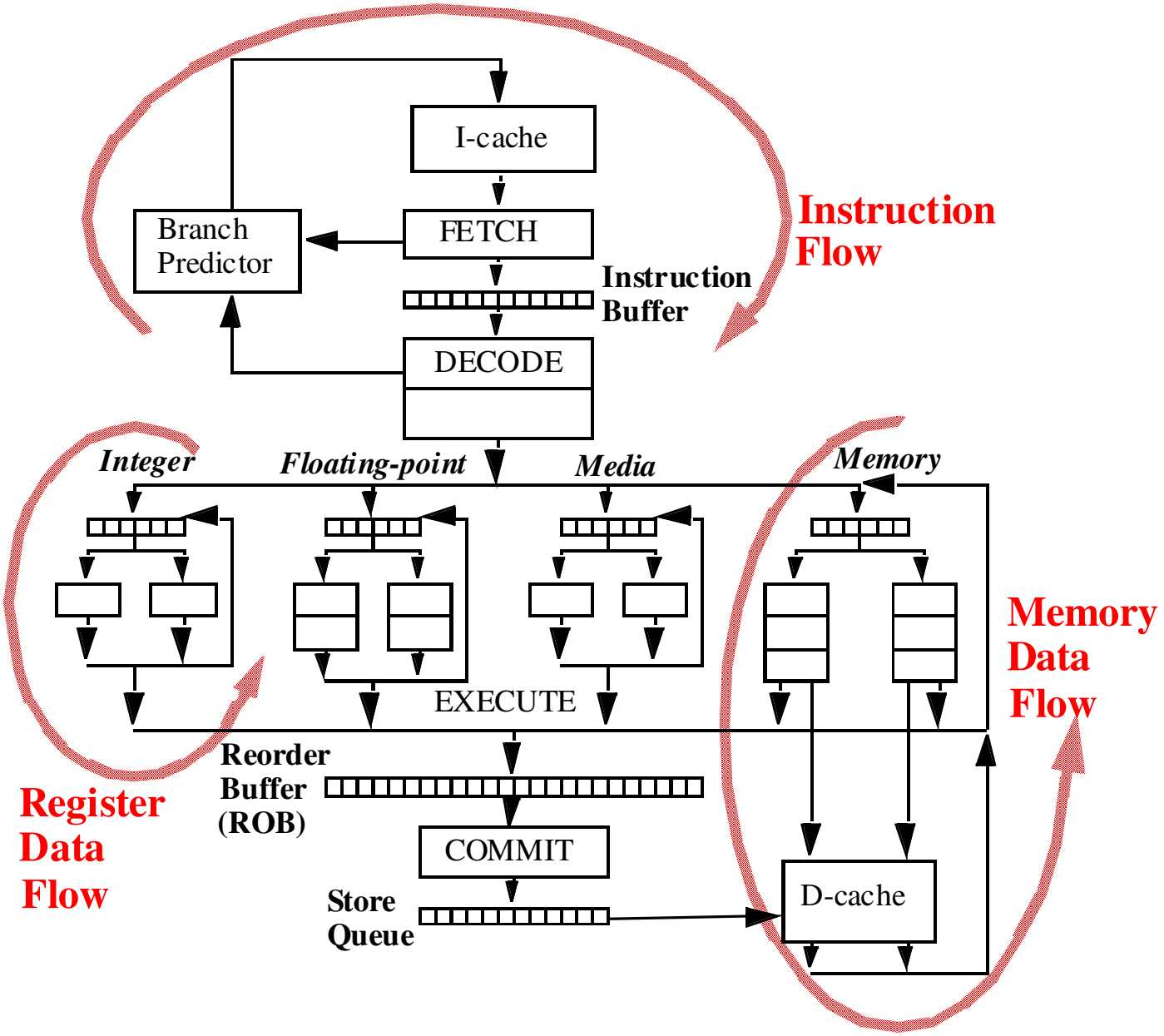
At least two circumstances **limit the superscalar processor efficiency**:

1. The degree of the ILP;
2. The complexity of a superscalar processor increases as the same rate, and even faster, as the number of concurrently executed instructions.

Doubling issue rates [in modern superscalar processors] ***above today's 3-6 instructions per clock, says to be 6-12 instruction, probably requires a processor to:***

- ***Issue 3 or 4 data memory accesses per cycle;***
- ***Resolve 2 or 3 branches per cycle;***
- ***Rename and access more than 20 registers per cycle;***
- ***Fetch 12 to 24 instructions per cycle.***

David Patterson



First-, Second- and Third-Generation Superscalar Processors

First Generation	Second Generation	Third Generation
Issue width		
2-3 RISC instr./cycle or 2 CISC instr./cycle	4 RISC instr./cycle or 3 CISC instr./cycle	4 RISC instr./cycle or 3 CISC instr./cycle
Processor core features		
No predecoding	Predecoding	Predecoding
Static branch prediction	Dynamic branch prediction	Dynamic branch prediction
Direct issue	Dynamic instr. scheduling	Dynamic instr. scheduling
Blocking execution of unresolved conditional branches	Speculative execution of unresolved conditional branches	Speculative execution of unresolved conditional branches
No renaming	Renaming	Renaming
No reorder buffer	Reorder buffer	Reorder buffer
No out-of-order loads	Out-of-order loads	Out-of-order loads
No store forwarding	Store forwarding	Store forwarding
Cache subsystem features		
Single ported data cache	Dual ported data cache	Dual ported data cache
Blocking L1 data caches or non-blocking caches	Non-blocking L1 data caches with multiple cache misses allowed	Non-blocking L1 data caches with multiple cache misses allowed
Off-chip L2 caches attached via the processor bus	Off-chip direct coupled L2 caches	On-chip L2 caches
Instruction Set Architecture features		
No multimedia and 3D support	No multimedia and 3D support	FX- and FP-SIMD instructions
Examples		
Alpha 21064, PA 7100, Power PC 601, Pentium	*Alpha 21264, PA 8000, Power PC 620, Pentium Pro*	*Power 4, Pentium III, Pentium 4, Athlon MP* (m 6)

RISC ARCHITECTURE**

CISC – **C**omplex **I**nstruction **S**et **C**omputer
RISC – Reduced **I**nstruction **S**et **C**omputer
//kärbikprotsessor//

Term ***RISC*** was first used by **David Patterson** in 1980 (Berkeley Stanford University).

Design target:

max P,
min C.

Basic (classical) RISC architecture versions:

University of California at Berkeley (David Patterson)
RISC I (1982) [44420 transistors; 32 instructions; 5µm NMOS; 1 MHz; die area 77mm^2]
RISC II (1983*)* [40760 transistors; 39 instructions; 3µm NMOS; 3 MHz; die area 60 mm^2]

Stanford University (John Hennessey)
MIPS (1981)
MIPS – **M**icroprocessor without **I**nterlocked **P**ipeline **S**tage
Predecessor - *IBM 801* **(**1979**)**

IBM 801 was a high-performance minicomputer which was never marketed; it was designed by John Cocke**.**

IBM 801 architecture main features:

1. The trivial instruction set (RISC-like),
2. HLL orientation (special compiler),
3. HW control,
4. One instruction per clock cycle,
5. 32-bit CPU with 32 registers,
6. ECL-technology.

RISC architecture development - 80/20 Rule

Increases in architectural complexity catered to the belief that a significant barrier to better computer performance is the semantic gap – *the gap between the meanings of high-level language statements and the meanings of machine-level instructions*.
The studies for CISC computers show that:

1. **About 80% of the instructions generated and executed use only 20% of an instruction set.**

Conclusion 1

If this 20% of instructions is speed up, the performance will be greater.

2. Analysis shows that these instructions tend to perform the simpler operations and use only the simple addressing modes.

Conclusion 2

If only the simpler instructions are required, the processor hardware required to implement them, could be reduced in complexity.
It should be possible to design a more performance processor with fewer transistors and less cost.

Conclusion 3

With a simpler instruction set, it should be possible for a processor to execute its instructions in a single clock cycle and synthesise complex operations from sequences of instructions.

RISC Architecture Main Principles

1. An efficient system operation can be attained if all instructions take the same number of cycles for the fetch and execution stages in the pipeline.

2. If the above take a single clock cycle, the operation will be the speediest for given technology.

3. We have to achieve uniform, single-cycle fetch, decode, execute, etc. operations for each instruction implemented on the processor.

4. A single-cycle fetch can be achieved by keeping all instructions at a standard size (32 bit).

5. The standard instruction size should be equal to the basic word length of the processor. Uniform word-width instructions simplifying instructions prefetching and eliminate complications that arise from instruction crossing word, block, or page boundaries.

6. Achieving same time duration execution of all instructions is much more difficult than achieving a uniform fetch.

7. Which instructions should be selected to be on the reduced instruction list?
The most often executed instructions are data moves, arithmetic and logic operations.

8. The general support of HLL.
This consideration supports the reduction of the semantic gap between the processor design and the HLLs.

9. The procedure call-return is the most time-consuming operation in typical HLL programs. The percentage of time spent on handling local variables and constants turned out to be highest, compared to other variables.

10. It must be minimized as much as possible the number of instructions that have to access memory during the execution stage.

- **a.** Memory access, during the execution stage, is done by LOAD and STORE instructions only.
- **b.** All operations, except LOAD and STORE, are register-to-register, within the processor.
- **c.** Systems featuring these rules are said to be adhering to a load/store memory access architecture.

11. Uniform handling of instructions can be achieved by using hardwired control.

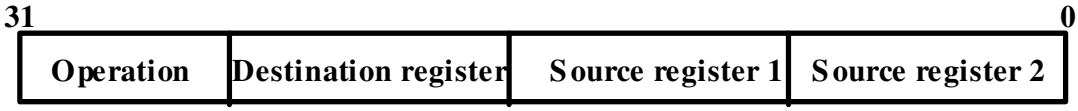
12. The central processor unit is provided with a large register set (register file).

13. RISC architecture exploits ILP in the instruction pipeline. As a rule, the instruction pipeline is supported by advanced cache memory system.

The original MIPS, SPARC and Motorola 88000 CPUs had classic scalar RISC pipelines, which contained five pipeline stages:

1. Instruction fetch;
2. Decode;
3. Execute;
4. Access;
5. Write back.

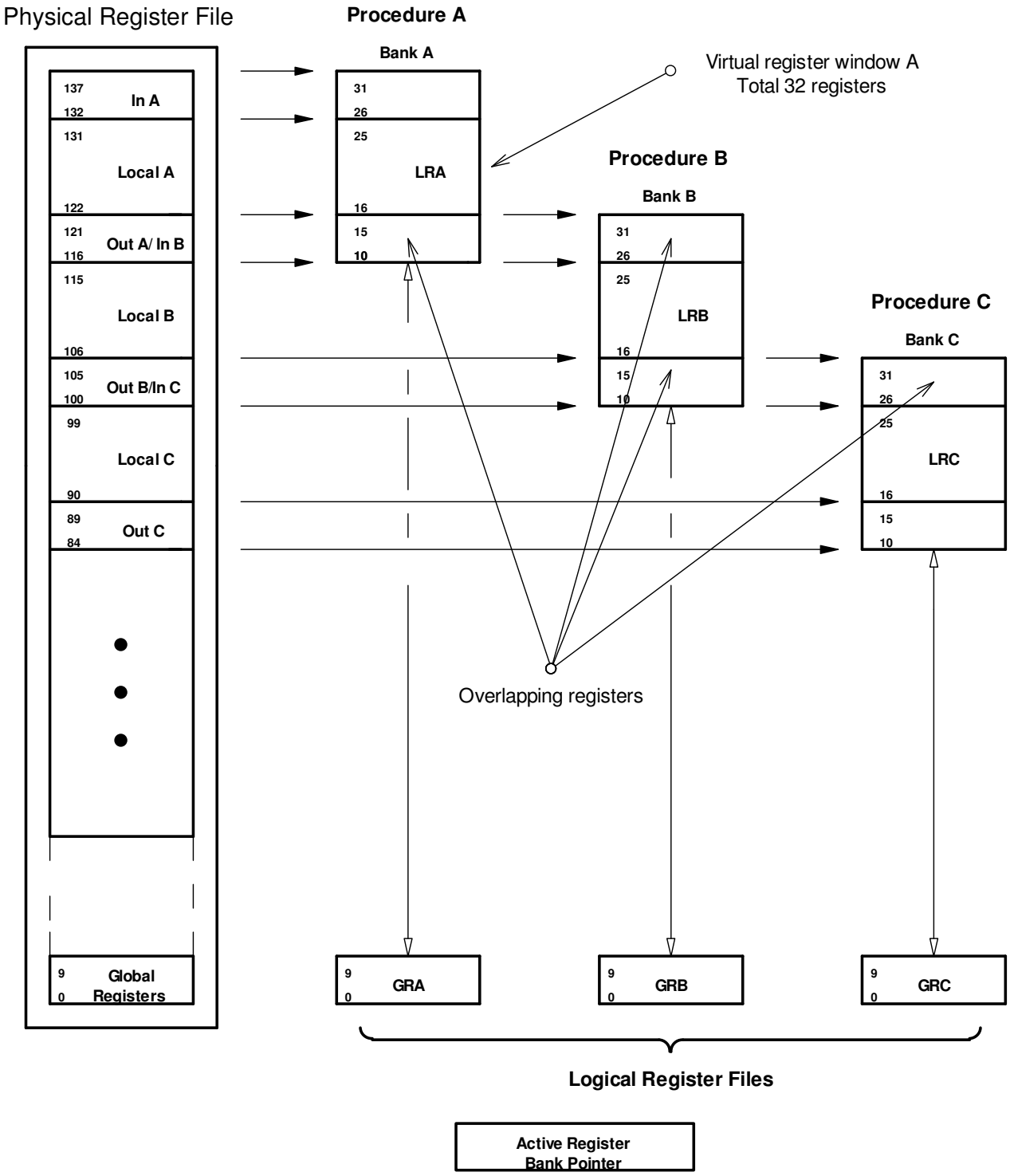
The general RISC instruction format



14. Exceptions (interrupts) have a great influence to the RISC processor pipeline's productivity. The most common kind of software-visible exception on one of the classic RISC processor is a TLB miss. Exceptions are different from branches and jumps, because branches and jumps which change a control flow are resolved in the decode stage.

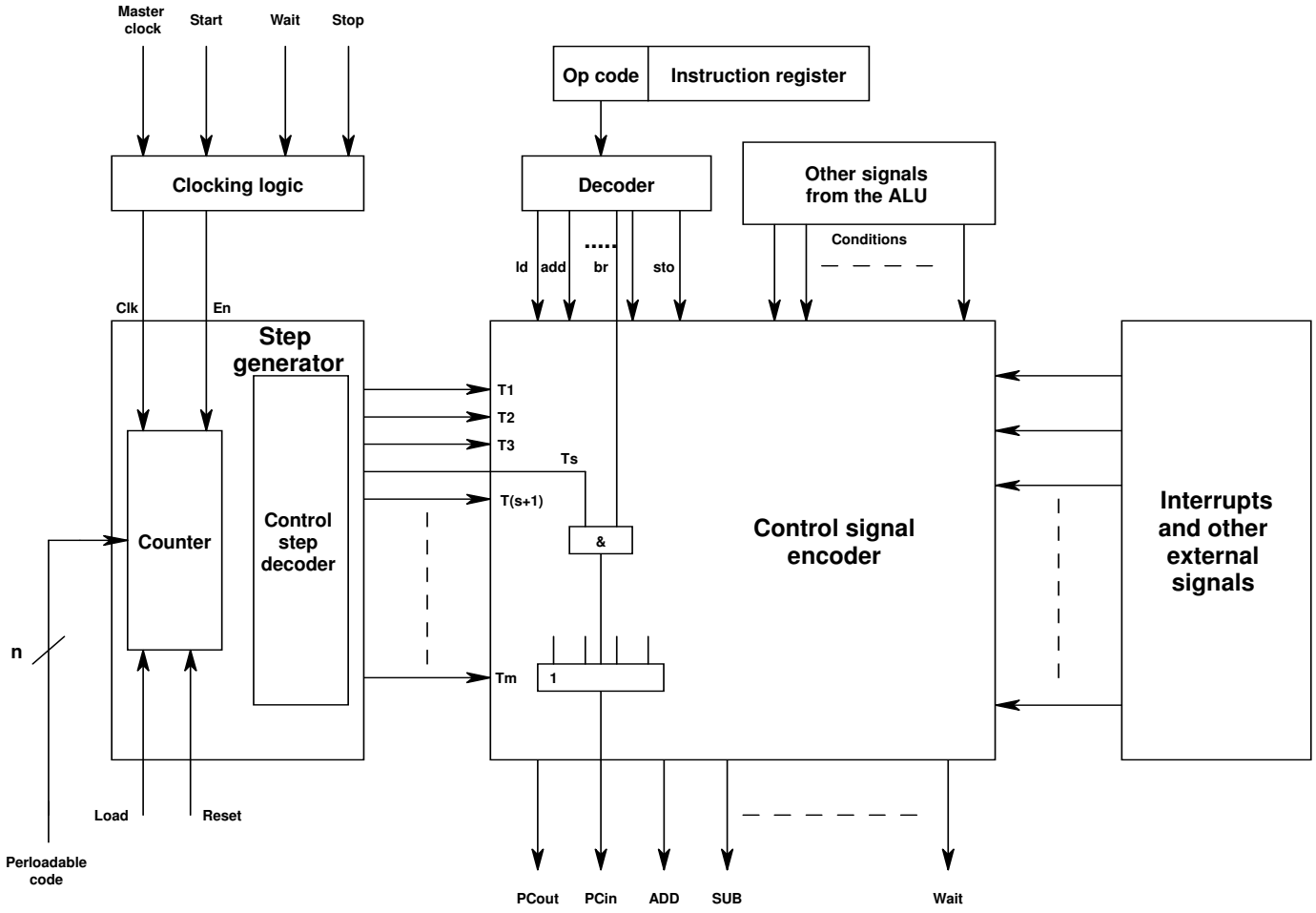
Exceptions are resolved in the writeback stage.

When an exception is detected, the following instructions in the pipeline are marked as invalid, and as they flow to the end of the pipe their results are discarded. The program counter is set to the address of a special exception handler.
The exceptions handling is based on the **precise exception model**.
A precise exception means that all instructions up to the excepting instruction have been executed, and the excepting instruction and everything afterwards have not been executed.



Register file organization (register windows) in RISC II microprocessor

Hardwired Contol Unit's Model



Generated control signals to the different functional units of CPU

The First Generation RISC Microprocessors

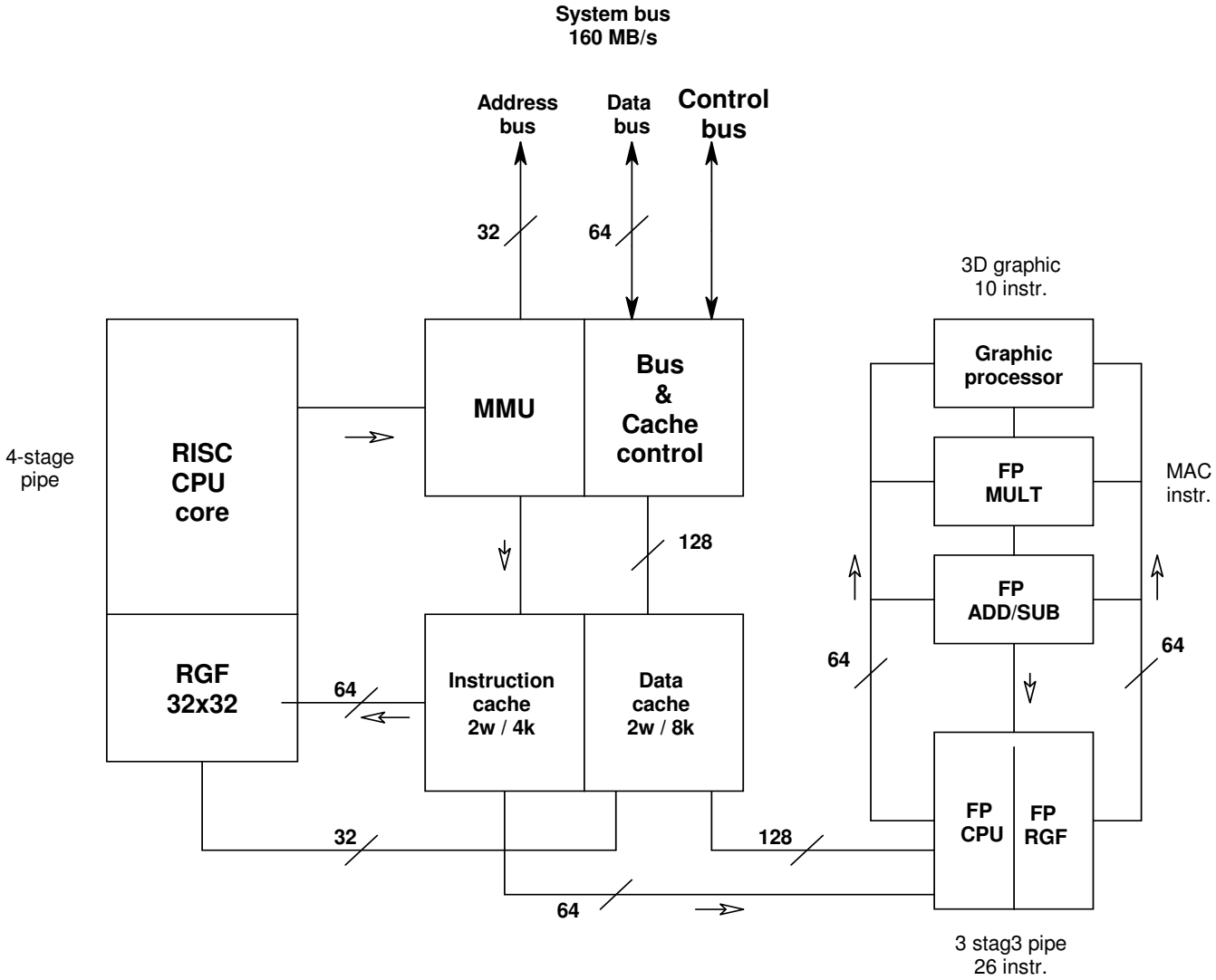
Model	Instructions	Addressing modes	Different types of instructions	Register file size
Experimental				
RISC II	39	2	2	138
MIPS	31	2	4	16
(IBM 801)	120	3	2	32
Industrial				
Acorn ARM	44	2	6	16
IBM ROMP	118	2	2	16
HP 3000/930	140	2	2	32
Motorola 88100	51	3 (4*)	2	32

The Second Generation RISC Microprocessors

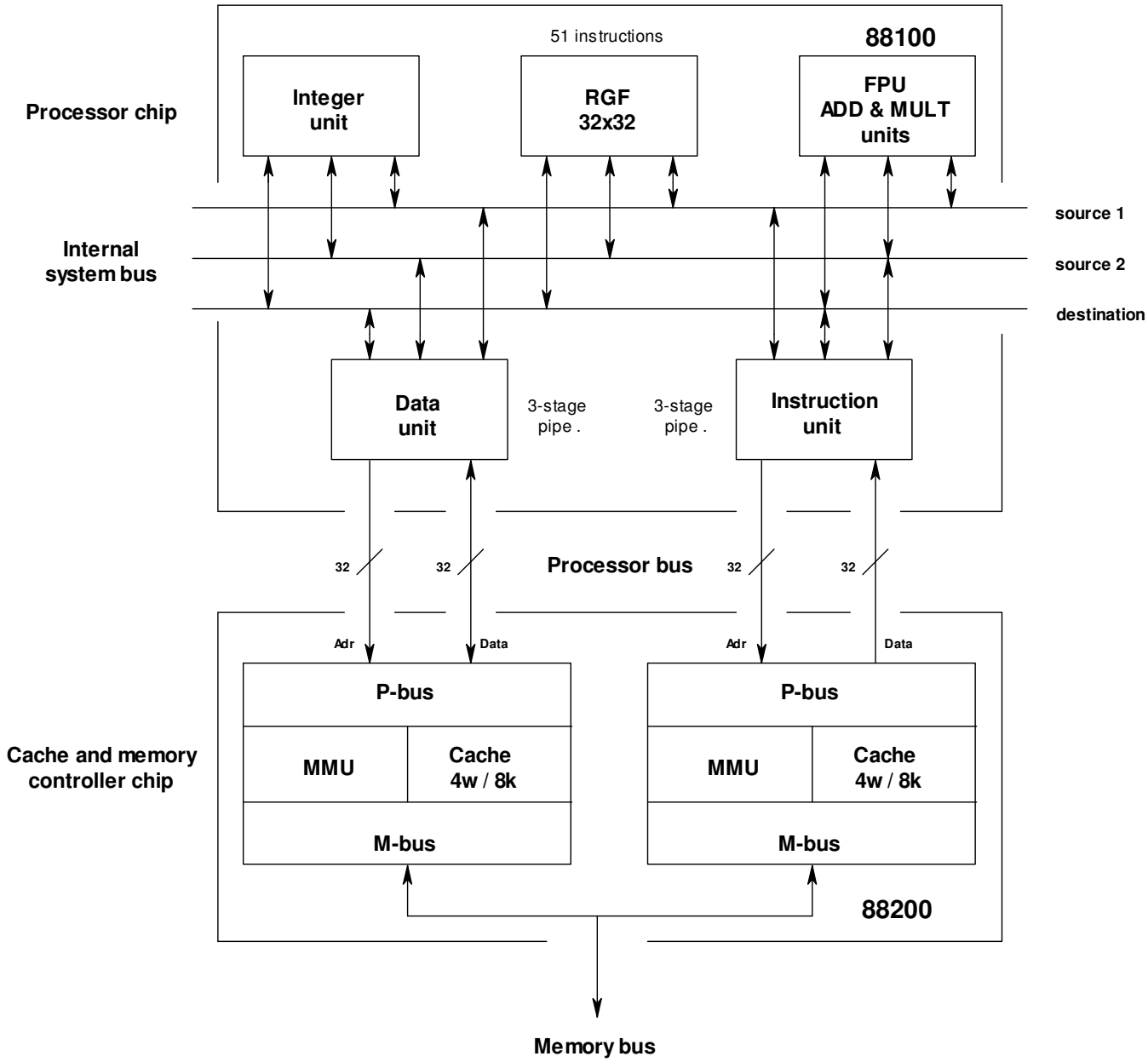
	MC88000	**i80860**	**SPARC**	**MIPS**
CPU	MC88100	80860	CY7C601	R3000
FPU	on CPU	on CPU	CY7C609	R3010
Clock (MHz)	20	40	33	25
Registers in CPU	32	32	136	32
Cache	MC88200 IC-16 kB DC-16 kB	on CPU IC-4 kB DC- 8 kB	CY7C604 64 kB	external IC-256 kB DC-256 kB
MMU	MC88200	on CPU	CY7C604	on CPU
CPU bus bandwidth (MB/s)	160	960	133	200
Memory bandwidth (MB/s)	64	160	266	100
The instructions number	51	65	89	74
Peak throughput (VAX-MIPS)	17	33	24	20
Technology (CMOS)	1,2	1,0	0,8	1,2
Cost per MIPS	65	23	53	10

Intel 80860

(1989)



Motorola 88000



Processor chip 88110

1. **88100**
2. **88200**
3. **3D graphics functions**

The Third Generation RISC Microprocessors

	Alpha 21066	**Power PC 601**	**Super SPARC+**	**PA-7150**	**R4400**
Firm	Digital	IBM & Motorola	SUN & TI	Hewlett-Packard	Toshiba
Clock (MHz)	166	80	60	125	150
Pipe stages	7	5	4	5	8
Cache (kB)	IC – 8 DC – 8	32	IC – 20 DC – 16		IC – 16 DC – 16
Transistors ($x10^6$)	1,747	2,8	3,1	0,85	2,3
Technology	0,68 CMOS	0,60 CMOS	0,72 BiCMOS	0,8 CMOS	0,6 CMOS
Chip area (mm^2)	209	121	256	196	184
Performance **SPEC int92** **SPEC fp92**	 70 105	 70 105	 77 98	 135 200	 96 105

RISC *versus* CISC

	Advantages	Disadvantages
CISC	**1.** Microprogramming is less expansive than hardwiring a control unit. **2.** The ease of microcoding instructions allows making CISC computers upwardly compatible. **3.** Fewer instructions could be used to implement a given task. **4.** More efficient use of the relatively slow main memory. **5.** The compiler does not have to be as complicated.	**1.** Instruction set and chip hardware become more complex with each generation of computers. **2.** Individual instructions could be of almost length. Different instructions will take different amounts of clock time to execute, slowing down the systems overall performance. **3.** Many specialized instructions aren't used frequently.
RISC	**1.** High speed. RISC processors achieve 2 to 4 times performance of CISC processors using comparable technology and the same clock rates. **2.** Simpler hardware. Less chip space is used, so extra functions (MMU, FPU, etc) can be placed on the same chip. **3.** Shorter design cycle.	**1.** Code quality. The processor's performance depends greatly on the code that is executed. **2.** Code expansion increases the needed main memory amount. **3.** The specialized compilers are needed.

In recent years, the RISC *versus* CISC controversy has died down to great extent.

RISC Problem

Today superscalar micros can issue 4 – 5 instructions per cycle, but execute only 1,5 – 2 instructions per cycle.

Cause

1. The code and data are not on-chip when needed.
2. A cache miss in a high-speed superscalar RISC is a real performance downer.

Some potential solutions

1. **Data prefetching**,
2. **Branch prediction**,
3. **Speculative execution**,
4. **Large, multilevel cache-system**,
5. **Smart memory controllers**,
6. **Multiple thread execution**,
7. **Code optimization, code preprocessing**,

Another approach to RISCs was the **zero-address instruction set** (the majority of space in an instruction was to identify the operands of the instruction).
In zero-address machines the **operands are placed into** a **push-down** (**LIFO-type**) **stack**.

Summary

One important development in architecture has been the focus on architecture's role in efficiently utilizing the underlying physical technology. The RISC breakthrough is a key example of this idea. RISC architects recognized that by scaling the complexity and cost of microprocessor architecture down to that point of which an effective pipeline could be implemented a single IC, a large increase in performance was possible.
They recognized that a good match between architecture and the implementation technology is crucial to achieving maximum performance and cost efficiency.
Today's RISC computers are typically complicated and feature-laden relative to the original RISC implementations, but they still reflect careful trade-offs by designers between architectural complexity, implementation cost and clock speed.

Literature

1. Arvo Toomsalu. RISC-mikroprotsessorite arhitektuur. Tallinn, 1995.
2. IEEE Standard 1754-1999. IEEE Standard for a 32-bit Microprocessor Architecture.
   [The standard is derived from SPARC, which was formulated at Sun Microsystems in 1985. IEEE 1754 is designed to be a target for optimizing compilers and easily pipelined hardware implementations.]

MODERN SUPERSCALAR ARCHITECTURE

CISC philosophy

- If added hardware can result in an overall increase in speed, it's good.
- The ultimate goal of mapping every HLL statement on to a single CPU construction**.**

RISC philosophy

- Whether hardware is necessary, or whether it can be replaced by software.
- Higher instruction bandwidth is usually offset by a simpler chip that can run at a higher clock speed, and more available optimizations for the compiler.

Modern or Post-RISC architecture characteristics

- The most significant Post-RISC changes are met in the architecture of a CPU.
- Superscalar post-RISC processors relied on the compiler to order instructions for maximum performance and hardware checked the legality of multiple simultaneous instruction issue.
- The post-RISC processors are much more aggressive at issuing instructions using hardware to dynamically perform the instruction reordering.
- These processors find more parallelism by executing out of program order.

Modern Superscalar Processors' Main Features

1. Speculative Execution

In a pipelined processor, branch instructions in the execute stage affect the instruction fetch stage. Advanced speculative techniques are used, as for branch predictions, data prediction, address prediction, memory dependencies and latencies prediction. To preserve program semantics, a speculative movement of instructions should only result in speculative execution that is safe and legal.

Conventional superscalar processor employ the strong-dependence model *//jäik sõltuvuste mudel//* for program execution. In the strong dependence model two instructions are identified as either dependent or independent, dependences are pessimistically assumed to exist.

To superspeculative processors the weak-dependence model *//lõtv sõltuvuste mudel//* is applied. In this case dependences can be temporarily violated during instruction execution as long as recovery can be performed prior to affecting the permanent machine state.

Data Value Speculation

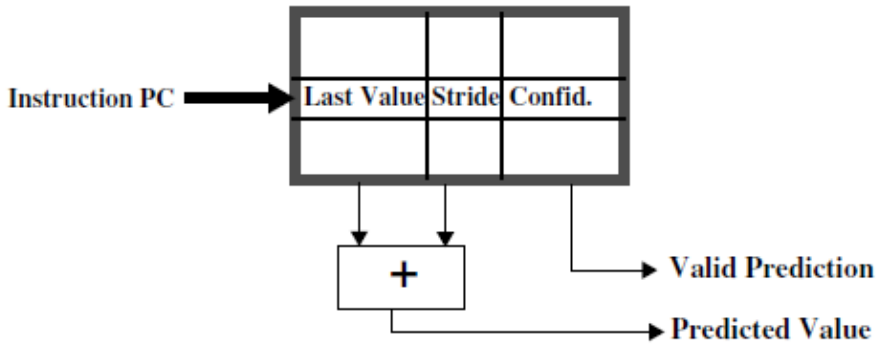
The term *data value speculation* refers to mechanisms that predict the operands of an instruction, either source or destination, and execute speculatively the instructions dependent on it before the

actual value is computed, allowing the processor to avoid the ordering imposed by data dependences.

Example

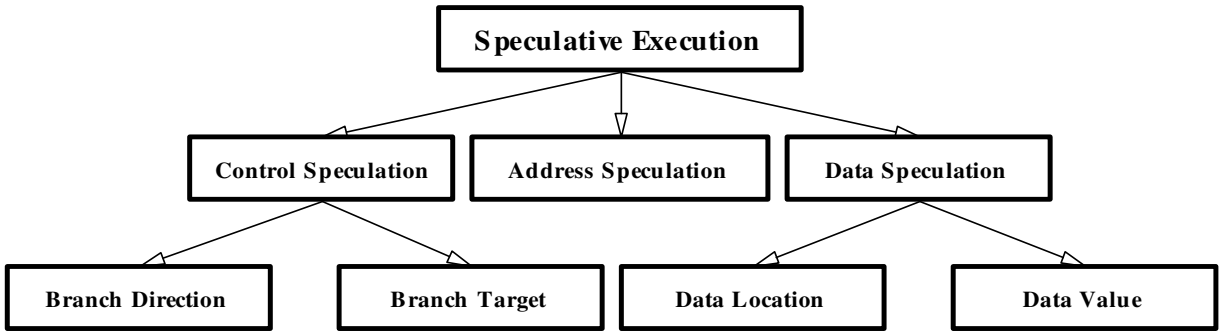
A stride-based data predictor //*indeksisammupõhine andmeennusti*// is implemented by means of a table that is direct-mapped, non-tagged and it is indexed with the least significant bits of the instruction address whose source or destination operands are to be predicted.
Each table entry stores the following information:

1. Last data value
2. Stride
3. Confidence



Predictor for arithmetic instructions stores the last result in the last value field.
Load address predictors store the last effective address.
Load value predictors store the last value read from memory.
Store predictor uses two tables: one for predicting the effective address and the other for predicting the value to be written.

When an instruction is to be predicted, the prediction table is accessed and the predicted value is computed adding the stride to the previous last value. If the most significant bit of the confidence field is set and the prediction is correct, the predicted value can be used instead of the actual value if the former is available earlier.



2. Instructions Predecoding

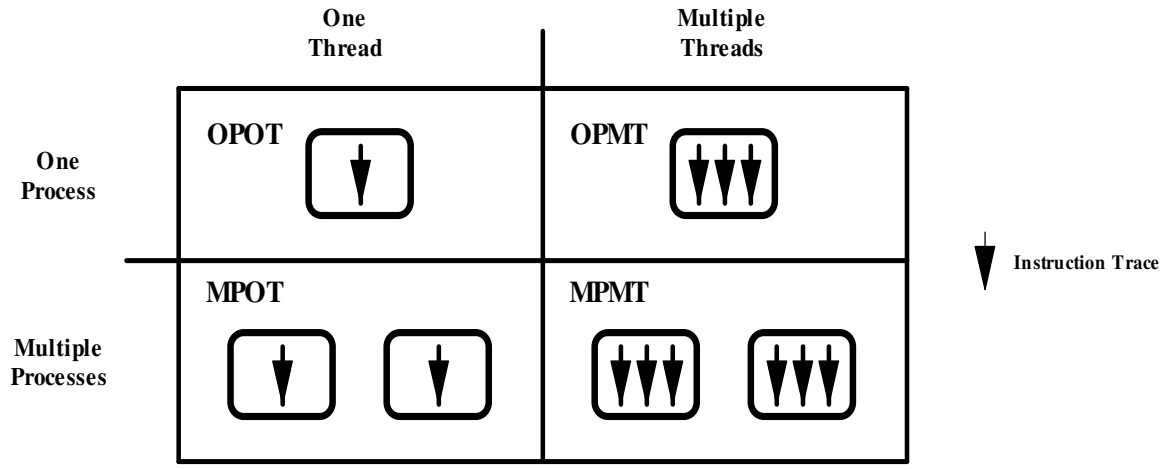
Instruction decoding involves the identification of the instruction types and detection of instructions dependences among the group of instructions that have been fetched but not yet dispatched.
The complexity of the instruction decoding task is influenced by the ISA and the width of the parallel pipeline.

- For a RISC scalar pipeline, instruction decoding is trivial. Usually decode stage is used for accessing the register operands and is merged with the register read stage.
- For a RISC parallel pipeline with multiple instructions being simultaneously decoded, the decode stage must identify dependences between instructions and determine independent instructions that can be dispatched parallel.
- For a CISC parallel pipeline the instruction decoding stage is more complex and usually requires multiple pipeline stages.
- The modern CISC pipelines have an additional task – the decoder must translate the architected instructions into internal low-level operations that can be directly executed by the HW.

These internal operations resemble RISC instructions and can be viewed as vertical microinstructions.

3. Threading

In virtual memory systems is distinguished between multitasking of processes and threads.
Thread **//lõim, haru//** - a sequential execution stream within a task (process).
A thread state is entirely stored in the CPU, while a process includes extra state information – mainly operating system support to protect processes from unexpected and unwanted interferences. Threads are sometimes called lightweight processes. Switching between threads does not involve as much overhead as conventional processes.



4. Multithreading

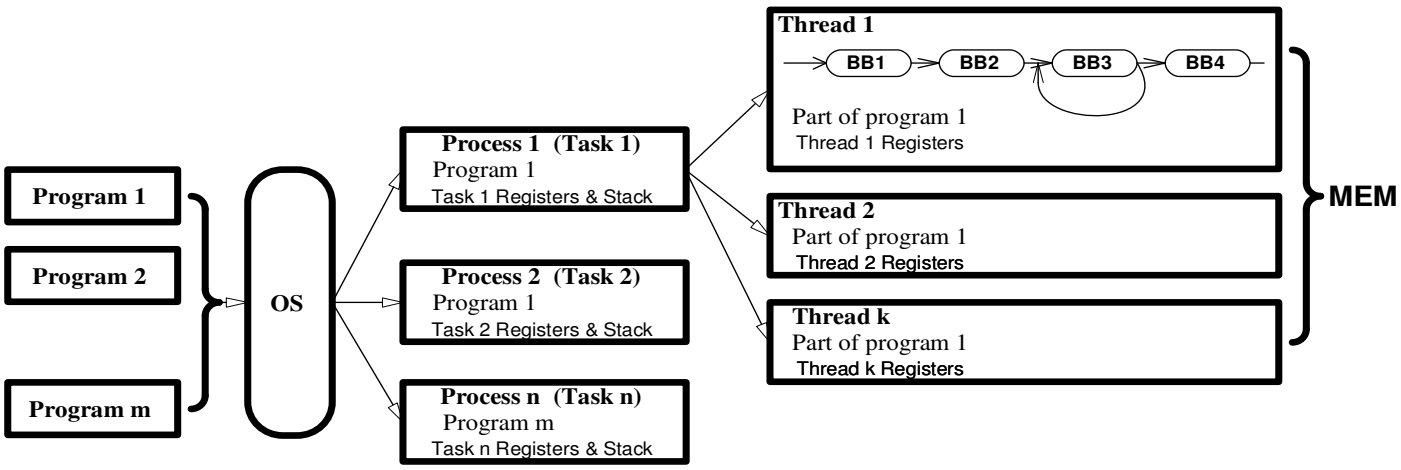
Contemporary superscalar microprocessors are able to issue 4-6 instructions at each clock cycle from a conventional linear instruction stream. New technology will allow microprocessors with an issue bandwidth of 8-32 instructions per cycle.
As the issue rate of microprocessors increases, the compiler or the hardware will have to extract more instruction-level parallelism from a sequential program.

ILP found in conventional instruction stream is limited.

Most techniques for increasing performance increase power consumption.
The multithreaded approach may be easier to design and have higer power efficiency, but its utility is limited to specific – highly parallel-applications.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. It improves the overall throughput of the processor.

Multithreading exploits a different set of solutions by utilizing coarse-grained parallelism.
A multithread processor is able to concurrently execute instructions of different threads of control within a single pipeline. Multithreading replicates only the program counter (PC) and register file, but not the number of execution units and memories.
Multithreading effectively divides the resources of a single processor into two or more logical processors, with each logical processor executing instructions from a single thread. It means that the processor must provide:
a) Two or more independent program counters,
b) An internal tagging mechanism to distinguish instructions of different threads within the pipeline
c) A mechanism that triggers a thread switch.



MULTITASKING and MULTITHREADING
{Program, Process (Task), Thread, Basic Block}

In explicit multithreaded processor //*ilmutatud lõim- e hargtöötlusega protsessor*// is used a coarce-grained parallelism. Multiple program counters are available in the fetch unit and the multiple contexts are often stored in different register sets. The execution units are multiplexed between the thread contexts.

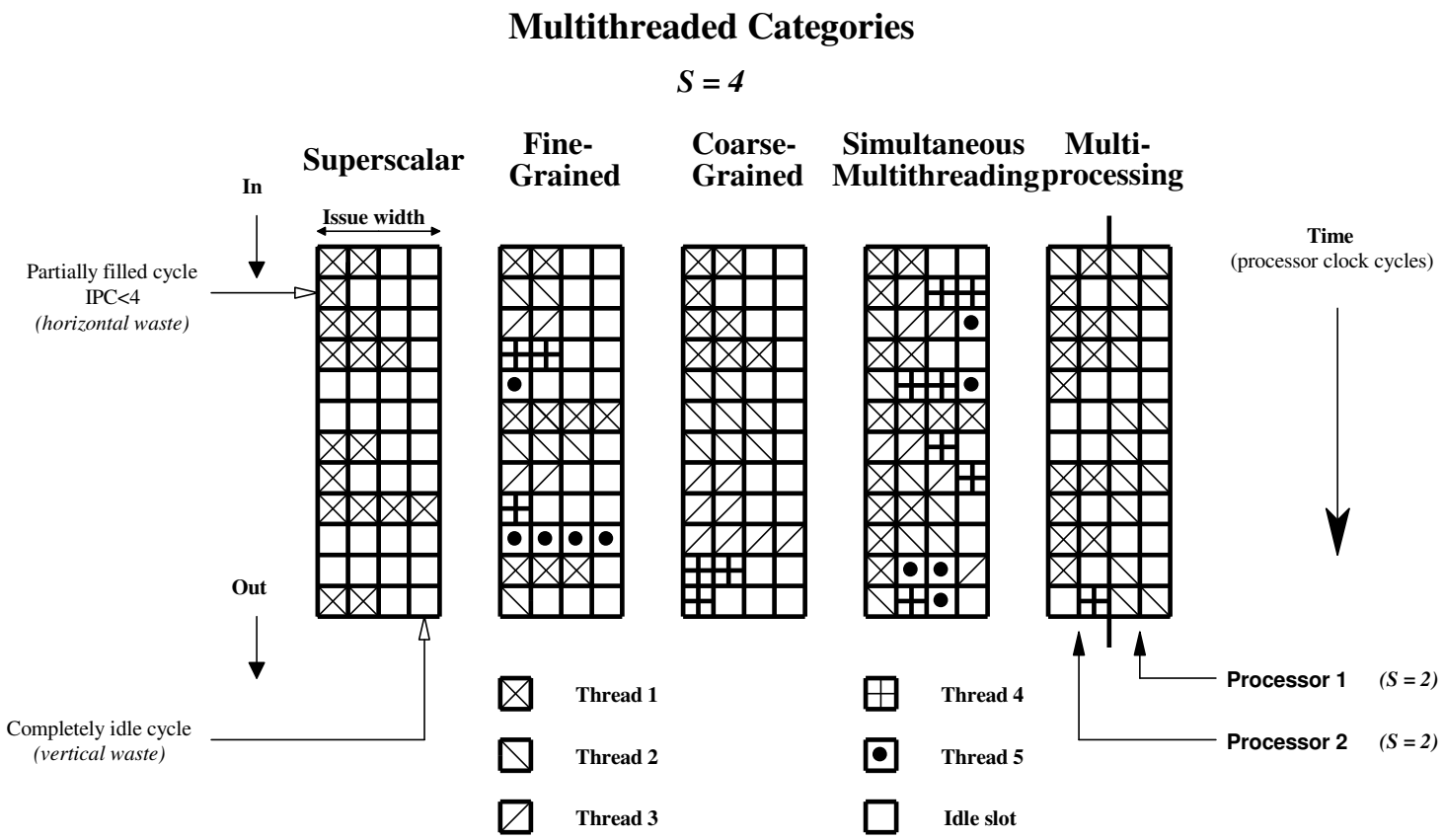
Explicit multithreading techniques are:

1. Interleaved multithreading (IMT)
An instruction of another thread is fetched and fed into the execution pipeline at each processor cycle ($\Rightarrow$ barrel processor).
2. Simultaneous multithreading (SMT)
Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor.
3. Blocked multithreading (BMT)
The instructions of a thread are executed successively until an event occurs that may cause latency. This event induces a context switch.

In implicit multithreaded processors //ilmutamata lõimtöötlusega protsessor// is the thread-level speculation applied. These processors can dynamically generate threads from single-threaded programs and execute such speculative threads concurrent with the lead thread.
In case of misspeculation all speculatively generated results must be squashed.

Threads generated by implicit multithreaded processors are always executed speculatively,
in contrast to the threads in explicit multithreaded processors.

Simultaneous multithreading allows multiple threads to execute different instructions in the same clock cycle. In the SMT processor the multithreading technique is combined with a wide-issue superscalar processor. The number of concurrent threads has practical restrictions and usually limits the number from 2 to 8 concurrent threads.



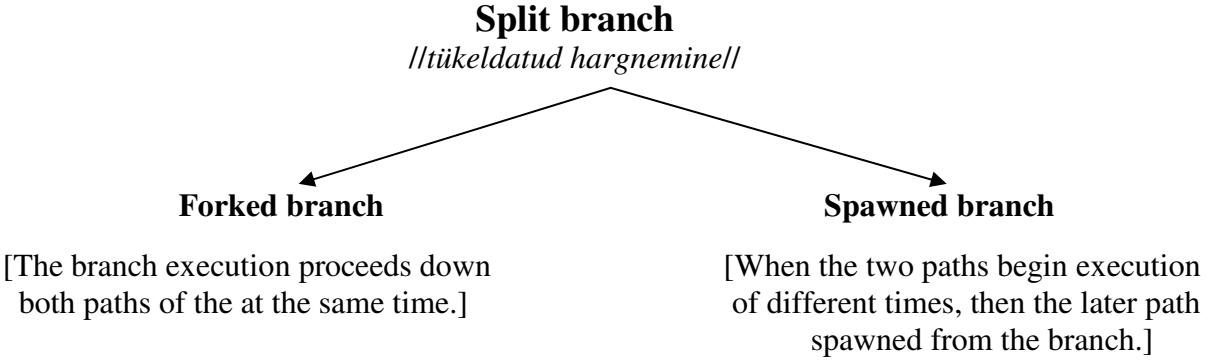
5. Predicated Instructions

The instructions which are executed only if conditions are true, bits in a condition code register have an appropriate value. This eliminates some branches, and in a superscalar processor can allow both branches in certain conditions to be executed in parallel, and the incorrect one discarded with no branch penalty.

6. Branch Prediction

All pipelined processors do branch prediction of some form. There are many different branch prediction methods in use - trivial prediction, next line prediction, bimodal branch prediction, local branch prediction, global branch prediction, combined branch prediction, agree prediction, overriding branch prediction.

Branch path – consist of the dynamic code between two conditional branches with
//hargnemistee// no intervening unconditional branches.



The earliest and simplest form of multipath execution was Y-pipe.

Multipath execution

Multipath execution is the execution of code down both paths of one or more branches, discarding incorrect results when a branch is resolved.
In typical speculative execution a branch's final state, taken or not taken, is predicted when the branch is encountered.
Multipath execution differs from unipath execution in that both paths of a branch may be speculatively executed simultaneously. Thus, any branch misprediction penalty normally occurring in unipath execution is reduced or eliminated, improving performance.

Speculative code can consist of both multipath and unipath sections, that is, section may proceed down one or both paths of a branch, and this situation may change over time.
Most forms of multipath execution use some form of branch prediction.

7. Prefetching

A technique which attempts to minimize the time a processor spends waiting for instructions to be fetched from memory. Instructions following the one currently being executed are loaded into a prefetch queue when the processor's external bus is otherwise idle. If the processor executes a branch instruction or receives an interrupt then the queue must be flushed and reloaded from the new address.

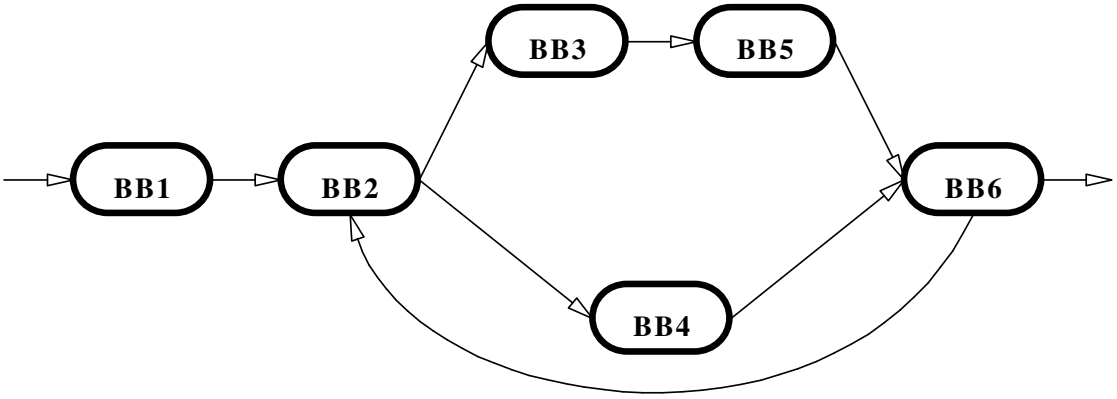
8. Trace Cache

The trace cache is a mechanism for increasing the instruction fetches bandwidth by storing traces of instructions that have already been fetched, and maybe even executed.

The trace cache line is filled as instructions are fetched from the instruction cache.
Each line in the trace cache stores a trace (a snapshot) of the dynamic instruction stream.
The same dynamic sequences of basic blocks that appear non-contiguous in the instruction cache are contiguous in the trace cache. Trace caches are caches that store instructions either after they have been decoded, or as they are retired.

A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream.

Trace lines stored in the trace cache are fully specified by a starting address and a sequence of branch outcomes which describe the path followed.



TRACE CACHE Content

T1= BB1, BB2, BB3, BB5	**T2 = BB2, BB4, BB6, BB2**
T4 = BB1, BB2, BB4, BB6	**T3 = BB6, BB2, BB4, BB6**

BBi - basic block i
Tj - trace j

Program flow and corresponding trace cache contents

In the instruction fetch stage of a pipeline, the current program counter along with a set of branch predictions is checked in the trace cache for a hit. If there is a hit, a trace line is supplied to fetch. The trace cache continues to feed the fetch unit until the trace line ends or until there is a misprediction in the pipeline. Because traces also contain different branch paths, a good multiple branch predictor is essential to the success rate of trace caches.

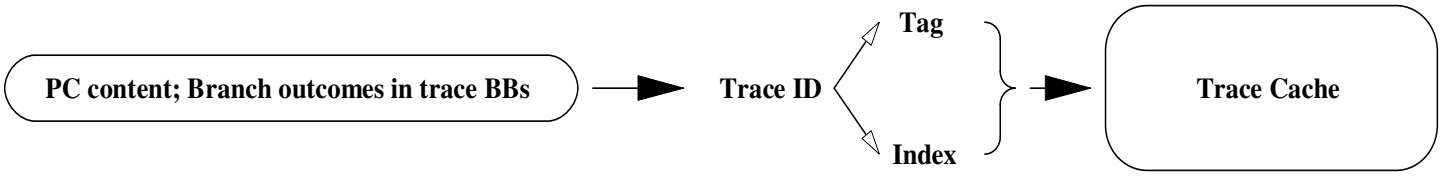
The limit n is the trace cache line size and m is the branch predictor throughput.

An average basic block size is 4-5 instructions.

Trace Cache Main Features

- A trace cache consists of control and data area. The control of each trace cache entry holds a trace tag composed out of its starting address, number of basic blocks and branch outcome of each block.

- The start PC and branch outcomes are collectively called the trace identifier or trace ID. Looking up a trace in the trace cache is similar to looking up instructions or data in conventional caches, except the trace ID is used instead of an address.
- A subset of the trace ID forms an index into the trace cache and the remaining bits form a tag.



- One or more traces and their identifying tags are accessed at that index. If one of the tags matches the tag of the supplied trace ID, there is a trace cache hit. Otherwise, there is a trace cache miss.
- New traces are constructed and written into the trace cache either speculatively or non-speculatively.
- A new predicated trace ID is supplied to the trace cache each cycle.
- In addition to typical parameters such as size, set-associativity and replacement policy, the trace cache design includes:

  1. Indexing methods;
  2. Path associativity;
  3. Partial matching;
  4. Trace selection;
  5. Trace cache fill policy;
  6. Parallel or sequential accessing of the trace, etc.

- A problem of trace caches is that they necessarily use storage less effectively than instruction caches.
- Longer traces improve trace prediction accuracy.
- Overall performance is not as sensitive to trace cache size and associativity.

The* front-end *architecture is critical to the performance of a processor as a whole, as the processor can only execute instructions as fast as the front –end can supply them.

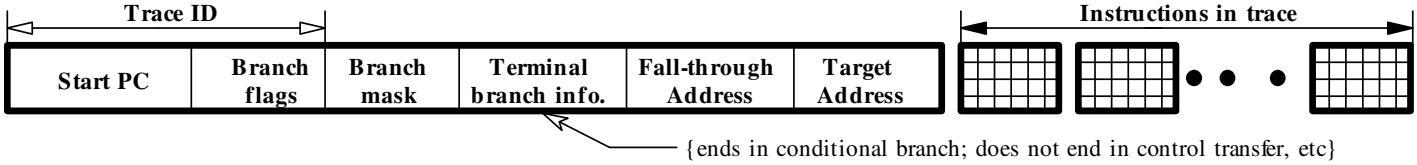
Trace Cache Model

The trace cache fetch mechanism consists of the trace cache (TC), the fill unit, a trace predictor, and a conventional instruction cache with a branch predictor.

The trace predictor treats threads as basic units and explicitly predicts sequences of traces.
The output of the trace predictor is an identifier (ID) indicating a starting instruction address of a trace. The identifier is used to index the trace cache and the trace is provided if the access is hit.
The index into the trace cache can be derived from the starting PC (program counter state), or a combination of PC and branch outcomes.
When a trace cache miss occurs, the instruction cache works as a backup and supplies instructions with the help of the branch predictor.

The fill unit collects the traces which are stored in the trace cache. It constructs the trace using the instructions supplied from the instruction cache, and updates the trace cache according to branch outcomes provided by the execution engine.
Each trace is dispatched to both the execution engine and an outstanding trace buffer (OSTB).
A trace line begins with a target and ends with a control transfer instruction.

Content of Trace Line



Each bit in the branch flag encodes whether or not the corresponding embedded branch is taken.
The branch mask is a sequence of leading 1`s followed by railing 0`s. Only internal branches and not a terminal branch distinguish the trace and considered by the hit logic.
If the maximum branch limit is m branches overall, then the maximum number of embedded branches in trace is no more than (m-1). There are (m-1) branch flags, and the branch mask is (m-1) bits long.

The b^{th} branch is predicted by branch predictor terminates a trace.

The branch mask is used by the hit logic to compare the correct number of branch predictions with branch flags, according to the actual number of branches within the trace.

The branch mask determines how many of the predictions supplied by the multiple-branch predictor are actually “considered” by the matching procedure.

If the terminal instruction is a conditional branch, then the two address fields are equal to the branch’s fall-through and target address respectively.
If the terminal branch is predicted taken, then the next PC is set equal to the trace`s target address.
If the terminal branch is predicted not-taken, then the next PC is set equal to the trace`s fall-through address.
The trace construction finishes when:

1. The trace contains n instructions;
2. The trace contains m conditional branches;
3. The trace contains a single indirect jump, return or trap instruction;
4. Merging the incoming block would result in a segment larger than n instructions.

A basic block cannot be split across two traces.

The performance of the TC is strongly dependent of trace selection, the algorithm used to divide the dynamic instruction stream into traces.

Trace Cache Types

1. **CTC** – conventional TC;
2. **STC** – sequential TC;
3. **BBTC** – block-based TC.

Full and Partial TC Hit

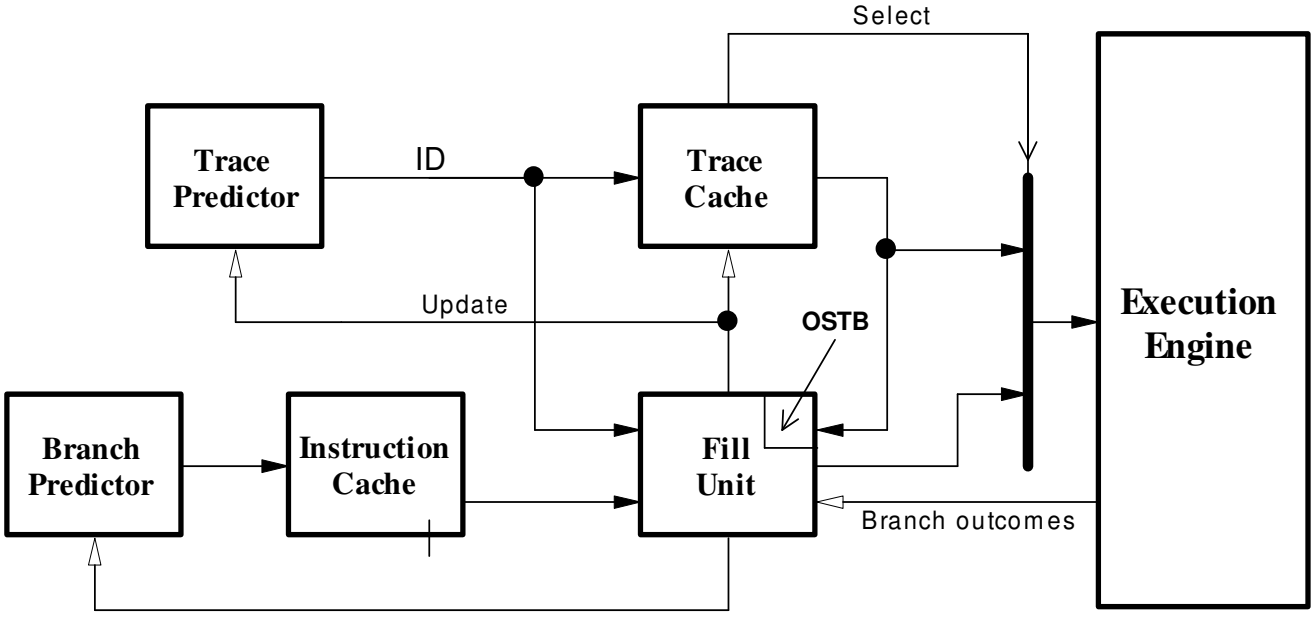
Sometimes partial trace cache hit occurs. When such event occurs, either the inter-trace prediction is incorrect or at least one of the intra-trace predictions is incorrect.

At least one of the branches in the trace results in a miss or the entire trace is invalid.

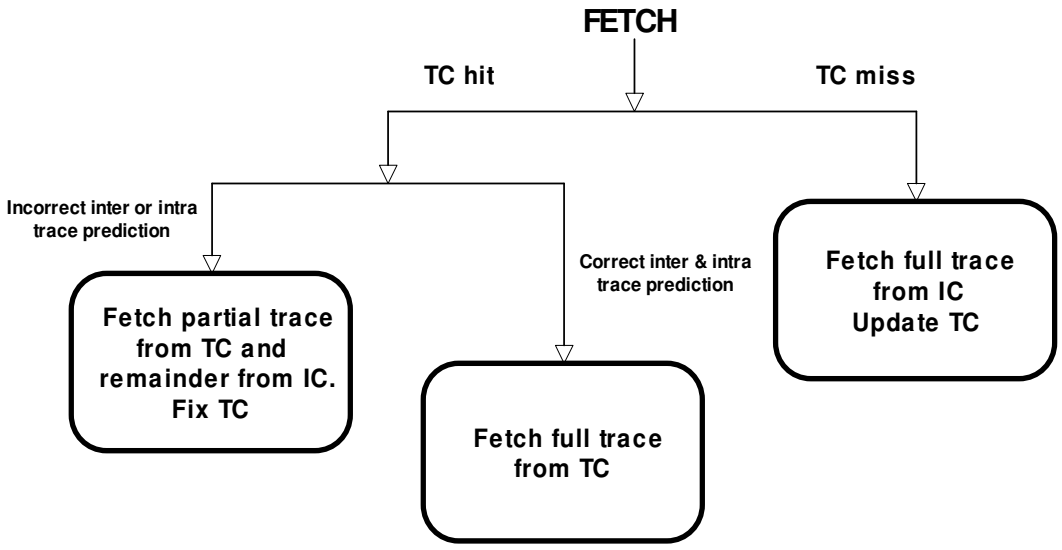
The CTC has the same instruction fetch organization as the STC, except that the backing instruction cache and the trace cache are probed in parallel.

Trace Cache Metrics - **Hit rate**, **Fragmentation**, **Duplication** and **Efficency**.

STC and CTC Model



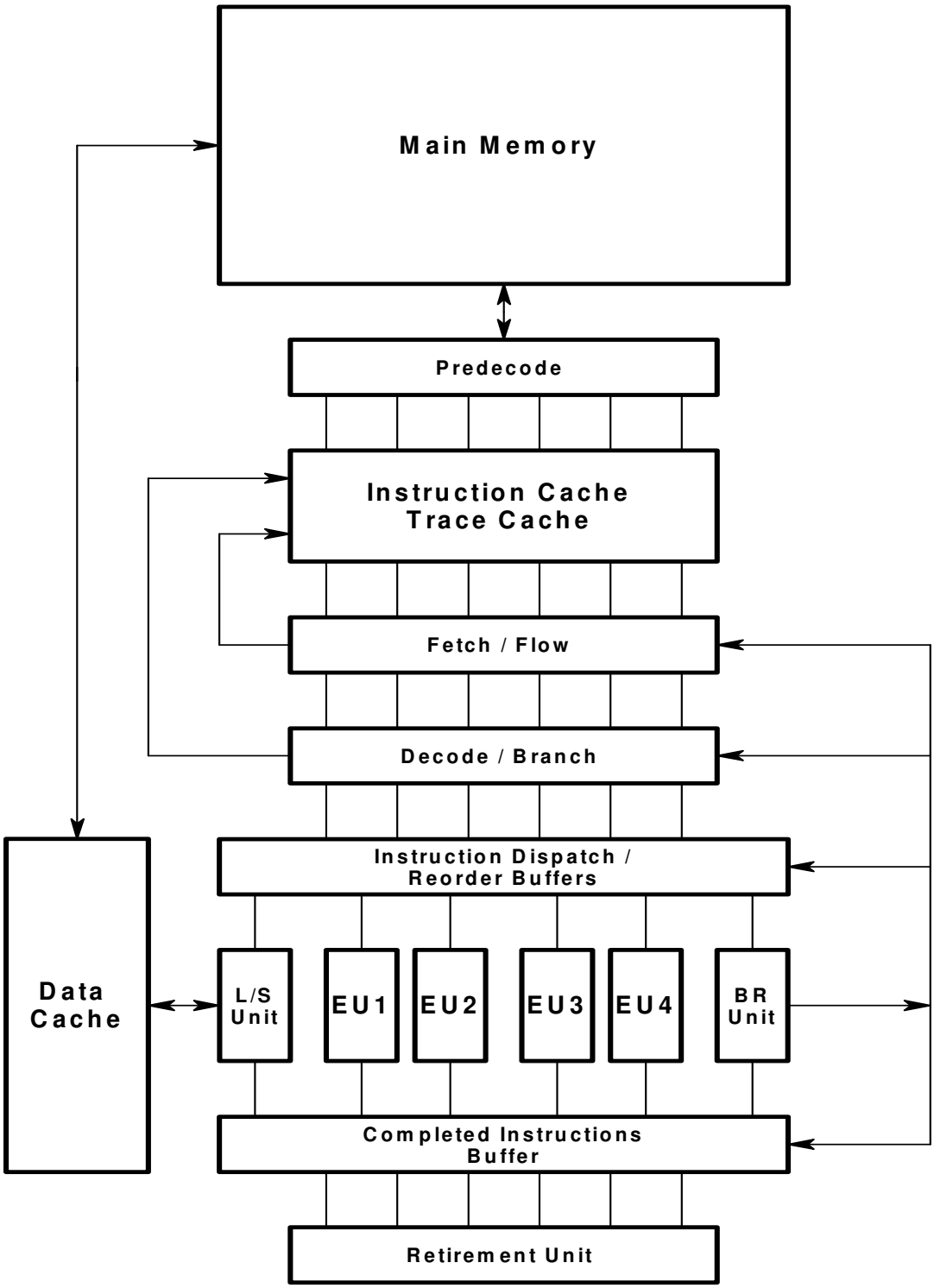
Trace Cache Fetch Mechanism



Instruction Fetch Models

Example

Model of the Modern 6-way Superscalar Processor



Fetch / Flow Prediction

1. The branch prediction is a part of the decode operation.
This information is available to the instruction fetch unit 1-3 cycles too late.
The Instruction Fetch Unit must know the next address to fetch before a decode phase starts.
To allow the Instruction Fetch Unit to fetch the next instruction, without waiting for decode phase and branch lookup, flow history table bits are added to each group of six instructions.

Decode / Branch

1. The instruction is decoded and more accurate branch prediction is performed.
2. At each branch instruction, the flow of the program diverges into two separate instruction streams. The CPU makes a prediction based upon certain heuristics and the past behavior of that instruction.

Instruction Dispatch and Reorder Buffer

1. The instructions are queued waiting to be dispatched.
2. Instructions are ready to be dispatched when:

- Their input values are available,
- Their output register is available,
- An execution unit is available.

3. Dispatched, but not ready, instructions are buffered into appropriate Reservation Stations.
4. Rename Registers hold results until the instruction retires.
5. At retirement, the Rename Registers are either copied to the architectural register.
6. When a branch is resolved, all of the Rename Registers allocated for wrong path are freed, and only instructions on the correct path are allowed to retire.

Execution Units

1. Functional Units classification:

a. Single-cycle.
b. Multiple-cycle:
1 Multi-cycle units that are pipelined
2 Multi-cycle units that are pipelined but do not accept a new operation each cycle
3. Multi-cycle units that are not pipelined
c. Variable cycle time units

2. The Branch Unit communicates the results of a branch evaluation to the other units:

- The Fetch/ Flow Unit must be *informed* of mispredictions so that it can update the flow history in the instruction cache and begin fetching at the correct target address;
- The Branch/Decode Unit must be *informed* so that it updates the branch history table.
- The Instruction Dispatch and Reorder Buffer must be *informed* so that it can discard any instruction that should not be executed.

- The Completed Instruction Buffer must be *informed* to discard instructions, which will never be retired.
- If there are multi-cycle Execution Units working on instructions that will never be executed because of a branch, they also must be notified.

3. **Completed Instruction Buffer** and **Retire Unit**

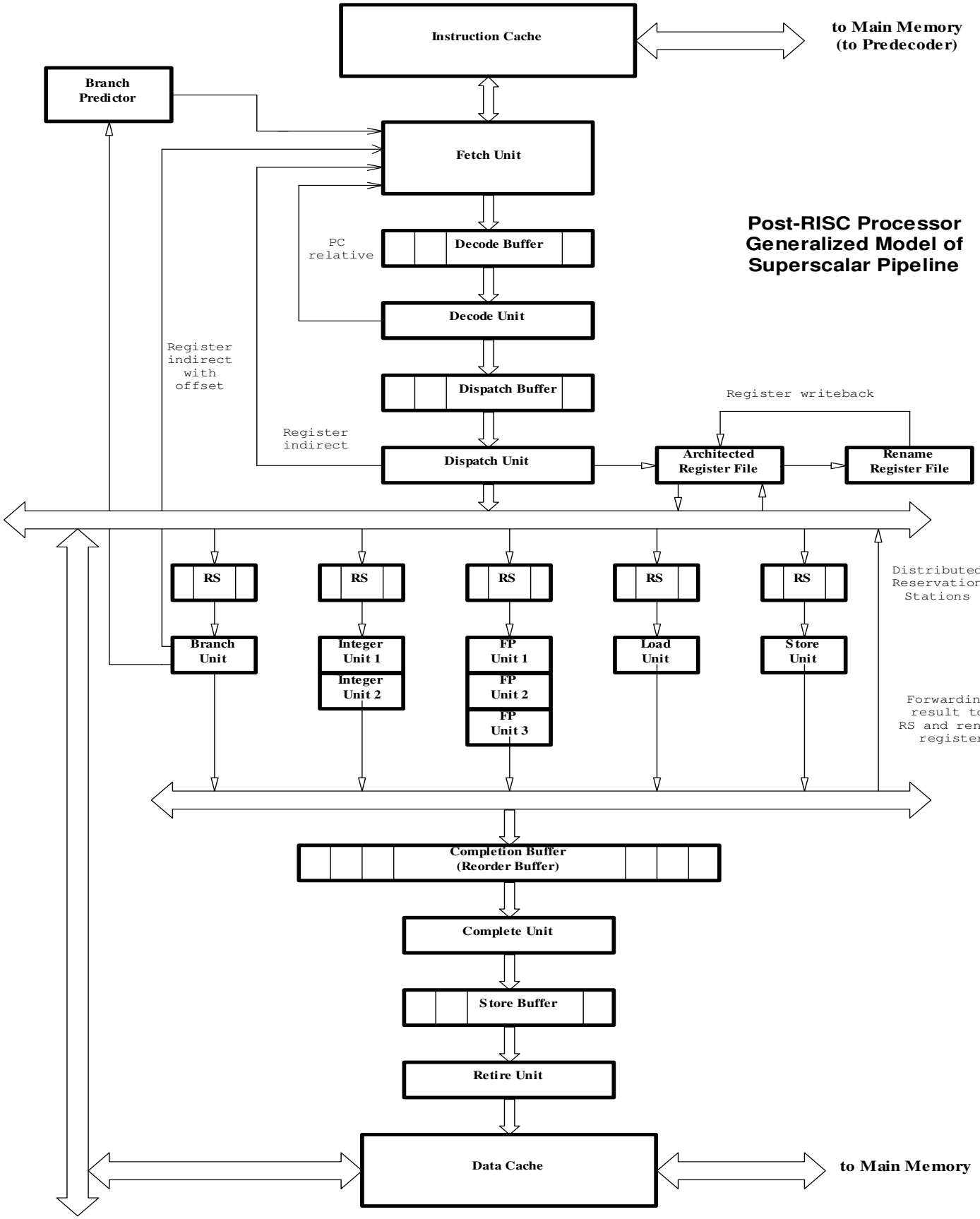
1. The Completed Instruction Unit holds instructions that have been speculatively executed.
2. The Retire Unit removes executed instructions in program order from the Completed Instruction Buffer.
3. The Retire Unit updates the architectural registers with the computed results from the Rename Registers.
4. By retiring instructions in order, this stage can maintain precise exception.
5. In post-RISC architecture the important performance metric is the number of instructions retired in parallel.

- Next step is Advanced Super Scalar Processors, which shall achieve instruction issue from 16 up to 32 instructions per cycle.
- Barrel processor is a CPU that switches between threads of execution on every cycle.

Summary

Multiple-Issue Architectures

	Issue structure	**Hazard detection**	**Scheduling**	**Characteristics**	**Examples**
Superscalar static	Dynamic	Hardware	Static	In-order execution	*Sun Ultra SPARC II / III*
Superscalar dynamic	Dynamic	Hardware	Dynamic	Some out-of-order execution	*IBM Power 2*
Superscalar speculative	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	*Pentium III Pentium IV R 10000 Alpha 21264*
VLIW/LIW	Static	Software	Static	No hazards between issue packets	*Trimedia i860*
EPIC	Mostly static	Mostly software	Mostly static	Explicit dependen-cies marked by compiler	*Itanium*


Instruction Cache
to Main Memory
(to Predecoder)
Branch
Predictor
Fetch Unit
Post-RISC Processor
Generalized Model of
Superscalar Pipeline
PC
relative
Decode Buffer
Decode Unit
Register
indirect
with
offset
Dispatch Buffer
Register writeback
Register
indirect
Dispatch Unit
Architected
Register File
Rename
Register File
RS
RS
RS
RS
RS
Distributed
Reservation
Stations
Branch
Unit
Integer
Unit 1
Integer
Unit 2
FP
Unit 1
FP
Unit 2
FP
Unit 3
Load
Unit
Store
Unit
Forwarding
result to
RS and rename
registers
Completion Buffer
(Reorder Buffer)
Complete Unit
Store Buffer
Retire Unit
Data Cache
to Main Memory

Summary

1. The performance of superscalar processors is dependent on scheduling instructions so that the utilization of the functional units in the processor is maximized.

2. The number of instructions that can be concurrently scheduled is determined by data and control dependencies. When scheduling in software it is only possible to use static information.

3. Superscalar microprocessors require a window of instructions, which can potentially be scheduled in a clock cycle.

4. For solving these tasks hardware support is needed.

5. Increasing the size of the instructions window increases the probability that functional units can be fully utilized.

6. Without out-of-order execution the instruction window is limited to instructions between the currently executing instructions and the first instruction with a data dependence on the result of an incomplete instruction.

7. Data dependencies can occur via registers or via the memory subsystem.

8. Out-of-order execution by itself limits the instruction window to a single basic block, which averages only about five instructions.

9. To maintain an instruction window large enough to keep a number of functional units busy, instructions from multiple basic blocks must be included.

10. One way to do this is to speculate on the outcome of branches by executing both paths through a branch and then discarding the untaken branch.

11. Instructions other than branch instructions can be executed speculatively.

12. Speculative execution necessitates some method of recovery from data dependence conflicts, usually by returning to an earlier state and re-executing.

13. While speculation increases the instruction window size it places additional requirements on the CPU.

14. Speculation is considered in the context of speculation on branch instructions.

15. Other possibilities to handle branches are:

- **a.** Loop buffers;
- **b.** Multiple instruction streams;
- **c.** Prefetch branch target;
- **d.** Data fetch target;
- **e.** Delayed branch;
- **f.** Look ahead resolution;
- **g.** Branch folding target buffers.

16. A different style of speculation is possible by noticing that some blocks within a program are guaranteed to execute no matter what combination of branches precede them.

Literature

Kaeli David R., Yew Pen-Chung. Speculative Execution in High Performance Computer Architectures. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis Group. LLC. CRC Press, 2005.

VLIW ARCHITECTURE

VLIW – Very **L**ong **I**nstruction **W**ord

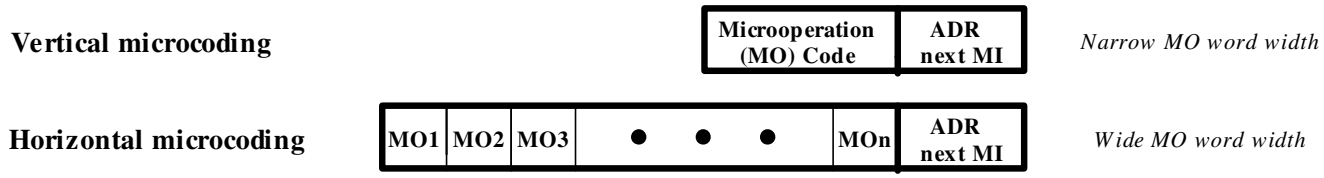
There are four major classes of ILP architectures, which can be differentiated by whether these tasks are performed by the hardware or by the compiler:

1. Superscalar processor architecture,
2. EPIC architecture,
3. VLIW architecture,
4. Dynamic VLIW architecture.

VLIW is an architectural approach that relies on compiler technology, it exploits well-scheduled code. In this architecture are generalized two concepts:

Horizontal microcoding and Superscalar processing

Microinstruction (MI) format



- In VLIW technology several instructions are issued during each clock cycle as in superscalar case.
- Programs written in conventional short instruction words must be compacted together to form the VLIW instructions.
- The compiler reveals parallelism in the program and notifies the hardware on which operations do not depend on each other.
- Different fields of the long instruction word carry the opcodes to be dispatched to differential functional units.

Increasing ILP by Compiler

1. Loop unrolling
2. Software pipelining
3. Trace scheduling

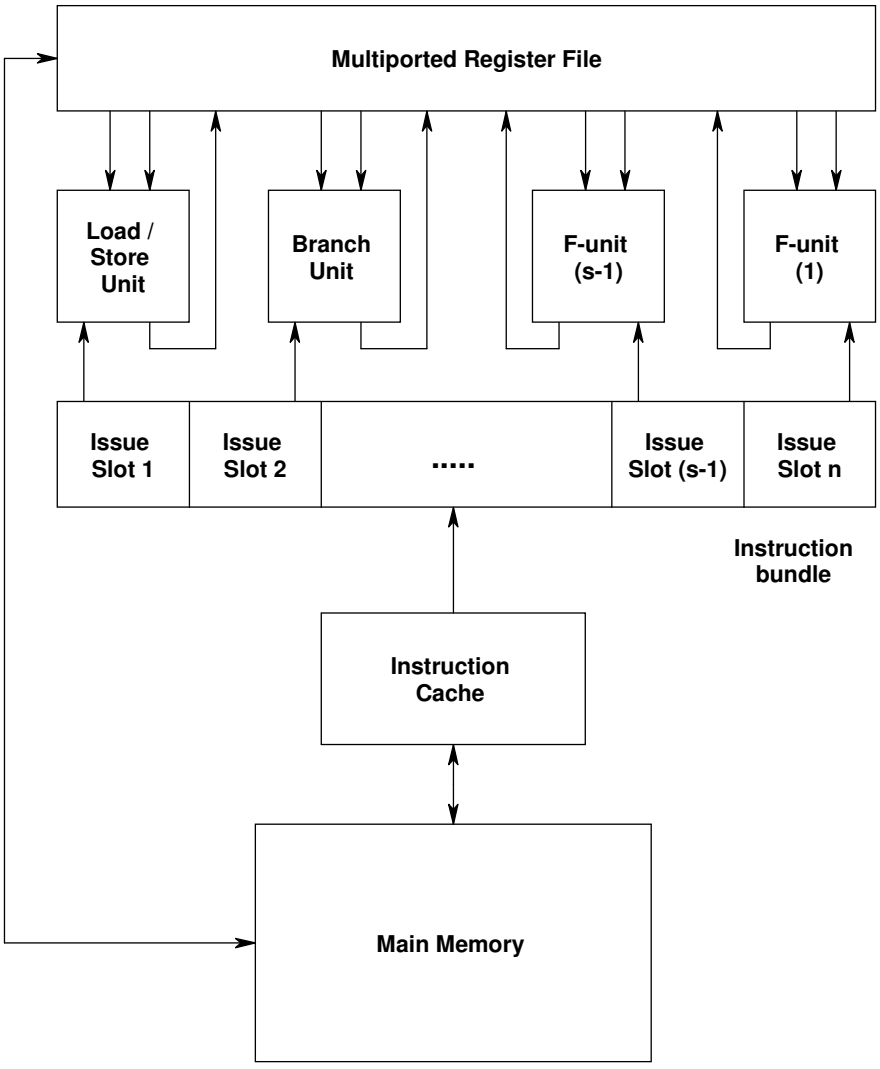
Trace scheduling consists of two phases:

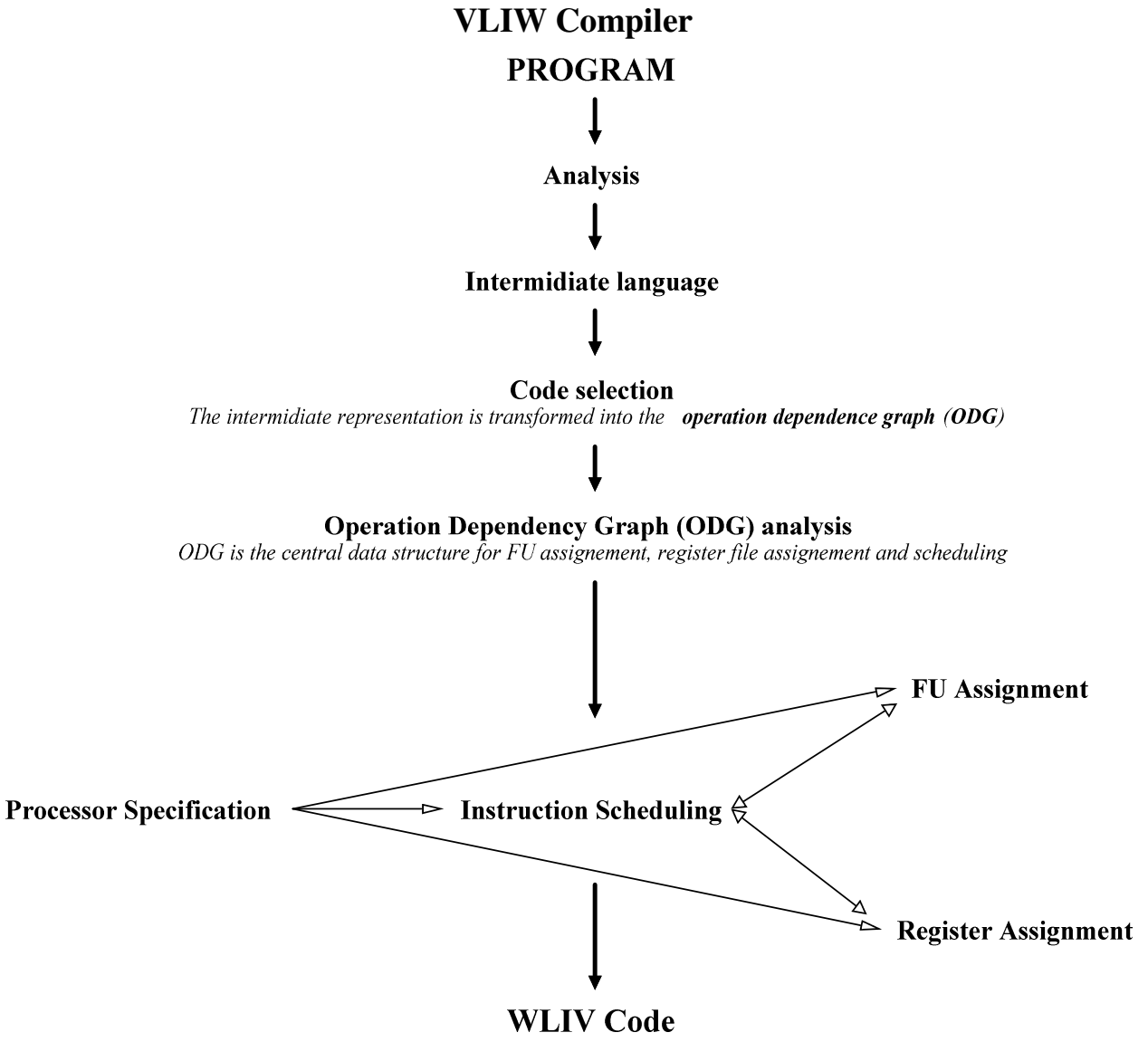
1. **Trace selection** (identifying frequent codes; may use loop unrolling to generate long trace);
2. **Trace compaction** (squeeze trace into a small of wide instructions; move instructions as early as possible; add compensation code).

VLIW *versus* Superscalar (I)

In VLIW processor the compiler guarantees that there are no dependencies between instructions that issue at the same time and that there are sufficient HW resources to execute them.
In superscalar processor the HW buffers and dynamically schedules instructions during operation. Pipelining utilizes temporal parallelism, whereas VLIW and superscalar techniques utilize also spatial parallelism.

VLIW Processor's Model





Techniques to find more ILP out of the sequential instruction stream during preparing information for VLIW processor are:

1. **Loop Unrolling**

Loop unrolling is one basic technique to find more ILP out of the sequential instruction stream.

2. **Software Pipelining**

Software pipelining is a technique which enables the later iterations of a loop to start before the previous iterations have finished the loop.

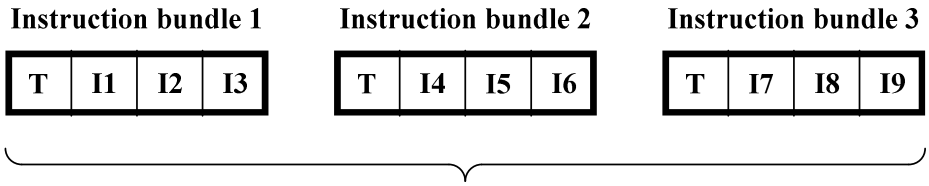
3. Trace Scheduling

The trace scheduling technique has been developed to find more ILP across the basic block boundary. Trace is considered a large extended basic block which has the control path which is frequently taken. This technique considers the branch instructions as jump instructions inside a trace, and it schedules the instructions inside the trace as a large basic block.

All the three techniques can be used to increase the amount of parallelism by extending the instruction pool. They are all strongly depending on an appropriate and accurate control flow prediction.

VLIW Architecture Features

1. Compiler packs independent instructions into VLIW.
2. The multiple operations are tied into one very long instruction, into instruction bundle //*käsukimp*//.
3. Compiler schedules all HW resources.
   The compiler and the architecture for VLIW processor must be co-designed.
4. Instruction latencies are fixed.
   Instruction latency is the inherent execution time for an instruction.
5. Entire long instruction word is issued as a unit.
6. Multiple independent functional units.
7. To keep functional units busy, there must be enough parallelism in a straight-line code.



A super instruction bundle of nine instructions that can be executed in parallel in different ALUs

The template field T //*malliväli*// indicates which of the instructions in the bundle or in the neighbouring bundles can be used in parallel on different functional units.

The possible instruction handling in one instruction bundle:

1. **I1 ‖ I2 ‖ I3** - instructions I1, I2 and I3 are all executed concurrently;
2. **I1 & I2 ‖ I3** – first the instruction I1 is executed, and then the I2 and I3 instructions are executed concurrently.
3. **I1 ‖ I2 & I3** – first the I1 and I2 instructions are executed concurrently, then the I3 instruction;
4. **I1 & I2 & I3** – instructions I1, I2 and I3 are all executed sequentially.

Predicated Instructions

Predication - is a form of compiler-controlled speculation in which branches are removed in favour of fetching multiple paths of control.

Predication is the conditional execution of instructions based on the value of a Boolean source operand, referred to as the predicate. A qualifying predicate is in a 1-bit predicate register(s).

The values in the predicate register file are associated with each instruction in the extended instruction set through the use of an additional source operand. These operand specifiers which predicate register will determine whether the operation should modify processor state.
The values of the predicates registers are set by the results of instructions such as compare and test bit.

1. If the value in the predicate source register is *true* ("**1**"), a predicated instruction is allowed to execute normally.
2. Otherwise ("**0**"), the predicated instruction is nullified, preventing it from modifying the processor state.

Example

Original instruction flow:

if (x > y) // branch Bi //
a = b+c
else
d = e+f

g = h/j-k; // instruction independent of branch Bi //

Predicated instruction flow:

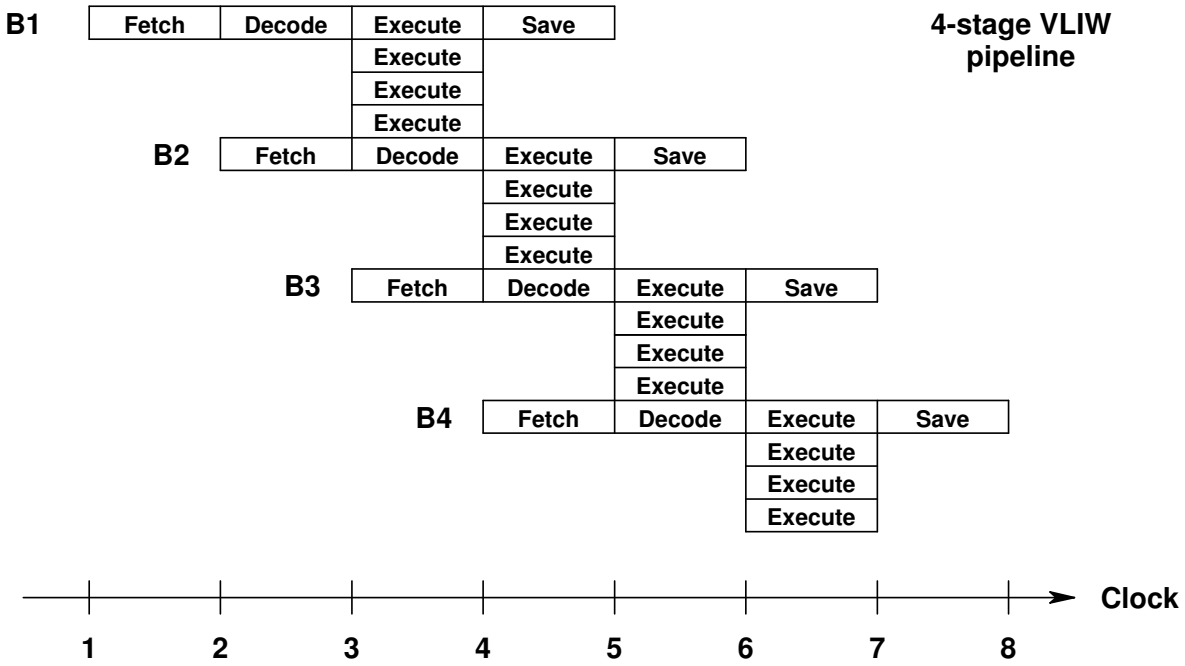
pT, pF = compare (x > y) // branch Bi: pT is set to **TRUE** if x > y, else pF is set to **TRUE** //
if (pT) a = b+c
if (pF) d = e+f
g = h/j-k;

The branch is eliminated and the **compiler** can schedule the instructions under **pT** and **pF** to **execute in parallel**.

Predication Features

It is able to eliminate a branch and therefore the associated branch prediction.
The run length of a code block is increased (better compiler scheduling);
Predication affects the instruction set, adds a port to the register file, and complicates instruction execution;
Predicated instructions that are discarded will consume processor resources, especially the fetch bandwidth;
Predication is most effective when control dependencies can be completely eliminated;
The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence.
Multi-way branch

VLIW Pipeline



4-stage VLIW pipeline with four functional units

- VLIW architecture uses instruction bundles. As for IA-64 defines a 128-bit bundle that contains three instructions, called syllables //*silp*//, and a template field.
- The processor can fetch instructions one or more bundles at a time. The interpretation of the template field is not confined to a single bundle. The template field bits, placed there by compiler, tell to the CPU which instructions can be executed in parallel. Template bits in each bundle specify dependencies both within bundles as well as between sequential bundles.
- When CPU finds a predicated branch, it begins to execute the code for every possible branch outcome. When the outcome of the branch is known, the processor discards the results along "wrong path" and commits results along the "right path". In effect there is no branch at the processor level.
- If the compiler cannot predict a branch then the CPU will behave like a conventional processor, it will try to predict which way the branch will turn.
- A compiler scans the source code to find upcoming loads from main memory. It will add a speculative load instruction and a speculative check instruction.
  At run time, the first instruction loads the data from memory before the program needs it. The second instruction verifies the load before letting the program use the data.

VLIW *versus* Superscalar (II)

1. The code density of the superscalar processor is better than in the VLIW processor.
2. The decoding of the VLIW instructions is easier than the superscalar instructions.
3. The VLIW-processors are exploiting different amounts of parallelism; they require the different instruction sets.
4. VLIW architecture supports only in-order issue of instructions.
5. VLIW is an architectural technique, whereas superscalar is a microarchitectural technique.

Dynamic VLIW Architecture

1. In a dynamic VLIW processor the grouping of instructions and functional units assignment are done by the compiler, but the initiation timing of the operations is done by hardware scheduling.
2. Dynamic VLIW has some advantage over traditional VLIW since it can respond to events at run time that cannot be handled by the compiler at compile times.
3. The dynamic VLIW processor handles run-time events by adding dynamic scheduling hardware for the individual operations.
   Instruction execution is split into two (or three) phases, with the first phase statically scheduled to read the registers, compute a result, and write the result to a temporary results buffer.
   The second phase will move results from the buffer into the register.

VLIW Architecture Limitations

1. Need for a powerful compiler.
2. Limited amount of parallelism available in instruction sequences.
3. Code size explosion.
4. Binary incompatibility across implementations with a varying number of FUs.
5. VLIW processors maximize throughput, not latency, they may not be attractive to the engineers designing embedded microcontrollers and computers.

EPIC Technology Principles

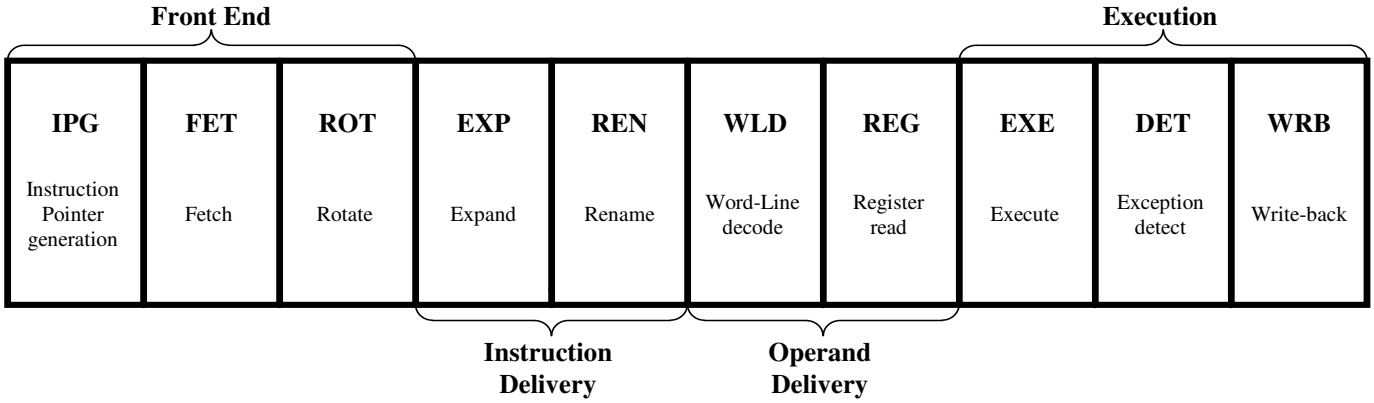
EPIC – Explicitly Parallel Instruction Computing

An instruction group is a sequence of consecutive instructions with no register data dependences among them. All instructions in a group could be executed in parallel, if sufficient HW resources existed.
The boundary is indicated by placing a stop between two instructions that belong to different groups. Unlike earlier parallel VLIW, EPIC does not use a fixed word length encoding.

1. In an EPIC architecture the compiler determines the grouping of independent instructions and communicates this via explicit information in the instruction set.

2. A template in the instruction bundle identifies the instruction type and any architectural stops between groups of instructions.
   The length of a bundle does not define the limits of parallelism.
3. To avoid conditional branches, each instruction can be conditioned (predicated) on a true/false value in a predicate register.
4. IF-THEN-ELSE sequences can be compiled without branches.
5. Multiple targets can be specified and instructions can be prefetched from those paths.
6. The change of the program counter can be delayed until an explicit branch instruction.
7. EPIC incorporates:
   1. Speculation (data and control);
   2. Prediction;
   3. Explicit parallelism;
   4. Scalability with respect to the number of FUs.
8. EPIC`s basic features are:
   1. Separate large register files (64-bit general purpose registers (GR0-GR127), 82-bit floating point registers (FR0-FR127), 1-bit predicate registers (PR0-PR63), 64-bit branch registers (BR0-BR7) – used to specify the target addresses of indirect branches) for execution units.
   2. Parallel instruction execution in separate execution units.
   3. Multi-way branches – by bundling 1-3 branches in a bundle, the execution may jump to any one of the three branches or may fall through to the next instruction. The first true branch will be taken.
   4. 128-bit instruction formats. Instructions are bundled in the groups of three instructions.
   5. Instruction set is optimised to address the needs of cryptography, video encoding and other functions needed by the servers and workstations.

IA-64 Architecture (Itanium) Processor 10-Stage In-Order Pipeline



Front-end (**IPG, Fetch, Rotate**)

1. Prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles;

2. Branch prediction is done using a multilevel adaptive predictor.

Instruction delivery (**EXP, REN**)

1. Distributes up to 6 instructions to the 9 functional units;
2. Implements register renaming for both rotation and register stacking.

Operand delivery (**WLD, REG**)

1. Accesses register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences.

Execution (**EXE, DET, WRB**)

1. Executes instructions through four ALUs and two load/store units, detects exceptions, posts ***NaTs***, retires instructions and performs write-back
2. Deferred exception handling for speculative instructions is supported by providing the ***NaTs*** (***Not a Thing***), for the GPRs, and ***NaT Val*** (***Not a Thing Value***) for FPRs.

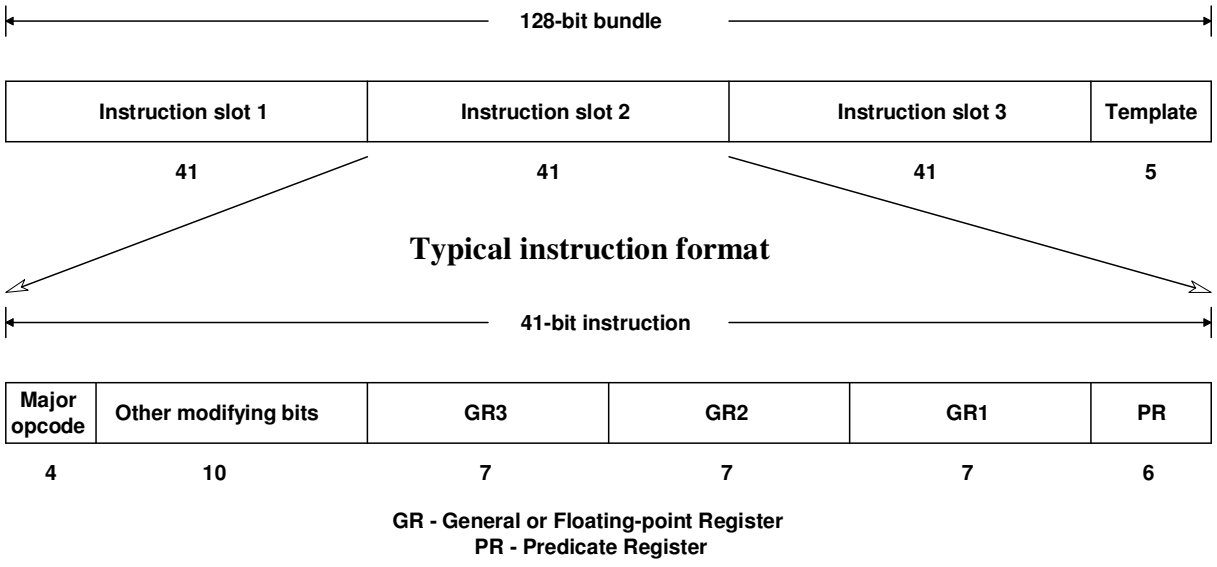
The IA-64 Instruction Bundle

The bundled instructions don't have to be in their original program order; they can represent entirely different paths of a branch.

Compiler can mix dependent and **independent instructions in a bundle**.

Unlike some other VLIW architecture, the IA-64 does not insert NOP instructions to fill slots in the bundles.

Instruction bundle



All RISC-like instructions have a fixed-length 41-bit format.
The template is formed to specify inter-instruction information, which is available in the compiler.

The template value accomplishes two purposes:

1. The field specifies the mapping of instruction slots to execution unit types.
2. The field indicates the presence of any stops.

Stops may appear within and/or at the end of the bundle. The absence of stops indicates that some bundles could be executed in parallel.

Handling Branches in IA-64

1. Normally a compiler turns a source-code branch statement (IF-THEN-ELSE) into alternate blocks of machine code arranged in a sequential stream.
2. Depending on the outcome of the branch the CPU will execute one of those basic blocks by jumping over the others.
3. After tagging the instructions with predicates, the compiler determines which instructions the CPU can execute parallel, even pairing instructions from different branch outcomes as they represent independent paths trough the program.
4. When the CPU finds a predicated branch, it doesn't try to predict which way the branch will work and doesn't jump over blocks of code to speculatively execute a predicted path.
5. At some point, the CPU will eventually evaluate the compare operation that corresponds to the IF-THEN-ELSE statement.

The IA-64 Register Model

The main components of the IA-64 register file are:

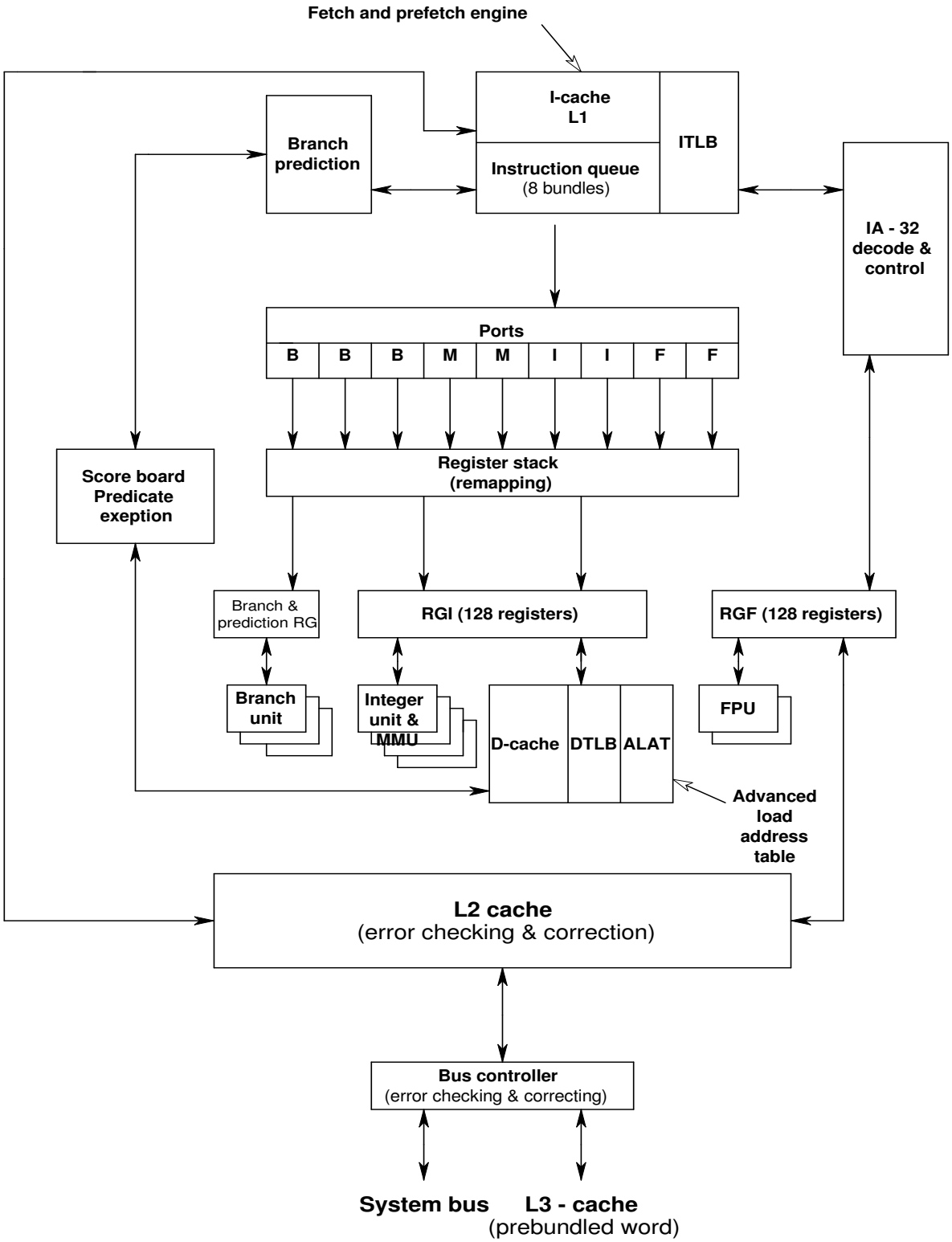
1. 128 general-purpose 64-bit registers, which are actually 65-bits wide;
2. 128 floating-point 82-bit registers, which provide two extra exponent bits over the standard 80-bit IEEE format;
3. 64 predicate 1-bit registers;
4. 8 branch 64-bit registers, which are used for indirect branches;
5. A variety of registers used for system control, memory mapping, performance counters and communication with the OS.

4. Each basic template has two versions – one with a stop after the third slot and one without.
5. Instructions must be placed in slots corresponding to their types based on the template specification, except for A-type instructions that can go in either I or M slots.

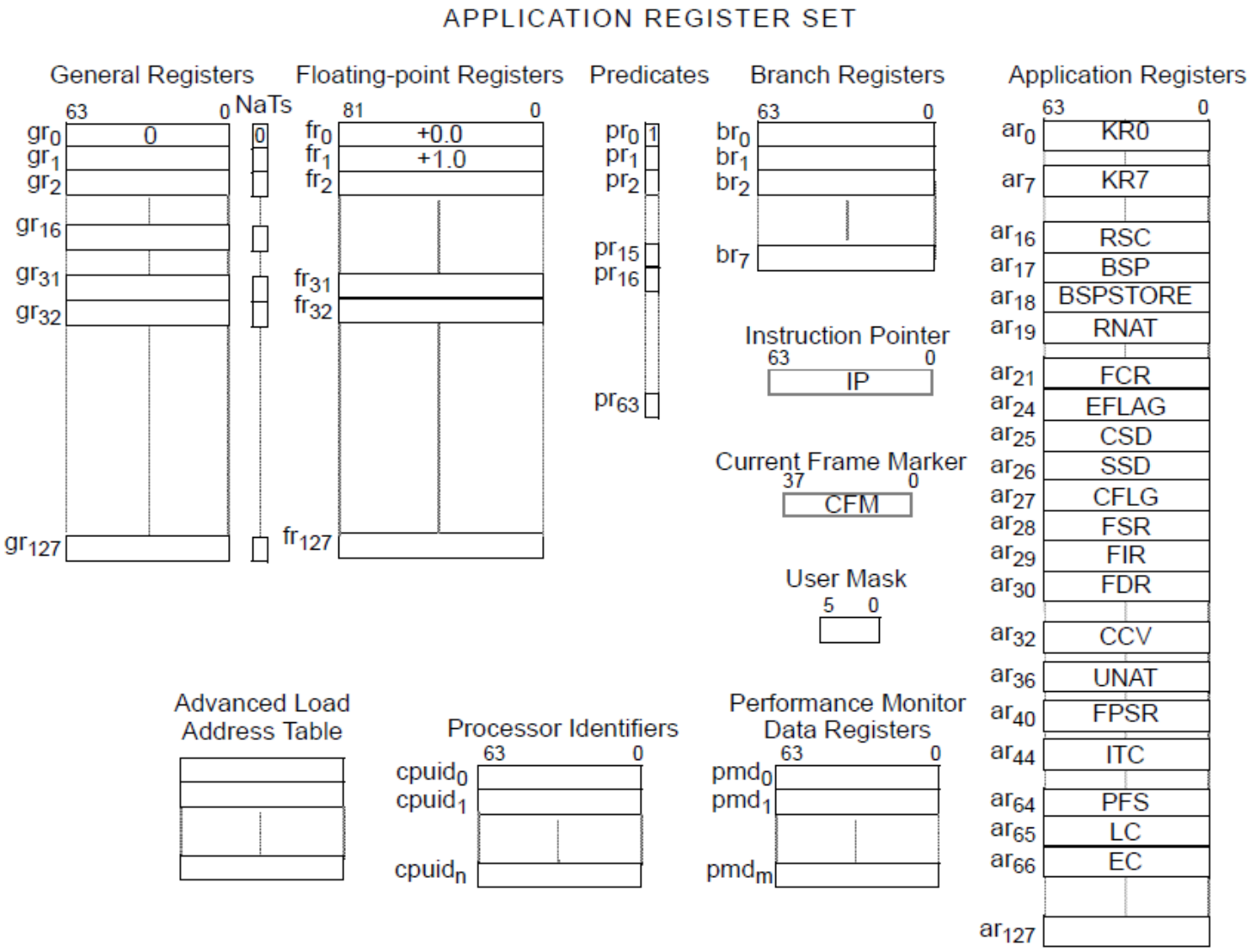
The *Execution Unit Slots* and *Instruction Types* they may hold

Execution Unit slot	Instruction type	Instruction description	Example instructions
I-unit	**A**	Integer ALU	add, sub, and, or, compare, subtract
	I	Non-ALU integer	integer and multimedia shifts, bit tests
M-unit	**A**	Integer ALU	add, subtract, and, or
	M	Memory access	loads and stores for Integer/FP registers
F-unit	**F**	Floating point	floating-point instructions
B-unit	**B**	Branches	conditional branches, calls, loop branches
L+X	**L+X**	Extended	extended immediate, stops and no-ops

Itanium


Fetch and prefetch engine
I-cache
L1
ITLB
Branch
prediction
Instruction queue
(8 bundles)
IA - 32
decode &
control
Ports
B
B
B
M
M
I
I
F
F
Score board
Predicate
exeption
Register stack
(remapping)
Branch &
prediction RG
RGI (128 registers)
RGF (128 registers)
Branch
unit
Integer
unit &
MMU
D-cache
DTLB
ALAT
FPU
Advanced
load
address
table
L2 cache
(error checking & correction)
Bus controller
(error checking & correcting)
System bus
L3 - cache
(prebundled word)

Itanium Register Model



Comment

Superscalar Architecture *versus* IA-64 Architecture

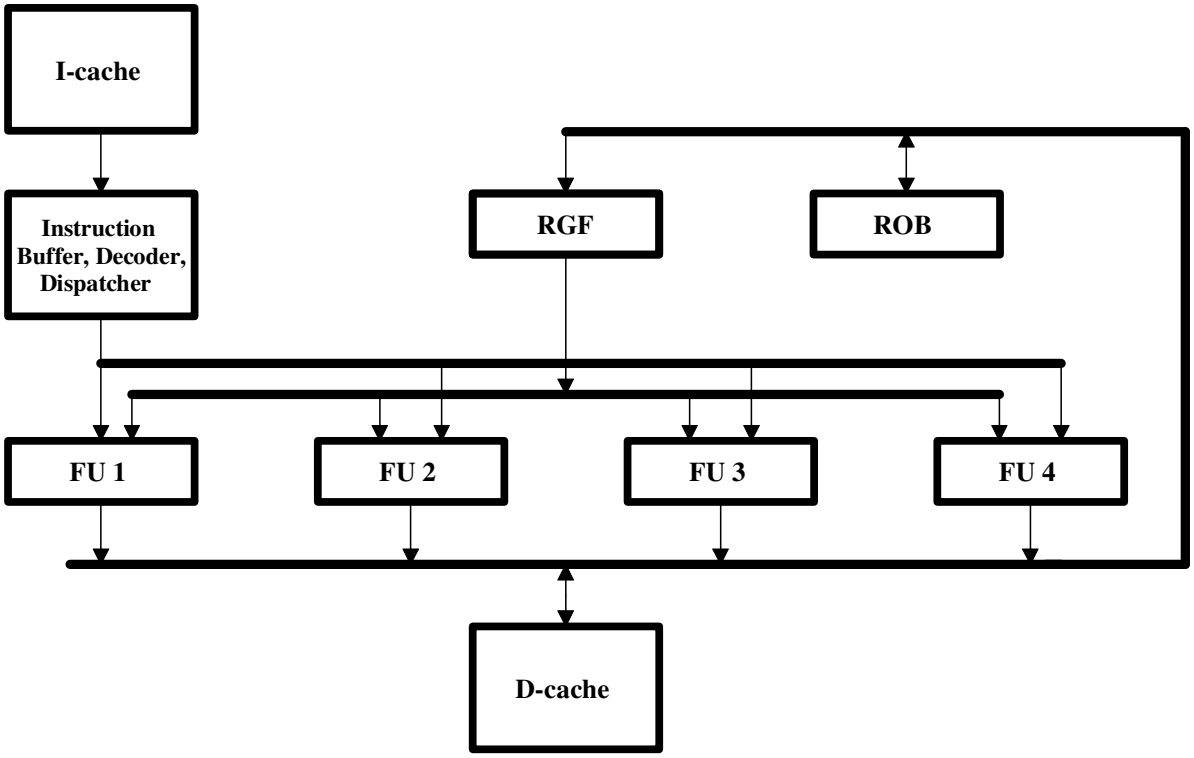
Superscalar	IA-64
RISC-like instructions.	RISC-like instructions bundled into groups of threes.
Instruction stream reordering and optimization at run time.	Instruction stream is reordered and optimized at compile time.
Branch prediction, speculative execution of one path.	Speculative execution along both paths of a branch.
Multiple parallel running execution units.	Multiple parallel running execution units.
Data is loaded from memory only when needed. Tries to find the data in the caches first.	Data is speculatively loaded before it is needed. Tries to find the data in the caches first.

Summary

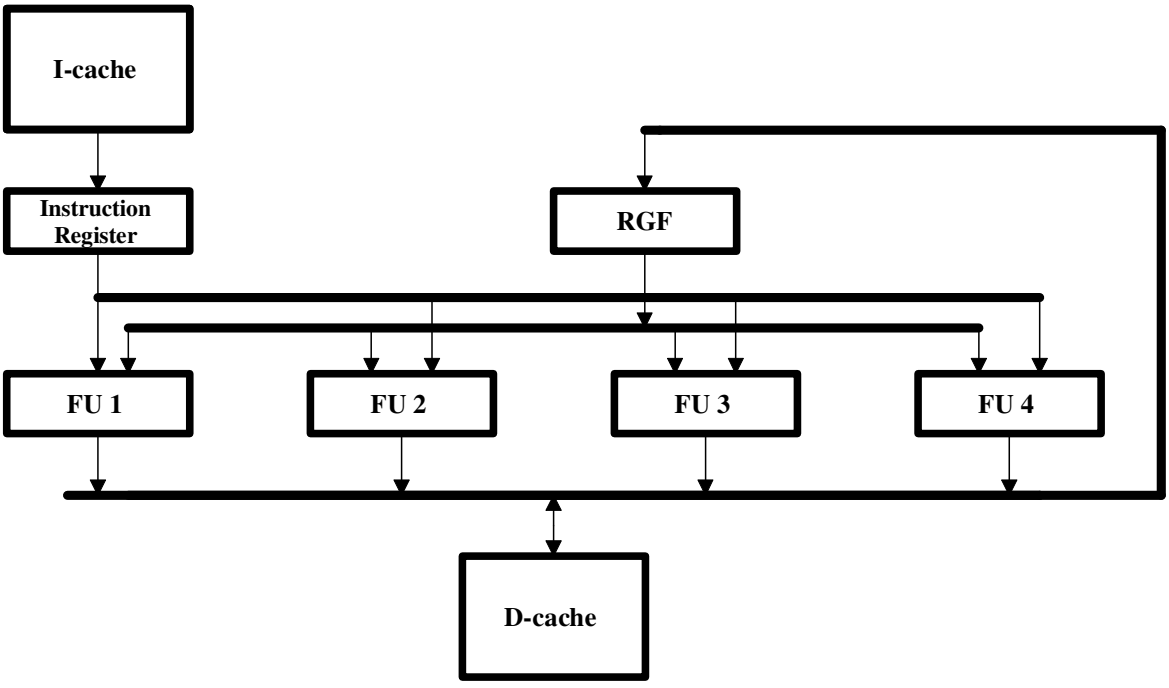
Major Categories of ILP Architectures

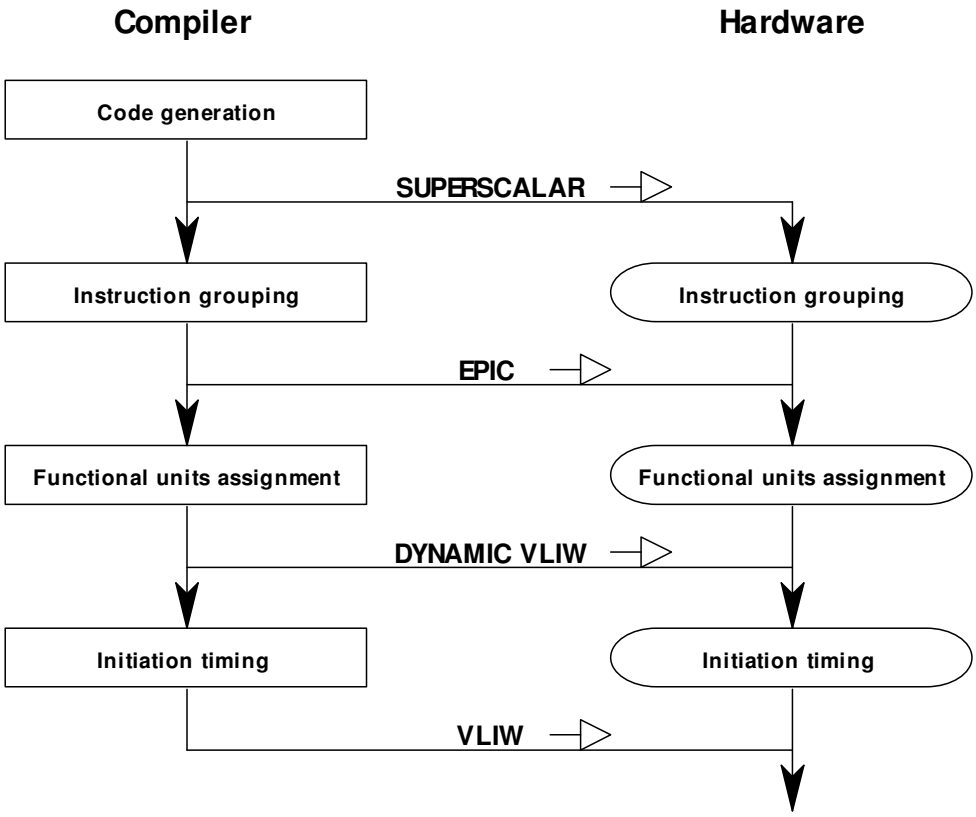
Architecture	Grouping	Functional unit assignment	Initiation
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Superscalar Architecture

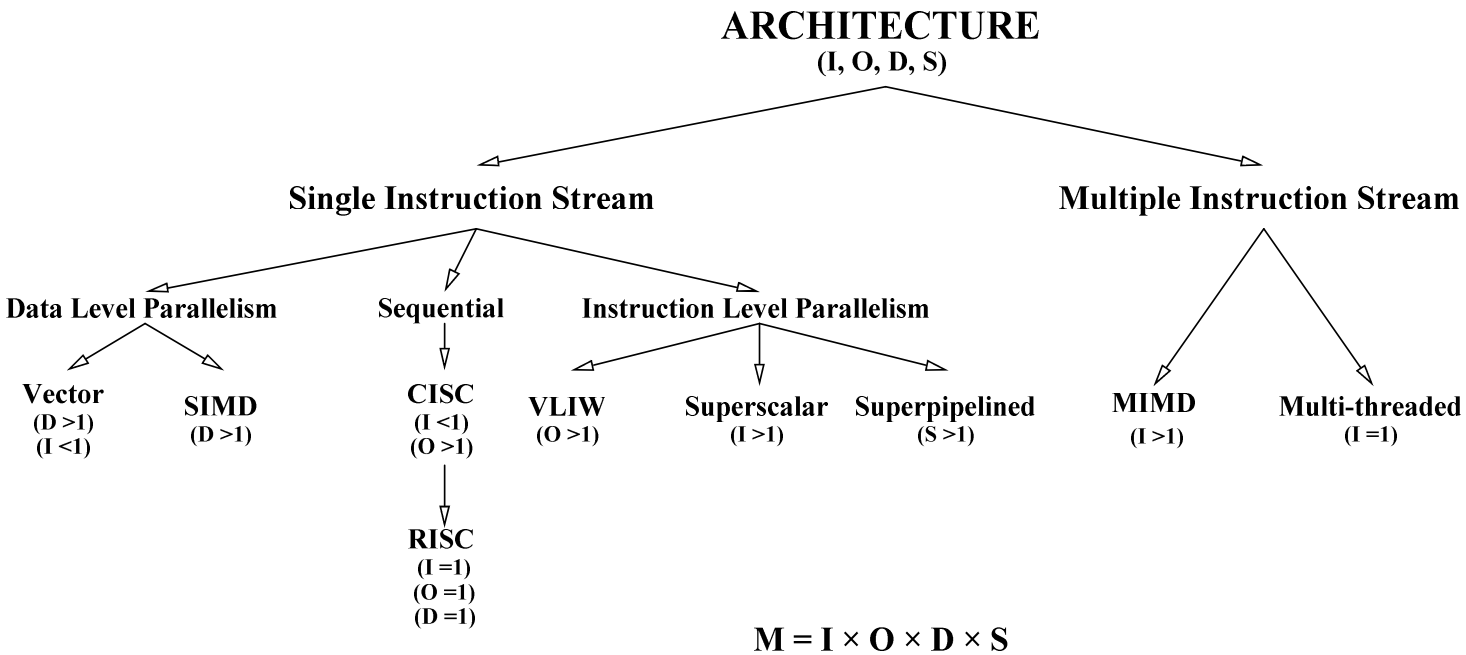


VLIW Architecture





Additional Readings
LM: *EPIC (IA-64)-arhitektuurist*

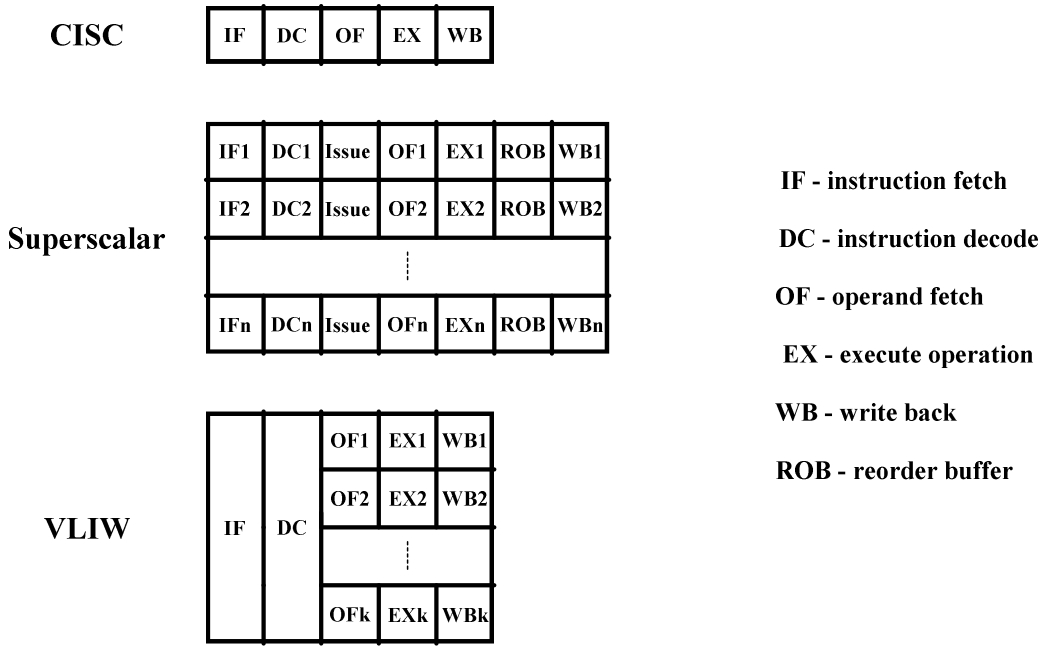


$$M = I \times O \times D \times S$$

I - issue rate (instructions per cycle)
O - the number of operations per instruction
D - the number of operands to which the operation is applied
S - the superpipelining degree (microcycles per instruction)*
M - the amount of parallelism

* The **superpieplining degree (S)** indicates how many delay slots have to be filled (on average) in order to keep the processor (pipeline) busy.

Different Architectures Instruction Pipelines



PROCESSOR ARCHITECTURE DEVELOPMENT TRENDS

The main directions in future processor architecture principles

A. Increase of a single-thread performance – use of more speculative ILP;

Speed-up single–threaded applications:

1. **Trace cache** – tries to fetch from dynamic instruction sequences instead of the static instruction cache;
2. **Superspeculative processor** – enhance wide-issue superscalar performance by speculating aggressively at every point.
3. **Advanced superscalar processor** – scale current superscalar processor designs up to issue 16 or 32 instructions per cycle.

B. Increase of multi-thread (multi-task) performance – utilize thread-level parallelism (TLP) additionally to ILP.

Speed-up multi-threaded applications:

1. Chip multiprocessor (CMP) **//***kiipmultiprotsessor***//**;
2. Simultaneous multithreading (SMT).

C. Speed-up of a single-treaded application by multithreading:

1. **Trace processor //***jäljeprotsessor***//**;
2. **DataScalar processor //***andmeskalaarne protsessor***//**;
3. **Multiscalar processor //***multiskalaarne protsessor***//**.

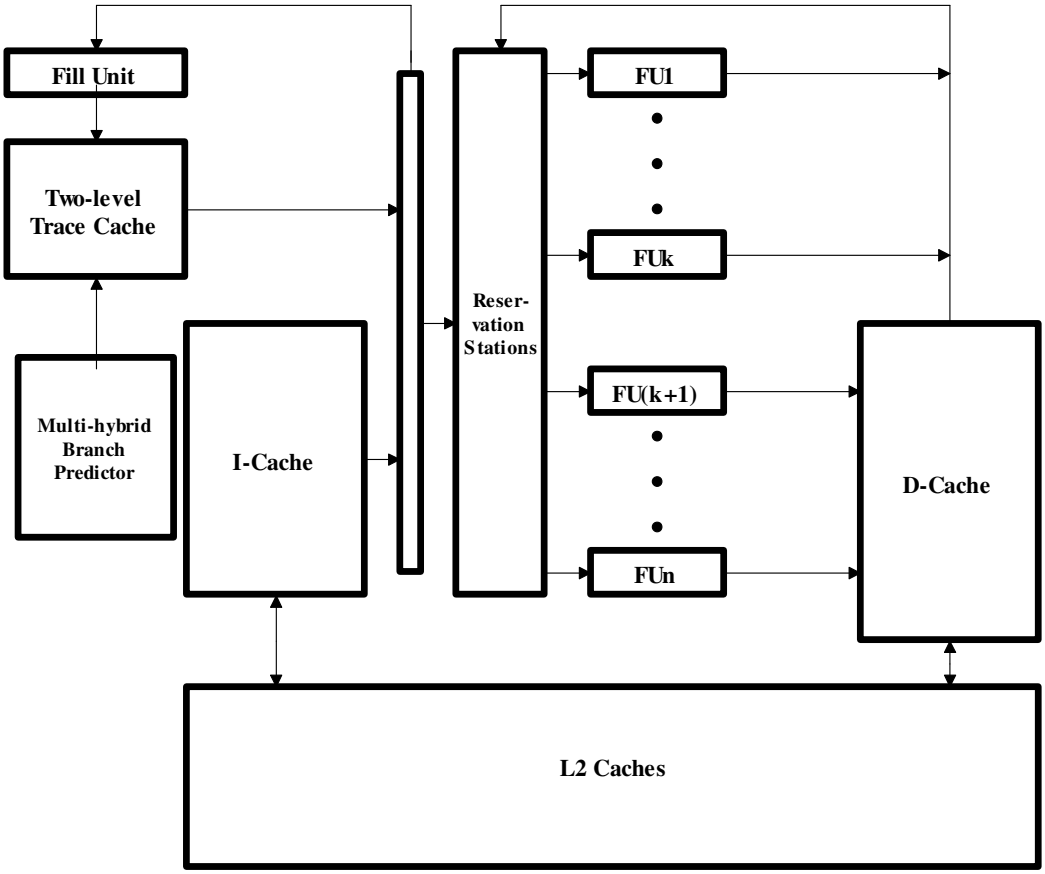
D. The other possibilities are:

1. **RAW** (**>100** reconfigurable processing elements);
2. **Asynchronous processor**.

Advanced Superscalar Processor

The main idea is – more and wider instruction issue.

- To improve instruction supply:
  1. Out-of-order fetch,
  2. Multi hybrid branch predictor,
  3. Trace cache.
- To improve data supply:
  1. Replicate first level cache,
  2. Huge on chip cache,
  3. Data speculation.



Superspeculative Processor

The basis for the superspeculative approach is that producer instructions generate many highly predictable values in real programs. Consumer instructions can frequently and successfully speculate on their source operand values and begin their execution with without results from the producer instructions.
The theoretical basis for superspeculative microarchitecture rests on the weak dependence model.
A superspeculative microarchitecture must maximize:

Instruction flow
It is important the rate at which useful instructions are fetched, decoded, and dispatched to the execution core. Instruction flow is improved by using a trace cache.

Data flow
It is important the rates at which results are produced and register values become available. Eliminates and bypasses as many dependencies as possible. There is used the register renaming mechanism.

Memory data flow
It is important the rate at which data values are stored or retrieved from data memory.

Simultaneous Multi Thread (SMT) Processor (SMP)

- Traditional multi-threaded processors support thread execution via time-sharing method. They schedule threads either at the block granularity or at instruction granularity.
- It is an augment wide superscalar to execute instructions from multiple threads on control concurrently, dynamically selecting and executing instructions from many active threads simultaneously.
- SMT processor (SMP) can safely simultaneously issue several instructions. This means that the instructions that have no intrathread dependence (ILP) and the collection of available instructions over all the threads can be combined and issued to the functional units.
  This will result in a better utilization of hardware resources since the two forms of parallelism are exploited instead of one (ILP), as in a conventional superscalar processor.
- To run multi-thread, it requires saving processor states.
- There are duplicated units for PC and registers.

One chip Multiple Processor CMP

It consists of multiple simple fast processors. Each processor couples to a small fast L1-level cache.

Main features:

- CMP supports the TLP explotation better than other architectures;
- Simple design and faster validation processor units,
- Shorter cycle time,
- Distributed cache lower demand on memory bandwidth,
- Code explicitly parallel.

Disadvantage - It is slower than SMP when running code that cannot be multithreaded, (only one processor can be targeted to the task).

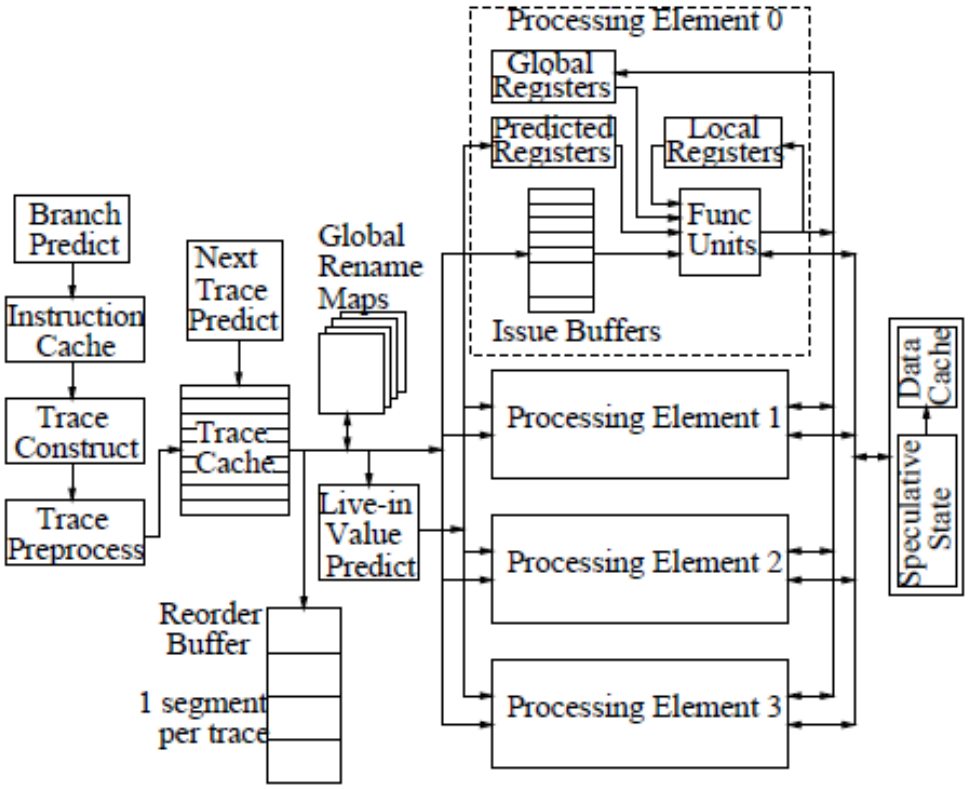
RAW Processor

A highly parallel RAW processor is constructed of multiple identical tiles //*klots*//.
Each tile contains 16kB instruction memory (IMEM), 32kB data memory (DMEM), an ALU, registers, configurable logic (CL) and a programmable switch associated with its 16 kB instruction memory.

- It does not use ISA.
- A program is compiled to hardware.
- The compiler schedules communication.
- Compiler has limits.

Trace Processor

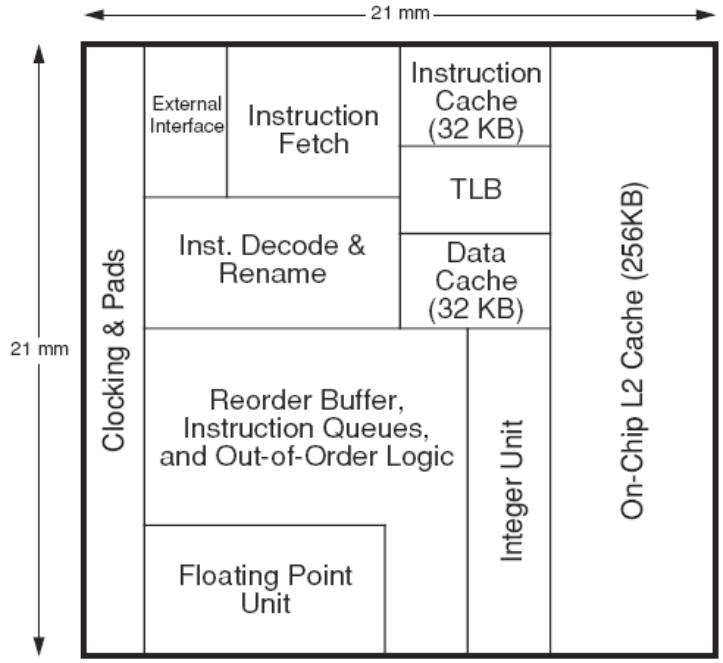
The trace processor consists of multiple superscalar processing cores, each one executing a trace issued by a shared instruction issue unit. It also employs trace and data prediction and shared caches.



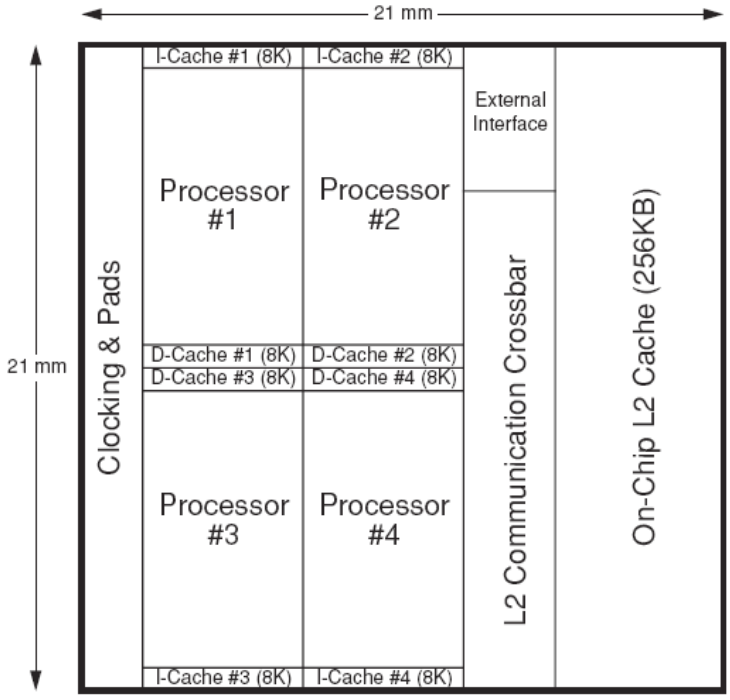
Instruction fetch HW fetches instructions from the I-cache and simultaneously generates traces of the 8 to 32 instructions, including predicted conditional branches.
Traces are stored in a trace cache. A trace fetch unit reads traces from the trace cache and sends them to the parallel processing elements (PE).

There are three different trends in development of future computer architectures:

1. **Multicores** - integrating a set of cores, each of them preserves the same "single thread" performance as previous generation;
2. **Many-cores** - integrating large number of cores, trading single threaded performance with MTL (Multi-Threaded Level) performance;
3. **Special cores** - an integration of multi-cores with many cores, fixed function logic and/or programmable logic such as FPGAs.



Floorplan for the 6-issue dynamic superscalar microprocessor



Floorplan for the 4-way chip multiprocessor (CMP) or 4-core microprocessor

Summary

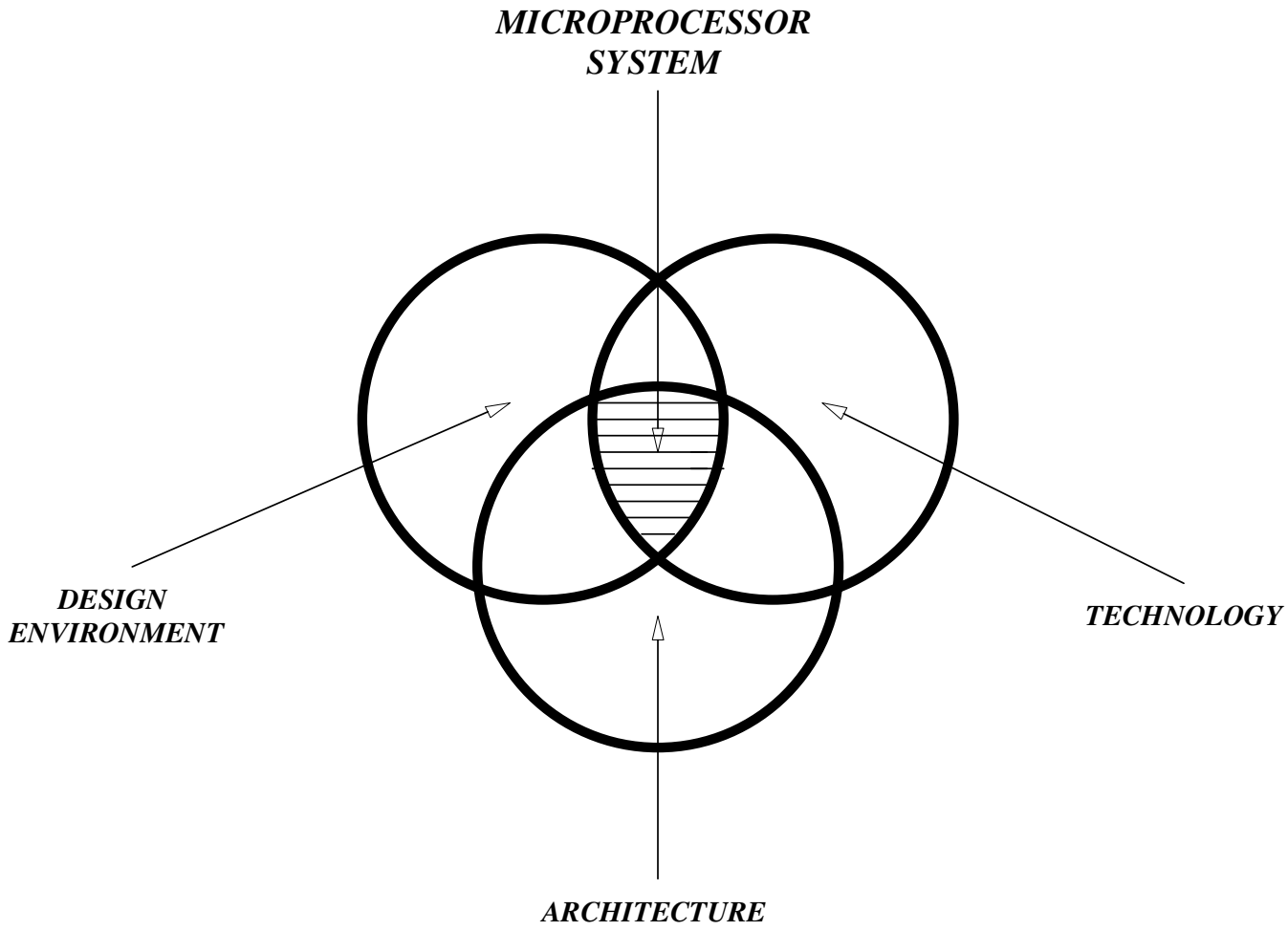
Architecture	Key Idea
Advanced Superscalar	Wide-issue superscalar processor with speculative execution and multilevel on-chip caches
Superspeculative Architecture	Wide-issue superscalar processor with aggressive data and control speculation and multilevel on-chip caches
Trace Processor	Multiple distinct cores, that speculatively execute program traces, with multilevel on-chip caches
Simultaneous Multithreaded (SMP)	Wide superscalar with support for aggressive sharing among multiple threads and multilevel on-chip caches
Chip Multiprocessor (CMP)	Symmetric multiprocessor system with shared second level cache
RAW	Multiple processing tiles with reconfigurable logic and memory, interconnected through a reconfigurable network

The Key Methods for Processor Performance Increasing

by Behrooz Barhami

Established Architectural Metod	Improvement Factor	New Architectural Trend	Improvement Factor
Pipelining (superpipelining)	**3 – 8**	**Multithreading** (super-, hyper-)	**2 – 5?**
Cache memory (2-3 levels)	**2 – 5**	**Speculation and value prediction**	**2 – 3?**
RISC and related ideas	**2 – 3**	**Hardware acceleration**	**2 -10?**
Multiple instruction issue (superscalar)	**2 – 3**	**Vector and array processing**	**2 – 10?**
ISA extensions (for multimedia)	**1 – 3**	**Parallel/distributed computing**	**2 – 1000?**

DEVELOPMENT TRENDS IN TECHNOLOGY



Technology – the process of applying scientific knowledge to industria process.

Computer Technology Development

	ENIAC [10 decimal digits]	× **n**	**Alpha 21164a** [64 bits]
Introduced	**1946**		**1996**
Complexity	17468 tubes // 70000 resistors, 10000 capacitors, 1500 relays, 6000 manual switches, 5×10^6 soldered joints //	> $\mathbf{10^2}$ ($0{,}53\times10^3$)	$9{,}3\times10^6$ transistors
Footprint	$1{,}67\times10^2$ m^2	> $\mathbf{10^{-6}}$ ($1{,}25\times10^{-6}$)	$2{,}09\times10^{-4}$ m^2
Weight	3×10^4 kg	> $\mathbf{10^{-6}}$ ($3{,}33\times10^{-6}$)	$<1\times10^{-1}$ kg
Clock speed*	10^5 Hz	> $\mathbf{10^3}$ (5×10^3)	5×10^8 Hz
Power **	$1{,}6\times10^5$ W	> $\mathbf{10^{-4}}$ ($23{,}8\times10^{-3}$)	3,8×10 W
Cost	$0{,}5\times10^6$ $	> $\mathbf{10^{-3}}$ (3×10^{-3})	$1{,}5\times10^3$ $

***** 5000 additions per second, 537 multiplications per second, 38 divisions per second.
****** Approximately 50 tubes had to be replaced a day**!**

Motor-car:

Speed	110 km/h	10^3	**$1{,}1\times10^5$** km/h
Weight	1 T	10^{-6}	**1,0** g
Power consumption (fuel)	0,1 l/km	10^{-4}	**10^{-5}** l/km

Performance Improvement in Computation

1. Data and instruction pipelining.
2. Multiple processors.
   - **a.** Coarse-grained parallelisl ⇒ processor clusters;
   - **b.** Medium-grained parallelism ⇒ symmetric multiprocessors;
   - **c.** Fine-grained parallelism ⇒ VLIW processors;

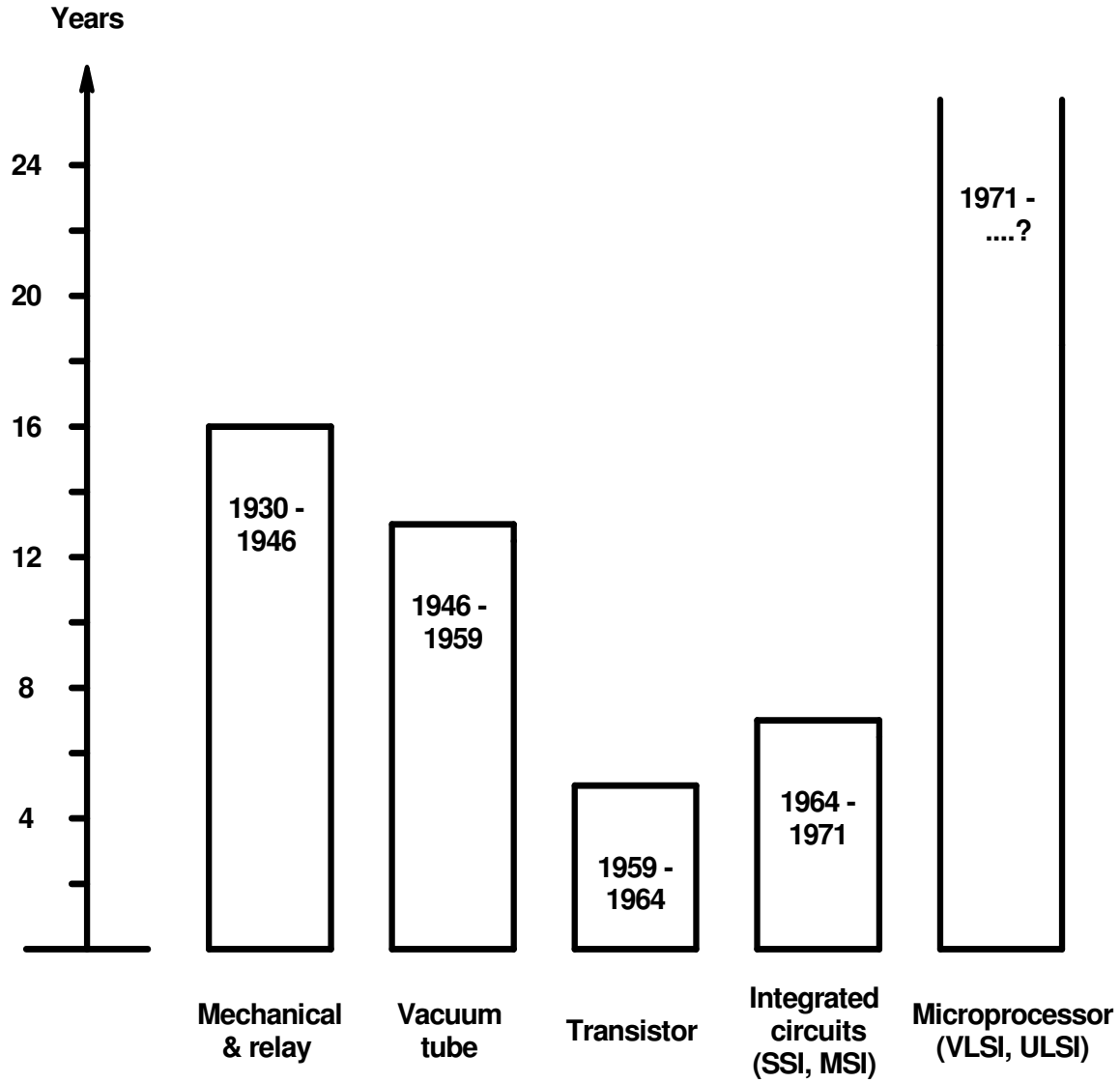
6. Hardware improvements since 1946 (approximately):

- **a.** CPU clock speed $\Rightarrow 10^{4,5}$ times;
- **b.** Bus clock speed $\Rightarrow 10^{3}$ times;
- **c.** RAM latency $\Rightarrow >10^{5}$ times.

Hardware technology improvements enable two evolutionary paths for computers:

Constant price, *increasing performance*;
Constant performance, *decreasing cost*.

Electronic Digital Computer Generations

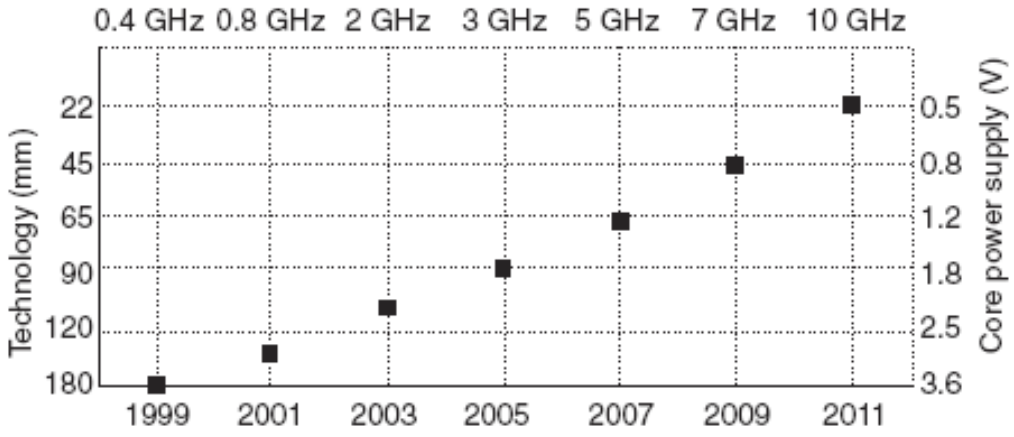


IC Technology Development

Intel Corporation

Year	Resolution (µm)	Wafer size (mm)	Memory DRAM	Processor
1973	8	75	1 kbit	8080 (1974)
1975	5	75	4 kbit	80186
1978	2	100	16 kbit	
1982	1,8	100	64 kbit	80286
1985	1,5	150	1 Mbit	80386
1989	0,8-1,0	150	4 Mbit	80486
1994	0,8	200	16 Mbit	Pentium
1996	0,4	200	64 Mbit	Pentium Pro
1998	0,2	300	64 Mbit	Celeron
2000	0,18	300	256 Mbit	Pentium 4

General Development Dynamics



Gordon Moore's Laws

(Moore's First Law from 1965)

Chip complexity (as defined by the number of active elements on a single semiconductor chip) will double about every device generation (usually taken as about 18 months).

For Moore's Law to remain valid, feature size must continue to be reduced, but since the reduction is insufficient in and of itself, chip size must continue to increase.

Microprocessor Development Trends

	Up to 2004	2005 – 2015
New generation every	2 years	2-3 years
Reduction in gate length (per 2 years)	35%	30%
Reduction in gate oxide thickness (per 2 years)	30%	
Reduction in voltage (per 2 years)	15%	15%
Reduction in interconnect horizontal dimensions (per 2 years)	30%	30%
Reduction in interconnect vertical dimensions (per 2 years)	15%	15%
Add metal layer	1 layer per two generations	1 layer per generation

Technology and Advanced Microprocessors

At the past decade, microprocessors have been improving in overall performance at a rate of approximately 50 – 60% per year.
The performance improvements have been mined from two sources:
1. Increasing clock rates by scaling technology and by reducing the number of levels of logic per cycle;
2. The increasing number of transistors on a chip, plus improvements in compiler technology, to improve throughput.

Scaling in General

Scaling theory is called for reducing transistors dimensions (**scale factor S** (where **S<1**).

> The result would be a transistor whose **gate delay is reduced by S^1**, **area is reduced by S^2**, and **amount of energy** used each time a gate switches *ON* or *OFF* is **reduced by S^3**.

Reduced feature sizes have provided two benefits:

> Since transistors are smaller, more can be placed on a single die, providing area fore more complex microarchitectures.
> Technology scaling reduces transistor's gate length and hence transistor's switcing time.

To support the transistor density improvements, minimum feature sizes have had to be reduced by ~0,7× every three year.

> New-generation *process technologies* generally come out every three years.

Using 0,7x as the scale factor S, gives a transistor area improvement of 0.49×, a gate delay improvement of 0,7×, and an energy reduction of 0,34× for every new process generation.
CMOS technologies allow digital circuits to be designed with signals that have full voltage swing and low standby (leakage) current.

Transistor leakage is the greatest problem facing continued scaling.

The average power dissipation **(P)** of a digital CMOS circuit consists of a static **(P_{stat})** and a dynamic **(P_{dyn})** components:

$$\mathbf{P = P_{stat} + P_{dyn}}$$

The static power (P_{stat}) characterizes circuits that have a constant source current between the power supplies.
The dynamic power Pdyn is the dominant part of the power dissipation in CMOS circuits, and it is composed of three terms:

$$\mathbf{P_{dyn} = P_{switching} + P_{short\text{-}circuit} + P_{leakage}}$$

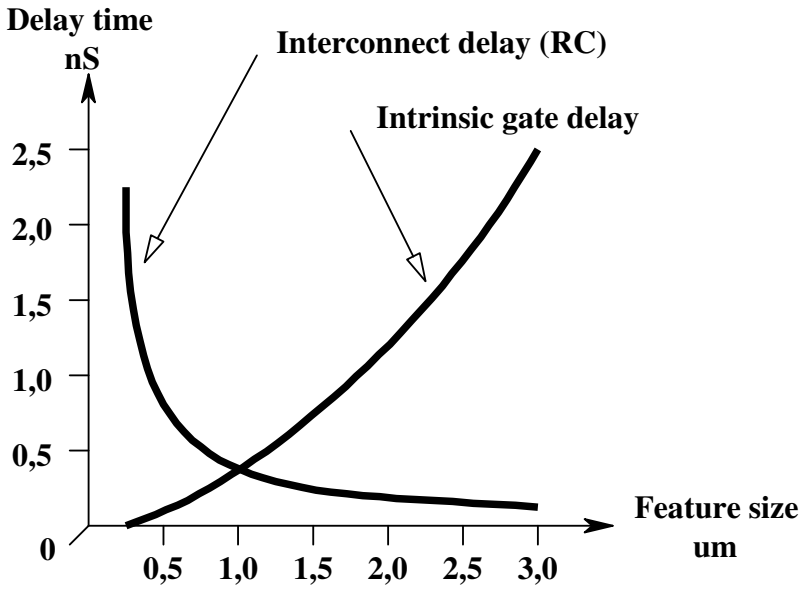
The switching power ($P_{switching}$) is due to the charge and discharge of the capacitances associated with each node in the circuit. The short-circuit power ($P_{short\text{-}circuit}$) derives from the short-circuit current from the supply to the ground voltage during gates output transitions. The leakage power ($P_{leakage}$) is due to the leakage currents in MOSFET transistors.
So the average power consumption of a circuit (transistor) is equal to:

$$\mathbf{P = 0{,}5 \times C \times U^2 \times f + I_{off} \times U,}\ \text{where}$$

f is the clocking frequency; **C** is the gate-oxide capacitance; $\mathbf{I_{off}}$ is the total transistor leakage current and U is the power supply voltage.

Interconnections Scaling

1. Interconnects don't have the same scaling properties as transistors. Interconnects are quickly becoming as critical as transistors in determining overall circuit performance.
2. The average width of interconnections is decreasing to provide the smaller dimensions needed to pack in more transistors and wires.
3. The speed at which a signal can propagate down a wire is proportional to the product of the wire resistance (**R**) and the wire capacitance(**C**). This speed is named as the wire's RC delay. Increased R or C results in increased wire delay and slower signal propagation.



Effect of interconnect scaling

Reducing the feature sizes has caused wire width and height to decrease; resulting in larger wire resistance due smaller wire cross-sectional area unfortunately wire capacitance has not decreased proportionally.

4. By convention the wiring layers are subdivided into three categories:

1. *Local*, for connection within a cell;
2. *Intermediate* or *mid-level*, for connection across a module;
3. *Global* or *top-level metal layers*, for chip-wide communications.

5. To reduce communication delays, wires are both wider and taller in the mid-level and top-level metal layers.

Gate length (nm)	Metal ρ (μΩ/cm)	Mid-level Metal			Top-level Metal		
		Width (nm)	R_{wire} (mΩ/μm)	C_{wire} (fF/μm)	Width (nm)	R_{wire} (mΩ/μm)	C_{wire} (fF/μm)
250	3,3	500	107	0,215	700	34	0,256
180	2,2	320	107	0,253	530	36	0,270
130	2,2	230	188	0,263	380	61	0,285
100	2,2	170	316	0,273	280	103	0,296
70	1,8	120	500	0,278	200	164	0,296
50	1,8	80	1020	0,294	140	321	0,301
35	1,8	60	1760	0,300	90	714	0,317

6. New interconnect materials are needed to improve performance without increasing the number of metal layers.

7. Another way to improve interconnect performance is to reduce the capacitance between wires.

Shrinking linewidths not only enables more components to fit onto IC (typically 2× per linewidth generation) but also lower costs (typically 30% per linewidth generation).
Shrinking linewidths have slowed down the rate of growth in die size to 1,14× per year versus 1,38× to 1,58× per year for transistor counts, and since the mid nineties accelerating linewidth shrinks have halted and even reversed the growth in die size.

Wire Delay Impact on Microarchitecture

1. The widening gap between the relative speeds of gates and wires will have a substantial impact on microarchitectures.
2. With increasing clock rates, the distance that a signal can travel in a single clock cycle decreases.
3. When combined with the modest growth in chip area, the time to send a signal across one dimension of the chip will increase dramatically.
4. Clock scaling is more significant than wire delay for small structures, while both wire delay and clock scaling are significant in larger structures.
5. The large memory-oriented elements will be unable to continue increasing in size while remaining accessible within one clock cycle.
6. For each technology the access time increases as the cache capacity increases.
7. The most significant difference between a cache and a register file is the number of ports.
8. If the number of execution units is increased, the distance between the extreme units will also increase.
9. Large monolithic cores have no long-term future in deep submicron fabrication processes.
10. The on-chip memory system is likely to be a major bottleneck in future processors.
11. Future microprocessors must be partitioned into independent physical regions and the latency for communicating among partitions must be exposed to the microarchitecture and possibly to the ISA.

Energy per Instruction (EPI)

The goal of modern processor is to deliver as much performance as possible while keeping power consumption within reasonable limits.

The power efficiency of a processor can be measured by Energy per Instruction (EPI).

{**P** [W] = **E** [J] **/ t** [s]; **Power** (**P**) is the **rate of using energy** (**E**).}

It is the amount of energy expended by processor for each executed instruction.
EPI is measured in Joules per Instruction (Joules / Instruction).

EPI is related to other commonly used power-efficiency metrics as for performance / Watt or MIPS / Watt.

EPI = Joules / Instruction

EPI = (Joules / second) / (Instruction / second) = Watt / IPS

EPI = Watt / IPS
IPS – Instructions per Second

EPI and MIPS / Watt do not consider the amount of time needed to process an instruction from start to finish. The EPI is a function of energy (E) expended per instruction when the instruction is processed.

$$\mathbf{E \approx C \times U^2},$$ where
C – switching capacitance; **U** – power supply voltage.

EPI is influenced by several factors, as for:

1. Processor microarchitecture $\Rightarrow$ **C**;
2. Process technology $\Rightarrow$ **C**, **U** (bounds);
3. Environment (supply voltage) $\Rightarrow$ **U**.

In each new processor generation has reduced both C and U compared to the prior generation. The combination of limited instruction parallelism suitable for superscalar issue, practical limits to pipelining, and a power limits has limited future speed increases within conventional processor cores. Microprocessor performance increases will slow dramatically in the future.

> The next stage of development for microprocessor systems is basis on the **CMP** (**Chip Multiprocessor**).

Multi-threaded software that can take advance of chip multiprocessors or CMP inherently will have phases of sequential execution.

- During phases of limited parallelism (low IPS) the CMP will spend more EPI.
- During phases of high parallelism the CMP will spend less EPI.
- In both scenarios power is fixed.

When a two-way CMP replaces a uniprocessor, it is possible to achieve essentially the same or better through put on server-oriented workloads with just half of the original clock speed.
It is because request processing time is more often limited by memory or disk performance than by processor performance.
The **lower clock rate** allows designing the system with a significantly **lower power supply voltage**, often a nearly linear reduction.

For a chip-multiprocessor it is necessary to parallelize most latency-critical software into multiple parallel threads of execution to take advantage of a CMP. CMPs make information handling process easier than with conventional multiprocessors.
The goal is to minimize the execution times of multi-threaded programs, while keeping the CMP total power consumption within a fixed budget.

Summary

Energy consumption in general is a sum of two components: active energy and idle energy.
Minimizing the idle energy consumption is relatively simple: the processor enters a deep-sleep power state, stops the clocks, and lowers the operating voltage to the minimum.
Minimizing the active energy consumption is more complex. A very slow execution consumes less power for a longer period of time, while heavy parallelism reduces the active time but increases the active power.

$$\text{Energy\_active} = \text{Power\_active} \times \text{Time\_active}$$
$$\text{Energy\_active} = \text{Power\_active} / \text{Performance\_active}$$

The processor's performance benefit must be greater than the additional power consumed.

Literature

Vasanth Venkatachalam, Michael Franz. Power Reduction Techniques for Microprocessor Systems. ACM Computing Surveys, Vol. 37, No. 3, September 2005, pp.195-237.

Dual and Multicore Processors

Core – the core set of architectural, computational processing elements that provide the functionality of a CPU.

Dual Core Processor

A dual core processor is a CPU with two separate cores on the same die, each with its own cache. Each core handles incoming data strings simultaneously to improve performance.
The dual-core type processors fall into the architectural class of tightly-coupled multiprocessors.

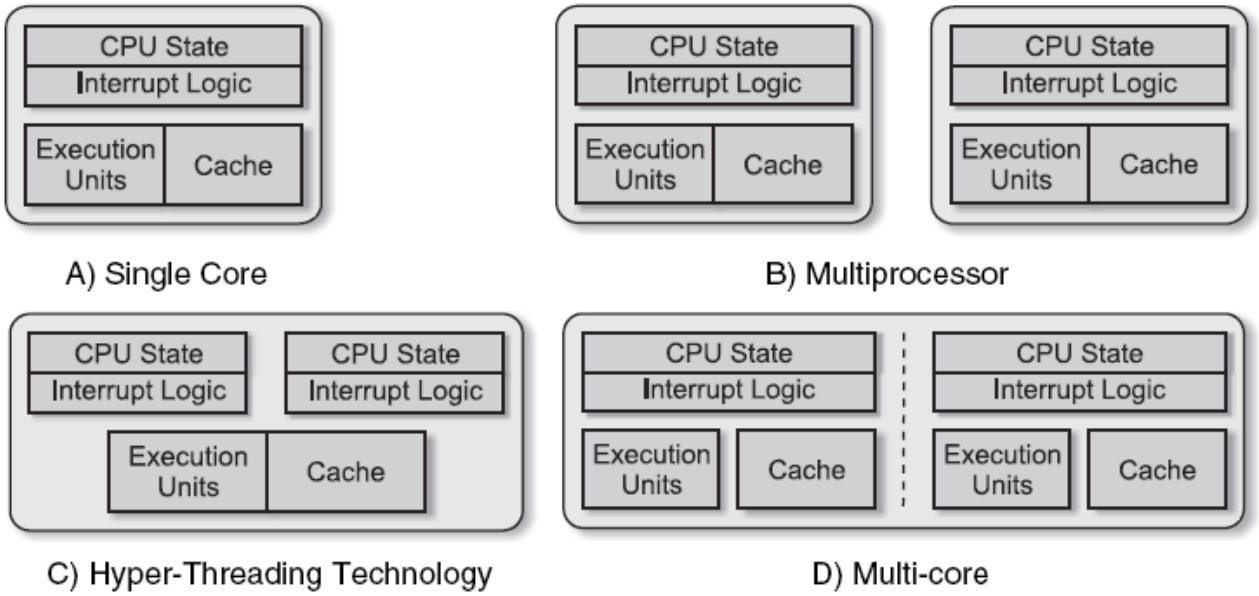
Advantages and Disadvantages of Dual-core Processors

Advantages	Disadvantages
1. Processor uses slightly less power than two coupled single-core processors (because of the increased power required to drive signals external to the chip and because the smaller silicon process geometry allows the cores to operate at lower voltage) **2**. The cache coherency circuitry can operate at much higher clock rate than is possible if the signals have to travel off-chip. **3.** The design requires much less printed circuit board (PCB) space than multi-chip designs.	**1.** Processor requires operating systems support to make optimal use of the second computing resource. **2**. The higher integration of the dual-core chip drives the production yields down and it is more difficult to manage thermally. 3. Scaling efficiency is largely dependent on the application (applications that require processing large amounts of data with low computer-overhead algorithms may have I/O bottleneck). 4. If a dual-core processor has only one memory bus, then the available memory bandwidth per core is half the one available in a dual-processor mono-core system.

Multi Core Processor

A **multi-core architecture** is actually a **symmetric multiprocessor** (**SMP**) implemented on a single VLSI circuit.
The goal is to allow greater utilization of thread-level parallelism (TLP), especially for applications that lack sufficient instruction-level parallelism (ILP) to make good use of superscalar processors.
It is called chip-level multiprocessing (CMP) or chip-level multithreading (CMT).
For instructions execution a simultaneous multithreading (SMT) technique is exploited in these processors.

Dual core SMT allows multiple threads to execute different instructions in the same clock cycle. It means that the processor has the ability to fetch instructions from multiple threads in a cycle and it has a larger register file to hold data from multiple threads.



Single-core, Multiprocessor, Intel's Hyper-Threading and Multi-Core Architectures

Multi-threaded applications running on multi-core platforms have different design considerations than do multi-threaded applications running on single-core platforms.

On **single-core platforms**, assumptions may be made by the developer to simplify writing and debugging a multi-threaded application. These assumptions may not be valid on **multi-core platforms**. There are two areas that highlight these differences:

1. Memory caching,
2. Thread priority.

In the case of memory caching, each processor core may have its own cache. At any point in time, the cache on one processor core may be out of sync with the cache on the other processor core. On multi-core platforms, the scheduler may schedule both threads on separate cores. Both threads may run simultaneously independently.

Multi-core Pocessor Prformance

The Amdahl's law states, that decreasing the serialized portion by increasing the parallelized portion is of greater importance than adding more processor cores.

Only when a program is mostly parallelized does adding more processors help more than parallelizing the remaining code.
To make Amdahl's Law reflect the reality of multi-core systems, rather than the theoretical maximum, system overhead from adding threads should be included:

$$\text{Speedup} = \frac{1}{S + (1 - S)/n + H(n)}$$

Where S is the time spent executing the serial portion of the parallelized version, n is the number of processor cores and H(n) is the overhead.
The overhead consists of two portions:

The operating system overhead[5]
Inter-thread activities.

If the overhead is big enough, it offsets the benefits of the parallelized portion.
The important implication is that the overhead introduced by threading must be kept to a minimum.

Intel's Hyper-Threading (HT) Technology *versus* Multi-core Designs

The HT Technology and multi-core designs differ significantly and deliver different performance characteristics. The key difference lies in how a program's instructions are executed.
The performance of HT Technology is limited by the availability of share resources to the two executing threads. In multi-core chips each core has the resources required to run without blocking resources needed by the other threads.

Example

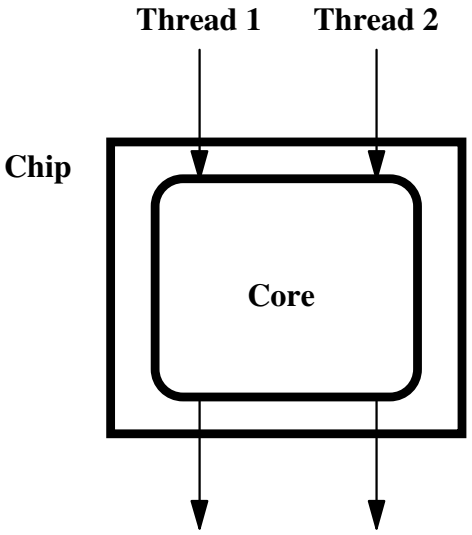
**IBM`s** dual-core on a single die ***Power 6*** processor run at speeds between 4-5 GHz with a total of 8 Mbytes L2 cache and a 75 GB/s peak to main memory. The processor is built in 65-nm process using IBM`s silicon-on-insulator (SOI) technology. IBM applies new technology in variable gate lengths and variable threshold voltages to squeeze maximum performance per Watt. The chip can be fully operated at 0,8V.

In 2006 **Intel** developed a prototype **80-core chip** that can process some 1,8 trillion operations per second. There is a cache memory under each core. Chips power consumption is handled by division of each tile //*protsessortuum*// into separate regions that can be powered individually.

---

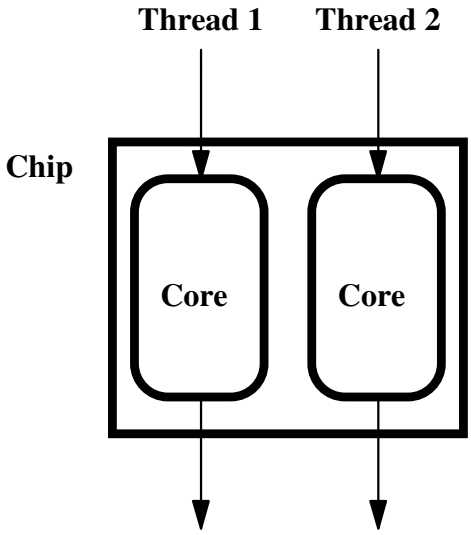
[5] **Overhead** – **a**. An extra code that has to stored to organize the program.
//*ballast*// **b**. The time a processor spends doing computations that do not contribute directly to prograess of any user tasks in the system.

The 80-core chip achieves over a teraflop of performance while power consuming is about 62 Watts.



Processor with HT Technology

One core, two executing threads, one executing pipeline



Multi-core Processor

One chip, multiple (two) cores, multiple (two) execution pipelines

THE NEXT STEP?

Computer Architecture Formulas

1. **Amdahl's Law**

$$\textbf{Speedup} = \text{(Execution old time) / (Execution time new)} = \frac{1}{(1-f)+\frac{f}{s}}$$

2. $\mathbf{CPU_{time}}$ = Instruction count × Clock cycles per instruction × Clock cycle time

$$\mathbf{CPU_{time} = IC \times CPI \times t_{CLK}}$$
$$\mathbf{CPI = IPC^{-1}}$$

3. **Performance** = 1 / Execution time

4. **Average memory access time** (**AMAT**) = Hit time × Miss rate × Miss penalty

5. **Total CPI** = Base CPI + Memory stall cycles per instruction

6. **Misses per instruction** = Miss rate × Memory accesses per instruction

7. **Cache index size**: $\mathbf{2^{index}}$ = Cache size / [(Block size) × (Set associativity)]

8. **Pipeline speedup** = $\frac{Pipeline\_depth}{1+(Branch\_frequency)\times(Branch\_penalty)}$

9. **The total execution time** ($\mathbf{T_{exe}}$) = IC × CPI × t_{clk} + T_{Mem} + $T_{I/O}$

10. **Power static** = $Current_{static}$ ×Voltage

11. **Power dynamic** = 0,5 × Capacitive load × $Voltage^2$ × Frequency switched

12. **Amdahl's Law for multi-core systems**

$$\textbf{Speedup} = \frac{1}{S+\frac{1-S}{n}+H(n)}$$

H(n) – overhead

GLOSSARY

Atomic instruction --- an instruction which can not be interrupted (cannot be split up). Instruction must be performed entirely or not at all (as for "test-and-set" instructions). These instructions usually are not atomic at bus level, only at the CPU instruction level.

Aliasing -- a situation in which the same object is accessed by two addresses.

Architectural registers – the instruction set visible registers of a processor.

Associativity - the number of locations where a single memory address may be stored in a cache memory.

Basic block --- a sequence of instructions without branches (except possibility at the end) and without branch targets or branch labels (except possibility at the beginning).

Benchmark --- program or program fragment and related data inputs that are executed to test the performance of a hardware component, a group of hardware or software components or an entire microprocessor system (MPS).

Branch Target Buffer - a memory cache supporting branch prediction by holding
BTB information on the past behavior of recent branches and their target addresses.

Bus --- communication channel shared by multiple devices within a MPS or network.

Bus cycle --- period time required to perform one data transfer operation on a bus.

Cache --- area of high-speed memory that holds portions of data also held within another storage device.

Card --- a printed-circuit panel (usually multilayer) that provides the interconnection and power distribution to the electronics on the panel, and provides interconnect capability to the next level package. It plugs into a mother board printed-circuit board.

Central Processing Unit (**CPU**) --- computer processor responsible for fetching, interpreting, and executing program instructions.

Chipset - a pair of chips responsible for communication between the processor and other components (between the high-speed components and the other with lower-speed components) on the motheboard.

Clock cycle --- frequency at which a system clock generates timing pulses.

Clock jitter – variation in clock frequency, causing some clock cycles to be longer or shorter
//taktivärin// than others.

Clock rate --- **a**. rate at which clock pulses are generated by a clock.
b. with respect to a CPU , the time required to fetch and execute the simplest instruction.

Control unit --- component of a CPU that is responsible for moving data,
access instructions and controlling the ALU.
In a processor the part that retrieves instructions in proper sequence,
interprets each instruction, and applies the proper signals to the
arithmetic and logic unit and other parts in accordance with this interpretation.

Datapath – the component of the processor that performs arithmetic operations.

Die --- integrated circuit chip as cut (diced) from finished wafer.

Embedded computer -- a computer inside another device used for running one predetermined
application or collection of software.

Engine --- a. a specialized processor (graphics processor).
b. a software that performs a very specific and repetitive function in contrast to an
application that has many functions offered to the user (search engine, database engine).

Etching - physical and chemical means of removing a material layer.

Extensibility --- the ease with which a third party is able to add capabilities to software or
hardware.
//laiendatavus//

Granularity --- refers to the number of instructions that can be performed concurrently before
some form of synchronization needs to take place. It is associated with the
ratio between computation and communication.

Fetch cycle --- portion of CPU cycle in which an instruction is loaded into the instruction
register and decoded.
Instruction --- the specification of an operations and the identification of any associated operands.

Instruction set --- the complete set instructions recognized by a given computer or provided
by a given programming language.

Instruction window – the set of instructions that is examined for simultaneous execution,

Interoperability --- the ability of two or more computer systems to exchange information
//koostalitlusvõime// and make o use of it transparently**.**

Functional unit --- an entity of hardware or software or both, capable of accomplish
a specified purpose.

Gate --- processing device that implements a primitive Boolean operations or processing function (AND, OR) by physically transforming one or more input signals.

Instruction level parallelism – the ability of individual machine language instructions from a
ILP single computer program to be executed in parallel.
The amount of ILP determines the maximum instructions per cycle possible for a particular program.

I/O port --- communication pathway from the CPU to a peripheral device.

Level one (L1) cache --- within-CPU cache when all three cache types are used.

Level two (L2) cache --- primary storage cache implemented within the same chip as a processor.

Level three (L3) cache --- when three levels of primary storage cache are used, the cache implemented outside the microprocessor.

Linear addressing space --- logical view of a secondary storage or I/O device as a sequentially numbered set of storage locations.

Logical access --- access to a storage location within a linear address space.

Loop unrolling – a technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Machine --- a computer, system or processor made up of various components connected
//masin// together to provide a function or perform a task.
//automaat//

Machine instruction --- an instruction that can be directly executed by a computer.

Microarchitecture – the overall design details of a processor that are not visible to the software.

Microcode --- **a**. a collection of microinstructions, comprising part of, all of, or a set of microinstructions.
b. a low-level set of instruction which performs basic, simple tasks. On one level t can be seen as a programming language, whilst at another level it is merely dynamic equivalent to a set of hard-wired circuits.

Model --- an abstract representation of a system or other physical reality.

Multi-chip module (**MCM**) --- an integrated circuit comprising of several chips all packaged within the same package.

Out-of-order execution – execution instructions in a different order than specified by the
OOO execution software in order to improve performance.

Peripheral device --- device on a system bus other than the CPU and primary storage.

Pipeline - the series of steps required to fetch, decode, execute, and store the results of a processor instruction.

Platform --- a standard type of hardware that makes up a particular range of computer.

Platform independence --- the fact that software or network can work or connect to different types of incompatible hardware.

Portability ---- the ease with which applications software can be transferred to an environment
//teisaldatavus// from another while maintaining its capabilities.
//mobiilsus//

Predication --- a mean of conditionally executing instructions.

Processor --- any device capable of performing data transformation **operations**.
In a computer it is a functional unit that interprets and executes instructions.
A processor consists at least an (instruction) control unit and an arithmetic and logic unit.

Processor core --- the part of microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches and memory management unit.

Profiling --- it is a form of a dynamic progam analysis.
The goal of this analysis is to determine which sections of a program to optimize.
Profiling in SW engineering, performance anlysis is the investigation of a program`s behavior using information gathered as the program executes.

Register alias table – a table listing which physical registers are currently holding the values
RAT of the architectural registers.

Register renaming - the microarchitecural algorithm assigning architectural registers to different physical registers in order to eliminated false data dependencies and improve performance.

Reorder buffer - the functional unit in an out-of-order processor responsible for committing
ROB results in the original program order

Scalability ------- the ease with which an existing system's performance can be increased
//mastaabitavus// or decreased as changes in the application demand.
//skaleeritavus//

Spatial locality - the tendency of memory accesses to a particular address to be more likely if nearby addresses have recently been accessed.

Split cache – a scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other with one handling instructions and with one handling data.

Superscalar - processors capable of completing more than one instruction per cycle.

System bus --- bus shared by most or all devices in a MPS.

System architecture --- MPS components and the interaction among them.

System design --- process of determining the exact configuration of all hardware and software components in an information system.

System on silicon (SoC) - all electronics for a complete, working product contained on a single chip. While a computer on a chip includes all the hardware components required to process, a SoC includes the computer and ancillary electronics.

Task --- in a multiprogramming or multiprocessing environment one or more sequences of instructions treated by a control program as an element of work accomplished by a computer.

Temporal locality - the tendency of memory accesses to a particular address to be more likely if that address has recently been accessed.

Thread --- subcomponent of a process that can be independently scheduled and executed.

Trace cache - a cache memory containing already partially decoded instructions, which may be stored in program execution order rather than the order they were original stored in main memory.

Wafer --- a flat round piece of semiconductor used as the starting material for making integral circuit.

Vectorizable --- the property of a computer program, or program segment, that allows for the simultaneous execution of operations on different data values (to accomlish the work in parallel).

Virtual (*adj*) --- pertaining to something that appears to have some characteristics of something else for which it serves as a surrogate.

Virtual machine --- a portion of a computer system or of a computer's time that is controlled by an operating system and functions as though it were a complete system, although in reality the computer is shared with other independent operating systems.

Virtual resource --- resource visible to a user or program, but not necessarily available at all times or physically extant.

WISC (**W**ritable **I**nstruction **S**et **C**omputer) – a CPU design that allows a programmer to add extra machine code instructions using microcode, to customize the instruction set.

Working set --- the set of memory locations referenced over a fixed period.

Workload ---------- **a.** a suite of programs that can be run and whose execution time can be measured.
//kasulik koormus// **b.** the mixture of user programs and operating system commands.
//töömaht// **c.** the amount of work which a computer has to do.

Workspace --- a space on memory which is available for use or is being used currently by an operator.

z-buffer --- an area at memory used to store the z-axis information for a graphics object displayed on screen.

ZISC --- **Z**ero **I**nstruction **S**et **C**omputer
ZISC is a technology based on ideas from artificial neural networks. It is a chip technology based pure pattern matching and absence of instructions in classical sense. The first generation of ZISC contains 36 independent cells that can be thought of as *neurons* (parallel processors).
Each of these compare an input vector of up to 64 bytes with similar vector stored in the cells memory. If the input vector matches the vector in cells memory, the cell "fires". The output signal contains the number of the cell that had matched (or no matches occurred).

Integer Formats (32-bit word)

Format	Binary Range	Decimal Range
Unsigned integer	**0** to $\mathbf{2^{32}-1}$	**0** to **4294967295**
Signed integer	$\mathbf{-2^{31}}$ to $\mathbf{2^{31}-1}$	**-2147483648** to **2147483647**

Floating-point Formats (IEEE std. 754)

Precision	Size (bits)	Sign (bits)	Exponent (bits)	Fraction (bits)	Binary Range	Decimal Range
Single	**32**	1	8	23	2^{-126} to 2^{127}	$1{,}18\times10^{-38}$ to $3{,}40\times10_{38}$
Double	**64**	1	11	52	2^{-1022} to 2^{1023}	$2{,}23\times10^{-308}$ to $1{,}79\times10^{308}$
Extended	**80**	1	16	63	2^{-16382} to 2^{16382}	$3{,}37\times10^{-4932}$ to $1{,}18\times10^{4932}$

Acronyms

A/D [*A to D*] – Analog to Digital
AI [*A-I*] – Artificial Intelligence
ALU [*al-loo* or *A-L-U*] – Arithmetic (and) Logic Unit
AMD [*A-M-D*] – Advanced Micro Devices
ANSI [*an-see*] – American National Standards Institute
API [*A-P-I*] – Application Program(ming) Interface
ARM [*arm*] – Advanced (Acorn) RISC Machines
ASIC [*a-sick*] – Application Specific Integrated Circuit
ASCII [*ass-key*] – American Standard Code for Information Interchange

BCD [*B-C-D*] – Binary Coded Decimal
BEDO [*bee-doh-ram*] – Burst EDO RAM
BGA [*B-G-A*] – Ball-Gird Array
BICMOS [*bi-sea-moss*] – Bipolar Complementary Metal Oxide Semiconductor
BIOS [*bye-oh-sss*] – Basic Input/Output System
BNC [*B-N-C*] – Bayonet Nut Connector

CAD [*cad*] – Computer-Aided Design
CAE [*C-A-E*] – Computer-Aided Engineering
CCD [*C-C-D*] – Charge-Coupled Device
CCITT [*C-C-I-T-T*] – Consultative Committee for International Telephony and Telegraphy
CD-PROM [*C-D-prom*] – Compact Disk-Programmable Read Only Memory
CD-R [*C-D-R*] – Compact Disk-Recordable
CD-RW [*C-D-R-W*] – Compact Disk-Rewritable
CDI [*C-D-I*] – Common Data Interface
CDRAM [*C-D-ram*] – Cached Dynamic Random Access Memory
CGA [*C-G-A*] – Color Graphics Adapter
CISC [*sisk*] – Complex Instruction Set Computer
cps [*C-P-S*] – characters per second
CPU [*C-P-U*] – Central Processing Unit
CMOS [*cee-moss*] – Complementary Metal-Oxide Semiconductor
CRC [*C-R-C*] – Cyclic Redundancy Check
CRT [*C-R-T*] – Cathode-Ray Tube

DAC [*D-A-C*] – Digital-to-Analog Converter
DFM [*D-F-M*] – Design For Manufacturing
DIMM [*dim*] – Dual In-line Memory Module
DIP [*dip*] – Dual In-line Package
DLL [*D-L-L*] – Dynamic Link Library
DMA [*D-M-A*] – Direct Memory Access
dpi [D-P-I] – dote per inch
DRAM [*dee-ram*] – Dynamic Random Access Memory

DSP [*D-S-P*] – Digital Signal Processor
DSU [*D-S-U*] – Data Service Unit
DVD [D-V-D] – Digital Versatile (Video) Disk
DOS [*doss*] – Disk Operating System
DUI [*D-U-I*] – Digital Video Interface
ECC [*E-C-C*] – Error Correction Code
EDORAM [*E-D-O-ram*] – Extended Data-Out Random Access Memory
EDRAM [*E-D-ram*] – Enhanced Dynamic Random Access Memory
EEPROM [*E-E-prom*] – Electrically Erasable Programmable Read-Only Memory
EGA [*E-G-A*] – Enhanced Graphics Adapter
EIDE [*E-I-D-E*] – Enhanced Intelligent Drive Electronics
EISA [*ee-suh*] – Extended Industry Standard Architecture
EMM [*E-M-M*] – Extended Memory Manager
EPP [*E-P-P*] – Enhanced Parallel Port
EPROM [ee-prom] – Erasable Programmable Read-Only Memory
ESDI [*E-S-D-I*] – Enhanced System Device Interface

FAMOS [*famous*] – Floating ate Avalanche MOS
FAT [*fat*] – File Allocation Table
FDD [*F-D-D*] – Floppy Disk Drive
FDDI [*F-D-D-I*] – Fiber Distributed Data Interface
FIFO [*fife-oh*] – Fist In, First Out
FLOPS [*flops*] – FLoating-point Operations per Second
FPU [*F-P-U*] – Floating-Point Unit
FRAM [*F-ram*] – Ferroelectric RAM

GDI [*G-D-I*] – Graphics Device Interface
GIF [*jiff*] – Graphical Interchange Format
GUI [*gooey*] – Graphical User Interface

HDD [*H-D-D*] – Hard Disk Drive

I/O [*eye-oh*] – Input/Output
ICA [*I-C-A*] – Independent Computing Architecture
IDE [*I-D-E*] – Integrated Drive Electronics
IEEE [*eye triple-E*] – Institute of Electrical and Electronics Engineers
IOS [*I-O-S*] – Input Output System
IRQ [I-R-Q] – Interrupt Request
ISA [*eye-suh*] – Industry Standard Architecture
ISO [*I-S-O*] – International Organization for Standardization

JPEG [*jay-peg*] – Joint Photographic Experts Group

kb [*kilobit*] – kilobit
KB [*K-B*] – Kilobyte
Kbps [*K-B-P-S*] – Kilobits per second

LAN [*lan*] – Local area Network
LCD [*L-C-D*] – Liquid Crystal Display
LED [*L-E-D*] – Light Emitting Diode
LIFO [*life-oh*] Last In, Fist Out
LPT [*L-P-T*] – Local Printer Terminal

Mb [*megabit*] – Megabit
MB [M-B] – Megabyte
MCA [*M-C-A*] – Micro Channel Architecture
MCI [*M-C-I*] – Media Control Interface
MFLOPS [*mega-flops*] – Millions of FLoating-point OPerations per Second
MIDI [*middy*] – Musical Instrument Digital Interface
MIPS [*mips*] – Millions of Instructions per Second
MMU [*M-M-U*] – Memory Management Unit
MOS [*moss*] – Metal Oxide Semiconductor
MPEG [*em-peg*] – Motion Picture Experts Group
MTBF [*M-T-B-F*] – Mean time Between Failures

NOP [*no-op*] – No Operation
NUMA [*new-mah*] – Non-Uniform Memory Access
NVRAM [*N-V-ram*] – Non-Volatile RAM

OCR [*oh-see-R*] – Optical Character Recognition

PSRAM [*P-S-ram*] – Pseudo-Static RAM
PAL [*pal*] – Programmable Logic Array
PBSRAM [*P-B-S-ram*] – Pipelined Burst Static RAM
PCB [*P-C-B*] – Printed Circuit Board
PCI [*P-C-I*] – Peripheral Component Interface
PGA [*P-G-A*] – Pin (Pad) Grid Array
PIO [*P-I-O*] – Programmed Input/Output
PIM [*pimm*] – Processor In Memory
PISO [*P-I-S-O*] – Parallel In/Serial Out
PLA [*play*] – Programmable Logic Array
PLD [*P-L-D*] – Programmable Logic Device
PRAM [*pram*] – Parameter RAM
PROM [*prom*] – Programmable Read-Only Memory

RAID [*raid*] – Redundant Arrays of Independent Disks
RAMDAC [*ram-dac*] – Random Access Memory Digital-to-Analog Converter
RISC [*risk*] – Reduced Instruction Set Computer
ROM [*rom*] – Read-Only Memory

SCSI [*scuzzy*] – Small computer System Interface
SDRAM [*S-D-ram*] – Synchronous Dynamic RAM
SGRAM [*S-G-ram*] – Synchronous Graphics RAM

SIMD [*simmed*] – Single Instruction Multiple Data
SIMM [*simm*] – Single In-line Memory Module
SLDRAM [*S-L-D-ram*] – Sync Link DRAM
SOC [*S-O-C*] – System On-a-Chip
SPARC [*spark*] – Scalable Processor Architecture
SRAM [*es-ram*] – Static Random Access Memory
SVGA [*S-V-G-A*] – Super video Graphic Array

TFT [*T-F-T*] – Thin Film Transistor
TIGA [*T-I-G-A*] – Texas Instruments Graphics Architecture
TTL [*T-T-L*] – Transistor-Transistor Logic

UART [*you-art*] – Universal Asynchronous Receiver/Transmitter
UDMA [*ultra-D-M-A*] – Ultra Direct Memory access
{a DMA version that is twice as fast as the original}
ULSI [*U-L-S-I*] – Ultra Large Scale Integration
UPS [*U-P-S*] – Uninterruptible Power Supply
USB [*U-S-B*] – Universal Serial Bus

VESA [*vee-suh*] – Video Electronics Standard Association
VMS [*V-M-S*] – Virtual Memory System
VRAM [*vee-ram*] – Video RAM

WAN [*wh-an*] – Wide Area Network
WORM [*worm*] – Write Once Read Memory
WRAM [*double-you-ram*] – Windows RAM

XGA [*X-G-A*] – Extended Graphics Array (Adapter)

1G [*one-G*] – First Generation
2G [*two-G*] – Second Generation
3G [*three-G*] – Third Generation