

universidade de aveiro



Arquitecturas de Alto Desempenho

Computer Abstractions and Technology

António Rui Borges

Summary

- *Information society*
 - *Highlights of information revolution*
- *Computer architecture vs. computer organization*
- *Structure and function*
- *Historic perspective*
- *Classes of computer systems*
- *Classes of parallelism*
- *Amdahl's law*
- *Quantitative principles of computer design*
- *Power and energy in integrated circuits*
- *Dependability*
- *Measuring performance*
- *The processor performance equation*
- *Suggested reading*

Information society - I

The presence of computer systems in today's society is ubiquitous.

They make up the basic building block of *internet* – the communications network infrastructure that connects computers worldwide, giving rise to the concept of *global village* through which people anywhere in the planet communicate with people in any other place by sending and/or receiving messages at almost real time.

They are also an integral part of the smart phones and tablets one owns and stand for the laptops and the desktops one uses at home or at the workplace. Through them, one can work from home, access a myriad of services such as participating in a video conference, banking and accounting, online shopping, paying bills remotely and contacting government agencies.

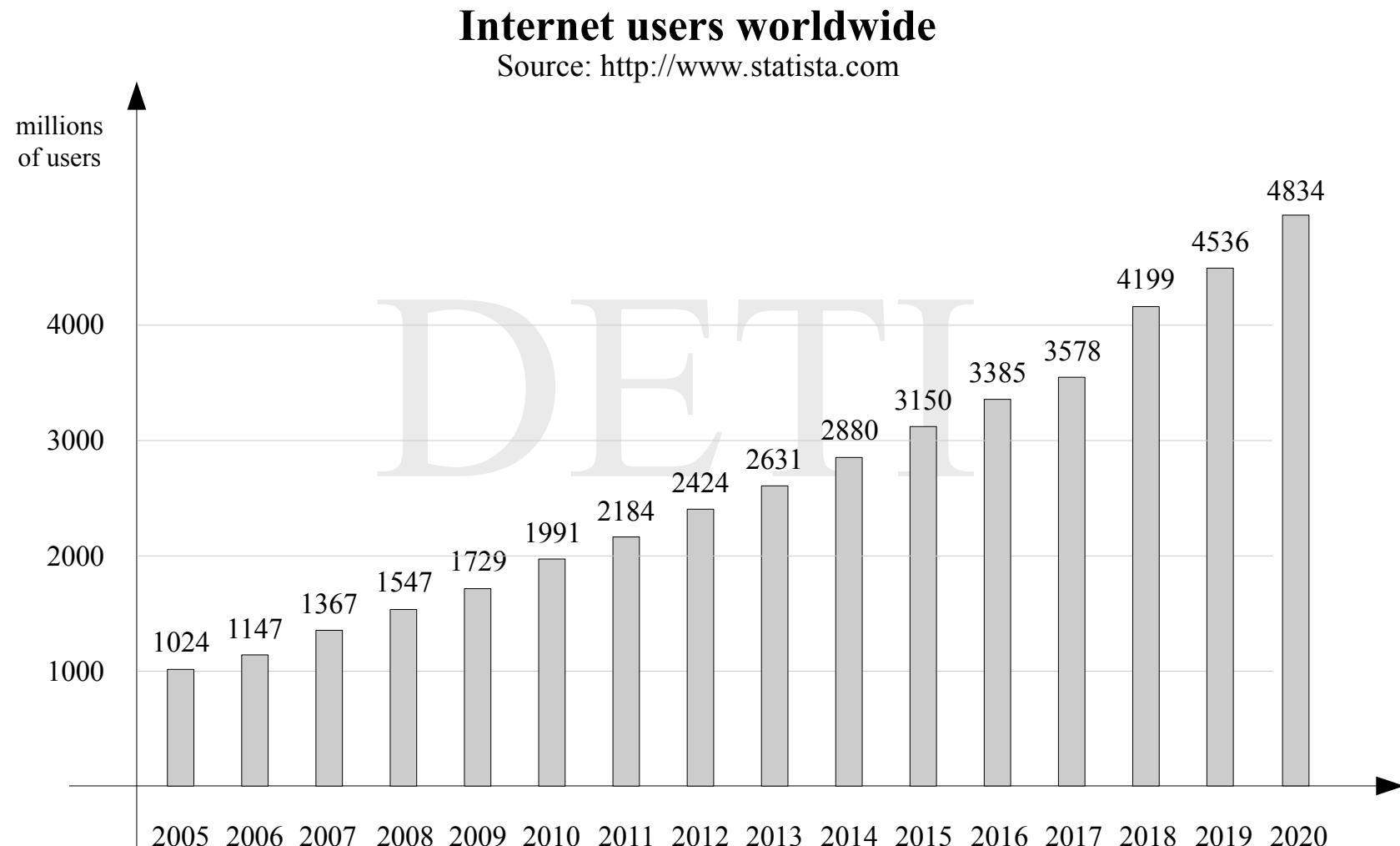
Industry has enormously benefited by their use. Automation of production lines has improved the efficiency of the manufacturing processes and lowered the prices of the products. By incorporating them, products themselves have become more suited to one's needs and more versatile.

Information society - 2

Computer systems have introduced profound changes to society and the rate of change is accelerating. For instance, it is foreseen for the next few years the possibility of eliminating completely the need of physical money (coins and bank notes) and replacing it by pure electronic transactions, and of building automobiles that can drive themselves for safety reasons and comfort.

Such is their impact on day to day living that it is often said humanity is presently enduring another civilizational revolution – the *information revolution*, as compared with the *agricultural revolution* that has lead the transition from hunting and gathering to settled agriculture about 12000 years ago, and the *industrial revolution* that gave rise to new more efficient manufacturing processes in the second half of the XVIII century.

Highlights of the information revolution - 1



Highlights of the information revolution - 2

Internet users worldwide (June 2022)

Source: <http://internetworkstats.com>

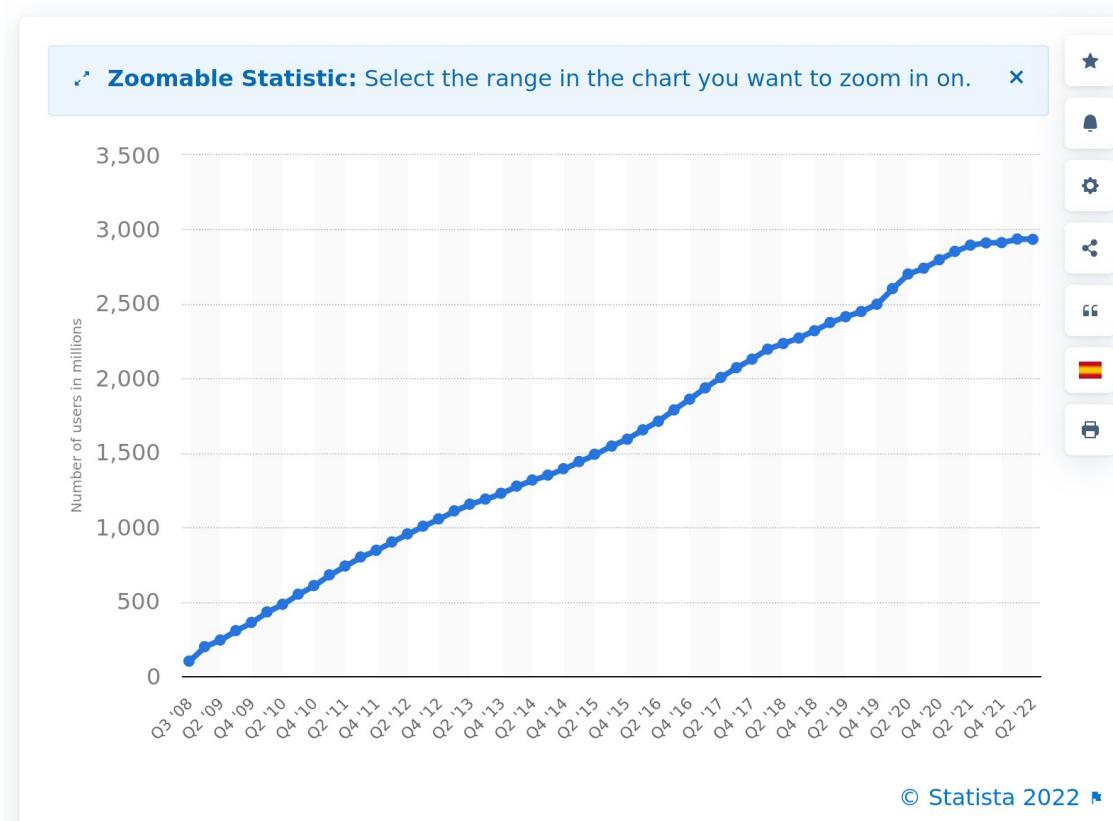
<i>World Regions</i>	<i>Population (est.)</i>	<i>Internet Users</i>	<i>Frac. Population</i>
Africa	1 394 588 547	652 865 628	46,8%
Asia	4 352 169 960	2 934 186 678	67,4%
Europe	837 472 045	750 045 495	89,6%
Portugal	10 148 962	8 841 100	87,1%
Middle East	268 302 801	211 796 760	78,9%
North America	374 226 482	349 572 583	93,4%
Latin America / Caribbean	664 099 841	543 396 621	81,8%
Oceania / Australia	43 602 955	31 191 971	71,5%
<i>World Total</i>	<i>7 934 462 631</i>	<i>5 473 055 736</i>	<i>69,0%</i>

Highlights of the information revolution - 3

Number of monthly active Facebook users worldwide

Source: <http://www.statista.com>

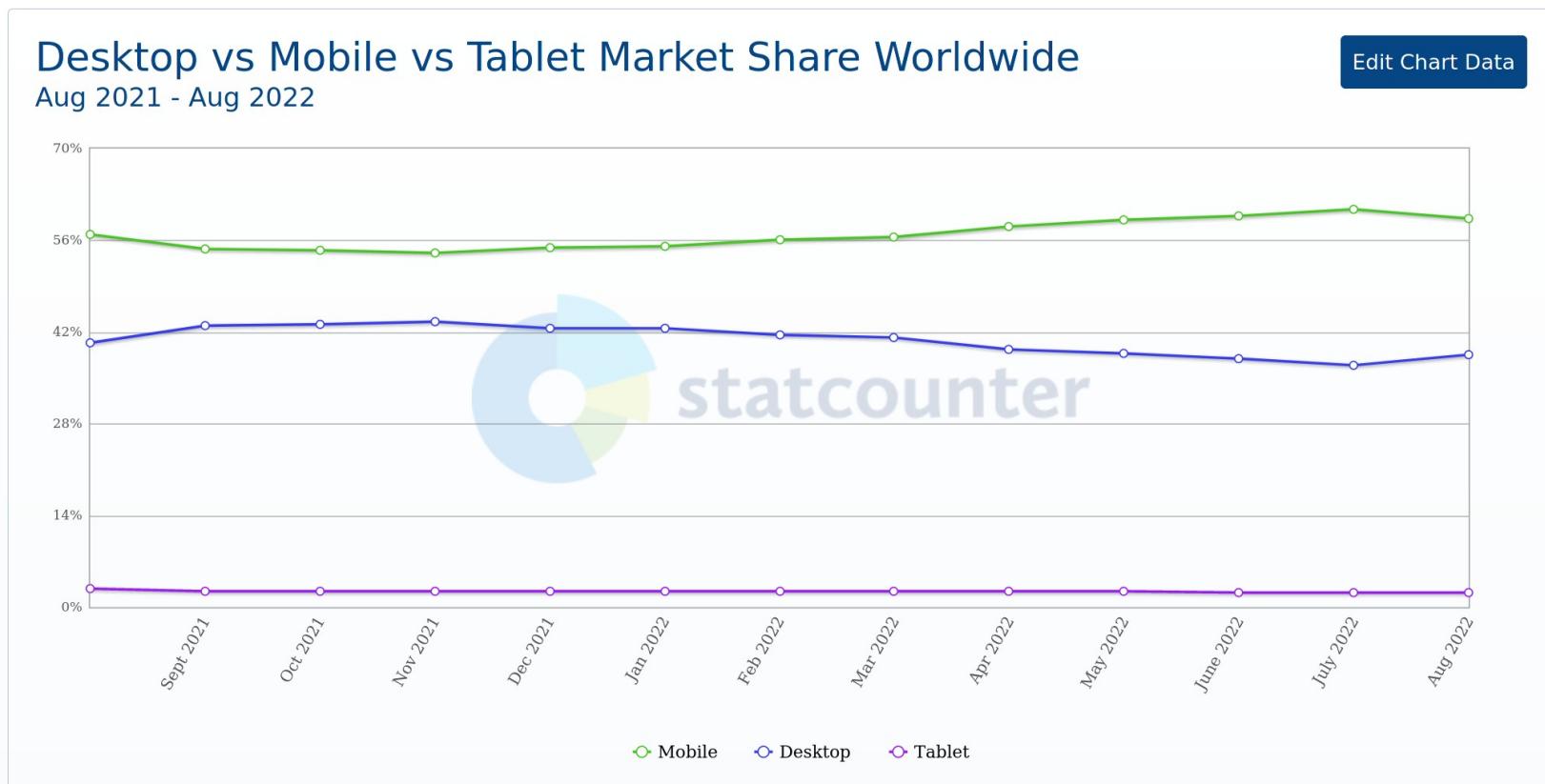
(in millions)



Highlights of the information revolution - 4

Type of access to internet

Source: <http://gs.statcounter.com>



Highlights of the information revolution - 5

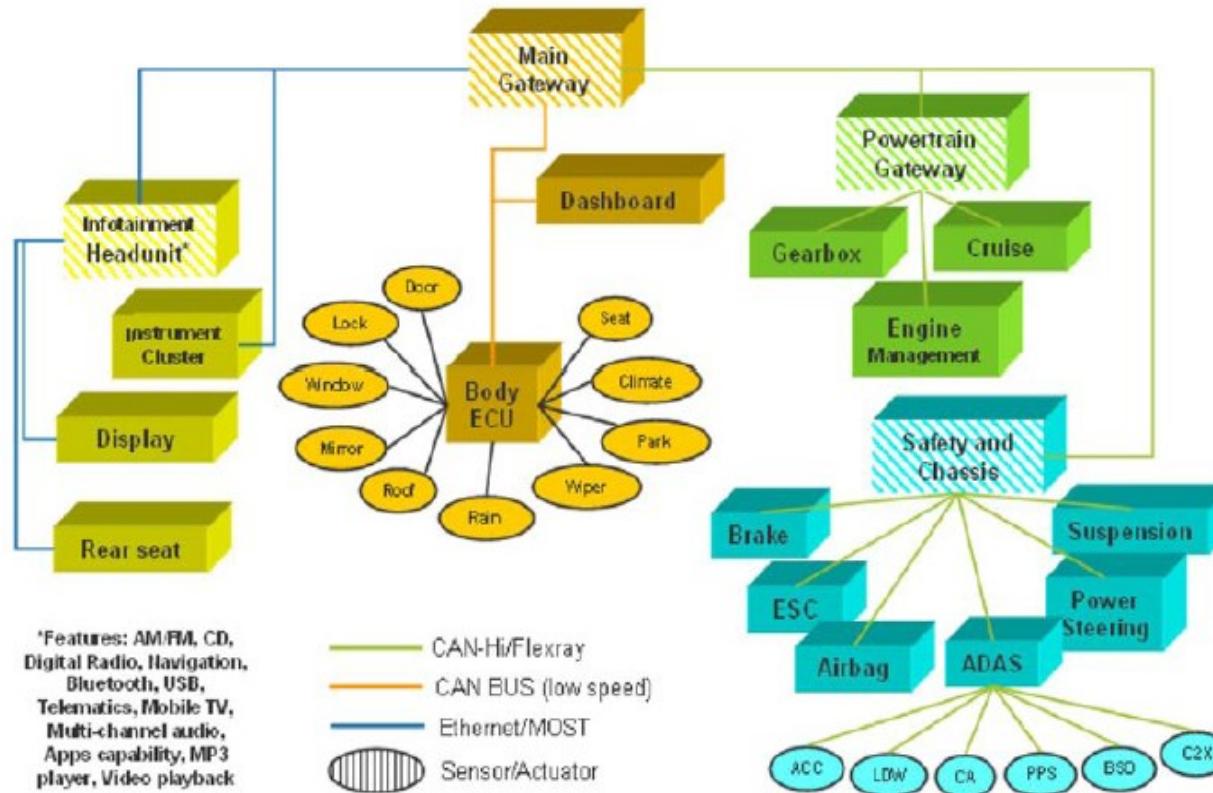
ECU Domain description

Domain	Key Information	Comments/Examples
Powertrain: 5-10 ECUs	Controls the car's power and its distribution to the wheels	<ul style="list-style-type: none">• Engine control• Transmission control
Chassis: 3-5 ECUs	Controls the functions that guides the car's direction and speed	<ul style="list-style-type: none">• Steering control• Brake control• Suspension control
Safety, Driver Assist and ADAS 5-10 ECUs	Controls the car's safety systems. Many new systems are emerging	<ul style="list-style-type: none">• Air bag control• Seat belt control• Driver assist systems-ADAS
Body and Comfort: 10-30 ECUs	Controls the driver and passengers' convenience and comfort systems. Includes dash board and related controls	<ul style="list-style-type: none">• Heater and air conditioning• Windows and seat control• Window wipers• Instrument cluster display
Infotainment: 5-7 ECUs	Controls the entertainment and information systems used by driver and passengers	<ul style="list-style-type: none">• Radio and music systems• Navigation systems• Telematics and mobile phone

Source: *Is Europe in the Driver's Seat? The Competitiveness of the European Automotive Embedded Systems Industry*, Juliussen E. and Robinson R., Institute for Prospective Technological Studies, 2010, EUR 24601 EN

Highlights of the information revolution - 6

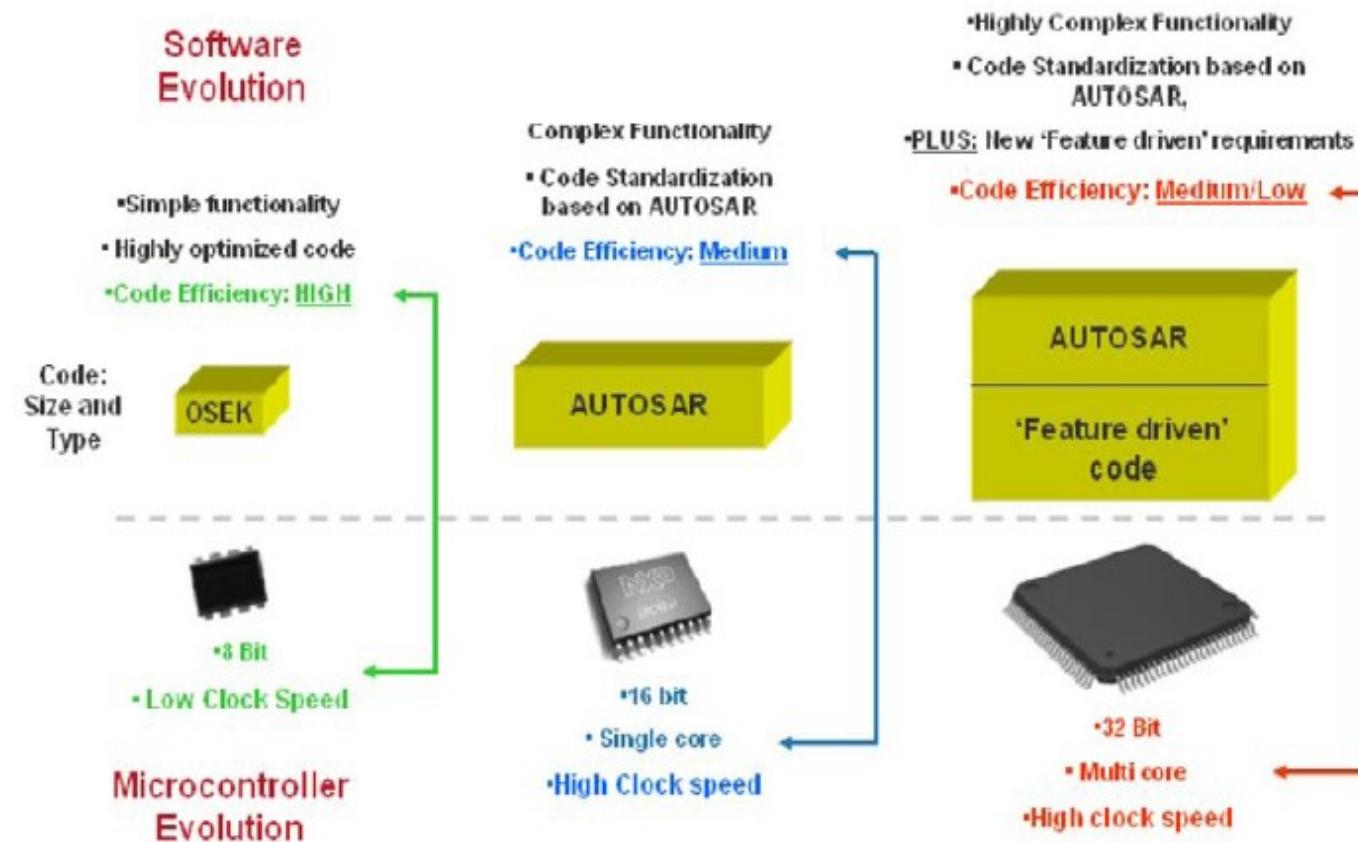
Domain controller architecture (from 2010 onwards)



Source: *Is Europe in the Driver's Seat? The Competitiveness of the European Automotive Embedded Systems Industry*,
Juliussen E. and Robinson R., Institute for Prospective Technological Studies, 2010, EUR 24601 EN

Highlights of the information revolution - 7

Microcontroller evolution (2000 to 2020)



Source: *Is Europe in the Driver's Seat? The Competitiveness of the European Automotive Embedded Systems Industry*,
Juliussen E. and Robinson R., Institute for Prospective Technological Studies, 2010, EUR 24601 EN

Computer architecture vs. computer organization - 1

Computer architecture – it refers to those attributes of a computer system which are visible to the programmer, that is, to those attributes which have a direct impact on the logical execution of a program.

Architectural attributes include the instruction set, the number and the size of the processor internal registers, the format of the different data types, the memory addressing modes and the I/O mechanisms.

Computer organization – it refers to the role of internal operational units and to the ways they interconnect to implement the architectural specification.

Organizational attributes include those hardware details which are transparent to the programmer such as control signals, interfaces between processor and memory and between the computer system and I/O devices (peripherals) and the memory technology used.

Computer architecture vs. computer organization - 2

The distinction between architecture and organization has been, and still is, very important. Many computer manufacturers offer a family of computer models, all with the same architecture, but with differences in organization, that is, representing different implementations of the same architecture, so that each model has different price and performance characteristics. Thus, a particular architecture may span many years and encompass a number of different computer models, its organization changing with changing technology.

A notable example in this sense is the IBM System 370 architecture which was first introduced in 1970 and included a number of different models. The customer with modest requirements could start by buying a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model without having to abandon the software that had already been developed. Over the years, IBM has produced many new models with improved technology to replace the older models, offering the customer greater speed, lower cost, or both. These newer models retained the same architecture, thus protecting the customer's software investment. Remarkably, the System 370 architecture, with a few enhancements, has survived till today as the architecture of IBM's mainframes product line.

Computer architecture vs. computer organization - 3



IBM's z13 mainframe (Augusto Menezes/Feature Photo Service/IBM)

The z13 is designed to be capable of real time encryption for mobile transactions, with real time analytics processing to identify trends and help detect fraud, while the system is able to scale up to process an impressive 2.5 billion transactions per day, according to IBM. (Source: <http://www.v3.co.uk>)

Computer architecture vs. computer organization - 4

For microcomputers, the relationship between architecture and organization is very close. Changes in technology not only influence organization but also result in the introduction of more powerful and more complex architectures. Generally, there is less of a requirement for generation-to-generation compatibility for these smaller machines.

Structure and function - 1

A computer system is a very complex system. The key point to design and/or to describe a complex system is to use *abstraction* to express the system, or one of its parts, at different levels of representation in a hierarchical manner. At a given level, the details of lower levels are hidden so that the image that is presented is based on the concept of *what one needs to know*. By concentrating at what is relevant at each moment, fewer details have to be considered and interrelated and one becomes more productive in designing and/or gets a clearer picture and a better understanding of what is described.

At each level, one needs to be concerned with structure and function

- ***structure*** – how many components there are and the way in which they are interconnected
- ***function*** – the operation of each individual component as part of the whole.

Structure and function - 2

The basic functions of a computer system are: *inputting data, outputting data, processing data, storing data* and *control*.

The computer system must be able to *process data*. Data takes a wide variety of forms and the range of processing requirements is broad. There are, however, only a few fundamental methods or types of data processing.

The computer system must also be able to *transfer data* between itself and the outside world. The operating environment consists of devices that either serve as sources or destinations of data. When data are received from or delivered to a device that is directly connected to a computer system, the process of doing this is known as performing *input – output* (I/O) and the device is referred to as a *peripheral*. When data are transferred over longer distances, to or from a remote device, the process is known as *data communication*.

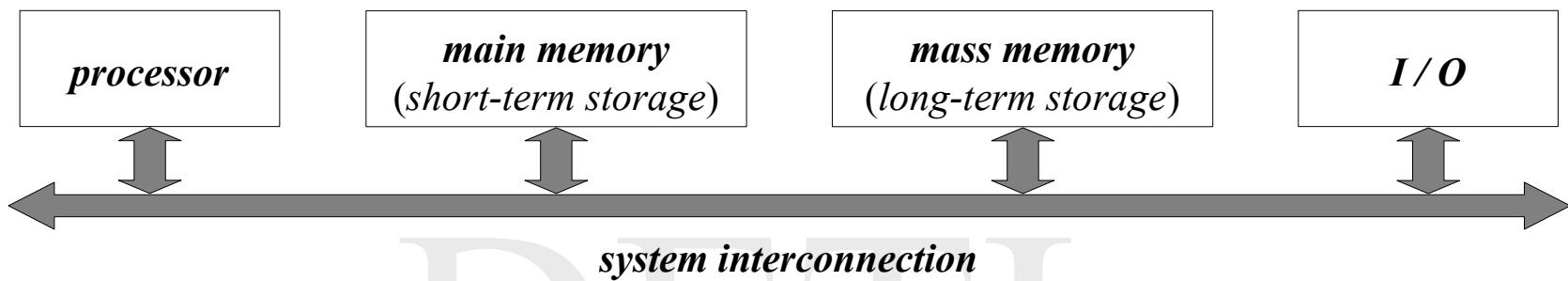
Structure and function - 3

It is essential for the computer system to *store data* too. Even if the computer is processing data on the fly (that is, data are inputted, get processed and the results are immediately outputted), the computer system must temporarily store at least those pieces of data that are being worked upon at any given moment. Thus, there is a *short-term data storage* function. Equally important, the computer system requires a *long-term data storage* function for data files to be stored and subsequently be retrieved and updated.

Finally, these functions must be *controlled* to achieve some useful work. This control is ultimately exercised by the individual (*programmer*) who provides the computer system with *instructions*, but within the computer system itself a *control unit* must manage the available resources and orchestrate the performance of its functional parts in response to those instructions.

Structure and function - 4

At top level, the structure of a computer system may be perceived as



- ***processor*** (or ***central processing unit*** – CPU) – controls the operation of the computer and performs its data processing functions
- ***main memory*** – stores data during processing; it has a *volatile* nature
- ***mass memory*** – stores data in-between processing runs and allows that large amounts of data be retrieved and eventually updated during processing; it has a *non-volatile* nature and is perceived as a special I/O device
- ***I/O*** – moves data between the computer system and its external environment
- ***system interconnection*** – ensures that data transfer takes place among the components; it is usually implemented as a *bus*.

Structure and function - 5

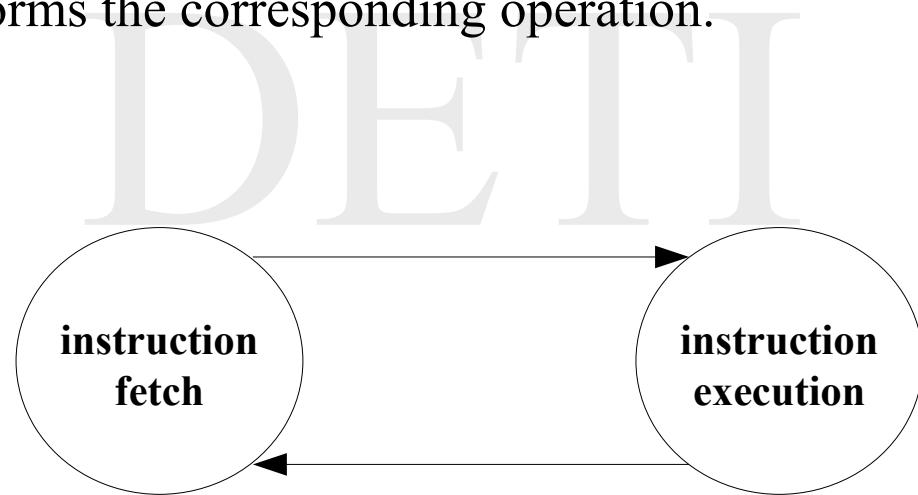
In order to have the computer system to carry out a specific task, one has to supply a group of instructions that taken together constitute the *program* to be executed. The key idea here is how to represent instructions?

Suppose they could be represented in a form suitable for being stored in the main memory alongside the data, thus constituting a *special* kind of data in some abstract sense. Then, a computer system could get its instructions by reading them from the main memory and a program could be set or altered by setting the values of that portion of the memory.

This idea, developed independently by John von Neumann and Alan Turing, has come to be known as the *stored-program concept* and has been universally adopted by computer designers. This is the reason why computer systems are sometimes called *von Neumann machines*.

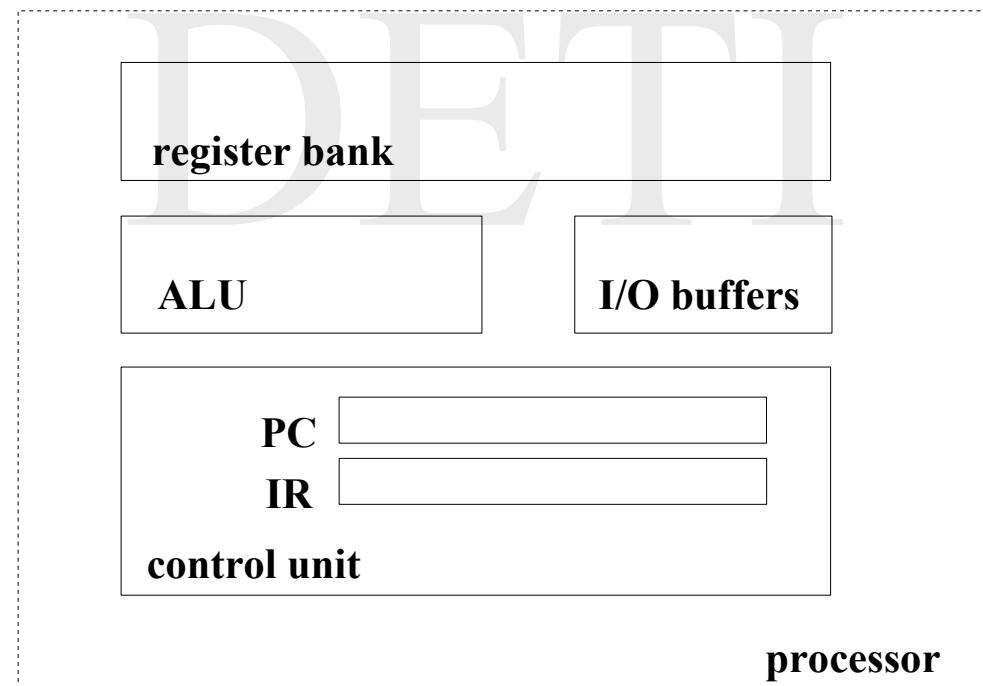
Structure and function - 6

In this sense, although a computer system is a very complex digital system, it may be perceived as a system which toggles continuously between two basic internal states: *instruction fetch*, where the processor accesses memory to get the next instruction, and *instruction execution*, where the processor decodes the just retrieved instruction and performs the corresponding operation.



Structure and function - 7

In a simplified way, the processor itself may be seen of consisting of a *control unit*, which takes mainly care of the *instruction fetch* and the *instruction decoding* phases; of an *arithmetic/logic unit*, which performs the prescribed operations; of a *register bank*, which stores temporary data; and of *I/O buffers*, which enable communication with the other components of the computer system.



Structure and function - 8

The *instruction set* of any processor always comprises instructions of the type

- *data movement* – transfer of data between some register of the register bank and main memory or some I/O controller
- *arithmetic / logic* – arithmetic instructions (add, subtract, multiply, divide), either in fixed or floating point format, logic instructions (not, and, or, x-or) and shifting / rotating register contents
- *branching* – modifying the strict sequentiality of instruction execution, either unconditionally or dependent on some condition
- *subroutine calling* – execution of subprograms (autonomous code segments within the whole group of supplied instructions), either unconditionally or dependent on some condition.

Historic perspective - 1

Computer systems performance has made an incredible progress since the time the first general purpose electronic computer was built. This rapid pace of improvement was due not only to advances in the technology of integrated circuits, but also to advances in computer design.

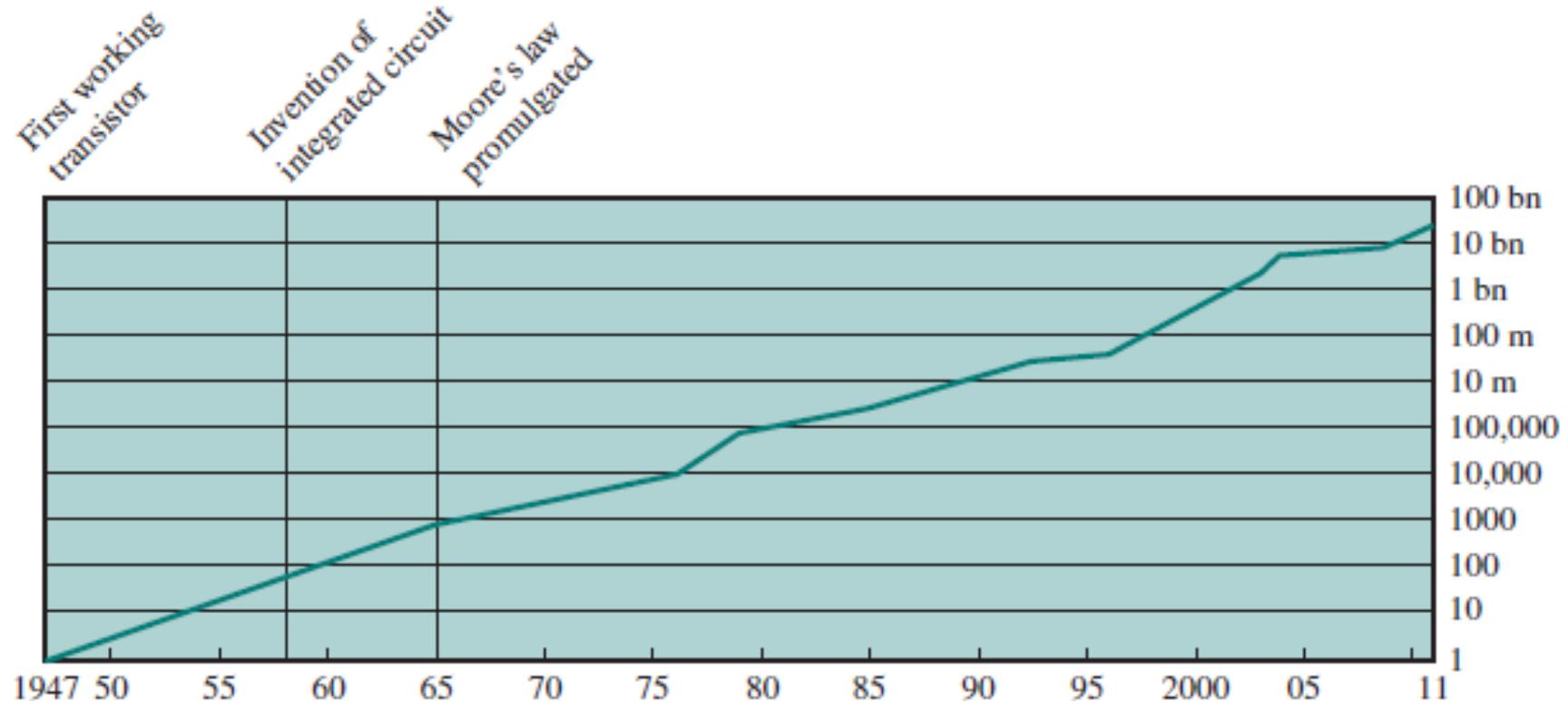
On the technological side, Gordon Moore, co-founder of Fairchild Semiconductor Corporation and Intel, observed in 1965 that the number of transistors that could be put on a single chip was doubling every year and correctly predicted that this pace would continue into the near future. This observation became known as the *Moore's Law*.

The pace went on year after year and decade after decade ever since. It has slowed somewhat to a doubling every two years in the 1970s and, more recently, 2015, Intel has announced that the pace is presently about a doubling every two and half years.

Historic perspective - 2

Growth in transistor count on integrated circuits

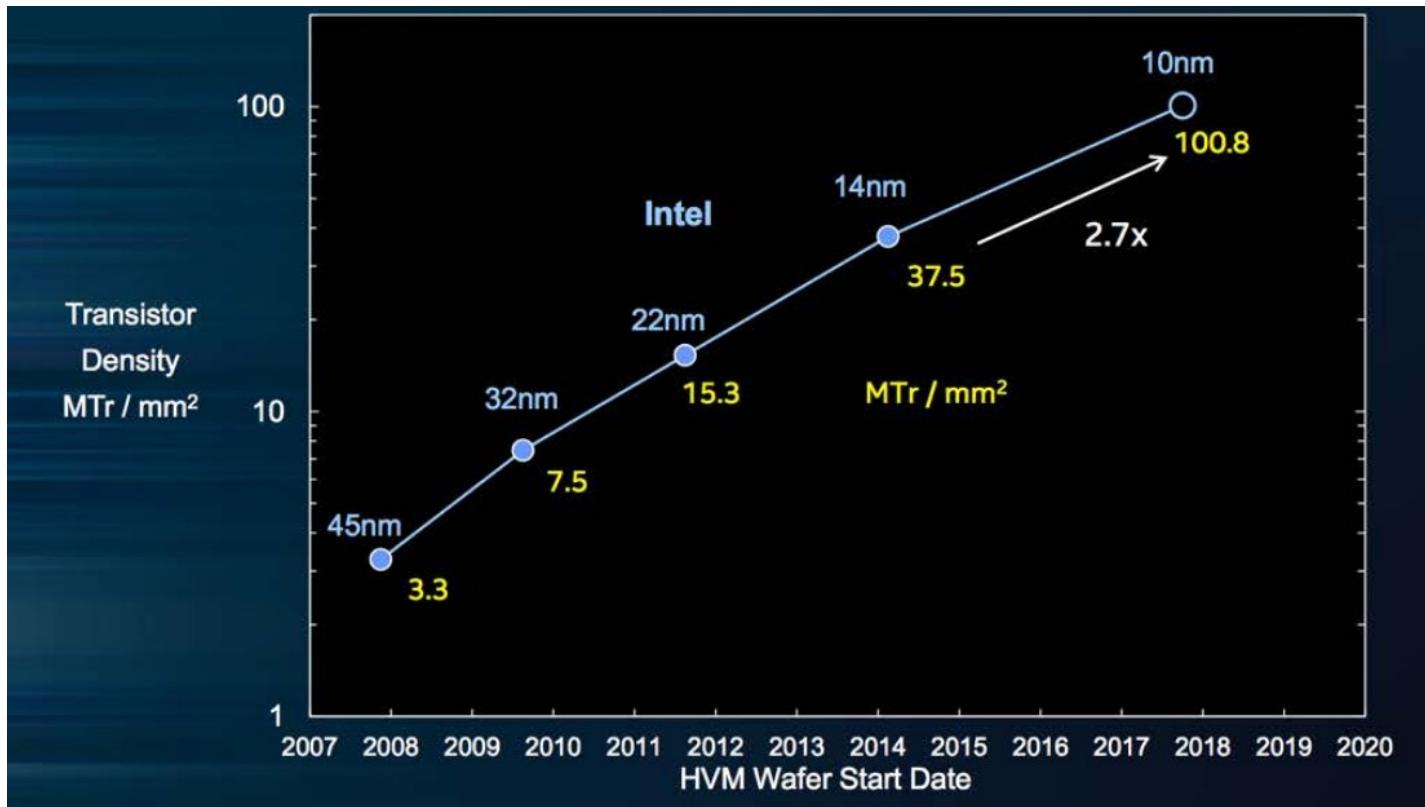
Source: Computer Organization and Architecture: Designing for Performance



Historic perspective - 3

Logic transistor density

Source: Intel's 10 nm Technology (Intel News Fact Sheet)



Historic perspective - 4

The middle 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuits technology and the cost advantage that resulted from a mass-produced microprocessor have led to an increasing fraction of the computer systems being based on microprocessors. In addition, two significant changes in the computer marketplace make it easier than ever before to succeed commercially with a new architecture

- the virtual elimination of assembly language programming reduced the need for object code compatibility
- the creation of standardized, vendor independent operating systems, such as Unix and later its clone Linux, lowered the cost and the risk of bringing out new processor hardware.

These changes enabled the successful development of a new set of architectures based on simpler instructions, known as RISC (Reduced Instruction Set Computers), in the early 1980s. The RISC processors focused the designers attention on two performance techniques: the exploitation of *instruction level parallelism* (first, using pipelining and, later, using multiple instruction issue) and the systematic application of caches to speed up the access of the processor to instructions and data (first, in simple forms and, later, using more sophisticated organizations and optimizations).

Historic perspective - 5

The effect of the combination of these architectural and organizational enhancements was fourfold

- it has significantly improved the computing power made available to users (the highest performant microprocessors of the day outperformed the supercomputers built 10 years previously)
- new classes of computer systems appeared due to the dramatic improvement in cost-performance: personal computers and workstations came into being in the 1980s and smart phones and tablet computers became the preferred primary computing platform of many people in the last decade
- continuing progress in semiconductor manufacturing, as predicted by Moore's Law, has led to the dominance of microprocessor-based computer systems across the entire range of computer design
- it brought forward a deep impact on software development: first, by allowing programmers to trade performance for productivity in many applications through the use of the object-oriented paradigm; second, performance is being pursued by replacing interpreters with just-in-time compilers and trace-based compiling for the traditional compiler and linker approach; third, the present popularity of Software as a Service (SaaS) over the Internet for running applications.

Historic perspective - 6

Since 2003 single-processor rate of performance improvement has decreased due to the barrier imposed by the allowed maximum amount of power dissipation of air-cooled chips and the lack of new ideas for instruction-level parallelism to be exploited efficiently.

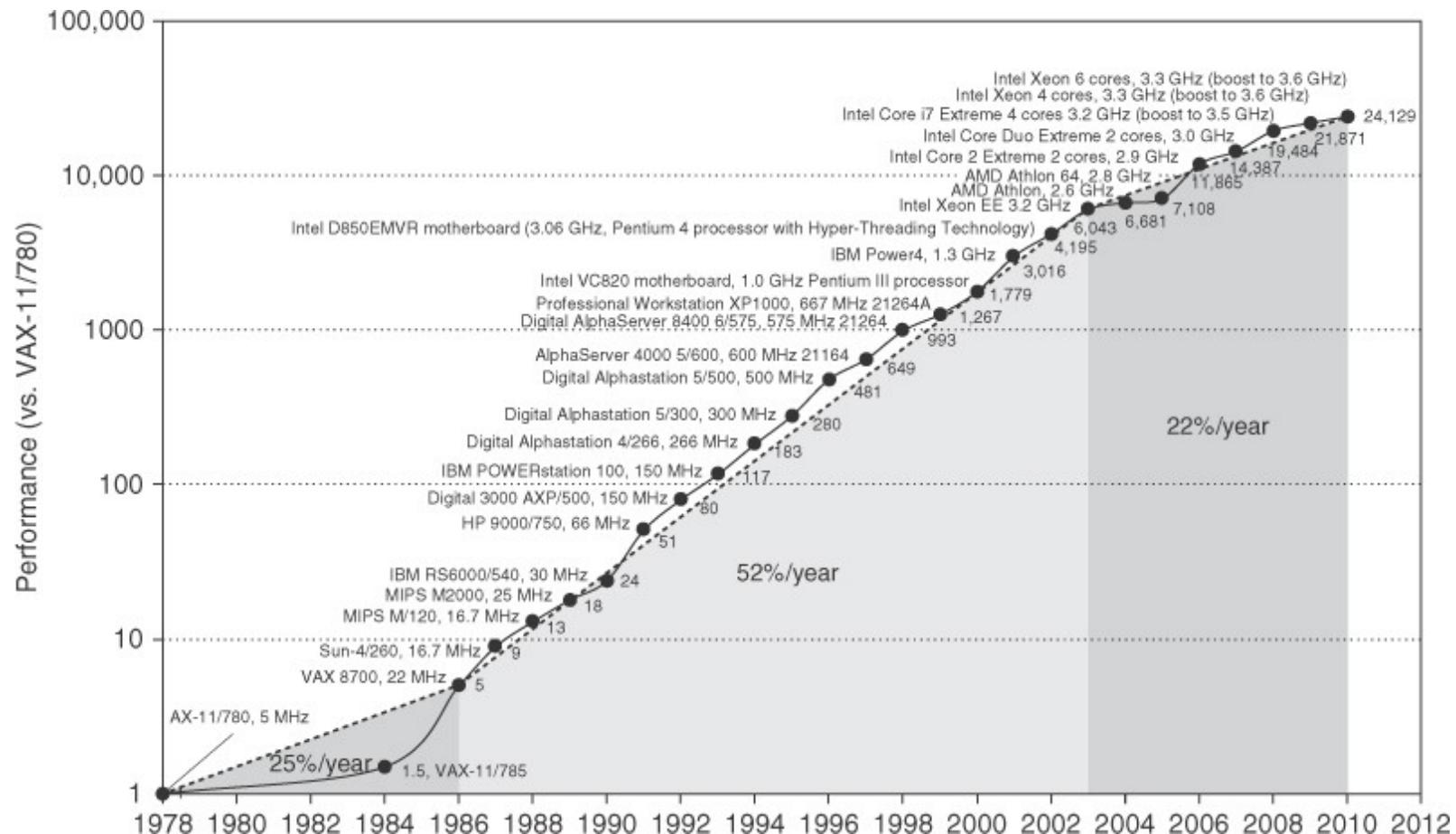
In 2004, Intel cancelled its high performance uniprocessor projects and joined other chip manufacturers in declaring that the road to higher performance would be via multicore chips rather than via faster uniprocessors.

This means that the emphasis now is placed not only in *instruction-level parallelism* (ILP), but also in *data-level parallelism* (DLP), *thread-level parallelism* (TLP) and *request-level parallelism* (RLP). Whereas hardware and compiler techniques exploit ILP implicitly without the programmer's attention, DLP, TLP and RLP are explicitly parallel and require the restructuring of applications in order to be efficiently exploited.

Historic perspective - 7

Growth in processor performance since the late 1970s

Source: Computer Architecture: A Quantitative Approach



Classes of computer systems - 1

Changes that are taking place in computer use have led to five different application markets, each characterized by specific requirements and technologies

- *personal mobile devices (PMD)* – they comprise a collection of wireless devices with multimedia user interfaces, such as smart phones and tablet computers; cost is a prime concern since they are primarily consumer products; although the emphasis on energy efficiency is frequently driven by the use of batteries, the need to use less expensive packaging and the absence of a fan for cooling also limit total power consumption; energy, size and weight requirements lead to the use of flash memory for mass storage; applications are often web-based and media-oriented
- *desktop computing* – they span from low-end laptops or notebooks to high-end, heavily configured workstations; the focus is on *price-performance optimization*, both in raw computing and graphics; as a result, the newest, highest performant microprocessors and cost-reduced microprocessors often appear first in desktop systems; computing tends to be well characterized in terms of applications and benchmarking, though the increasing use of web-centric, interactive applications poses new challenges

Classes of computer systems - 2

- *servers* – they represent the backbone of large-scale enterprise computing, replacing the traditional mainframe as the provider of larger-scale and more reliable file and computing services; they came into being in the 1980s when terminals were replaced by desktops as the primary interface to central services; both availability and scalability are critical, because servers are supposed to be always active and grow in response to an increasing demand for the services they support or an increase in functional requirements (thus, the ability to scale up the computing capacity, the main memory, the mass storage and the I/O bandwidth is crucial); servers are also designed for efficient throughput – responsiveness to individuals requests remains important, but overall efficiency and cost-effectiveness, as defined by the number of requests that can be handled per unit of time, are the key metrics to be taken into account

Classes of computer systems - 3

- *clusters / warehouse-scale computers* – *clusters* are collections of desktop computers, or servers, connected by local area networks to act as a single larger computer; each processing node runs its own [network] operating system and communicate with the others through a networking protocol; the largest clusters are called *warehouse-scale computers* (WSCs); price-performance and power are critical; availability is also critical, but contrary to servers where it is ensured by integrated computer hardware, WSCs use redundant inexpensive components as the building blocks, relying on a software layer to detect and isolate the many failures which occur; focus is placed on interactive applications, large-scale storage, dependability and high Internet bandwidth – they represent the building block of what is now called the *cloud*
supercomputers and, in general, *high-performance computing* (HPC) are related to WSCs, but differ by emphasizing floating-point performance and by running large communication-intensive batch programs whose execution takes weeks at a time; very fast internal networks are critical here
- *embedded computers* – they are found in everyday machines, 'intelligent' instruments and consumer products; they are related to PMD computers, but contrary to them, they do not run externally developed software; price is the key factor – performance requirements do exist, but the primary goal is meeting the performance needs at a minimum price.

Classes of computer systems - 4

Classes of computer systems and their systemic characteristics

Source: Adapted from Computer Architecture: A Quantitative Approach

<i>Feature</i>	<i>PMD</i>	<i>Desktop</i>	<i>Server</i>	<i>C/WSC</i>	<i>Embedded</i>
<i>System Price</i>	\$100 - \$1 000	\$300 - \$2 500	\$5 000 - \$10 000 000	\$100 000 - \$200 000 000	\$10 - \$100 000
<i>Microprocessor Price</i>	\$10 - \$100	\$50 - \$500	\$200 - \$2 000	\$50 - \$250	\$0.01 - \$100
<i>Critical Design Issues</i>	cost, energy, performance, responsiveness	price-performance, energy, graphics-performance	throughput, availability, scalability, energy	price-performance, throughput, energy	price, energy, application-specific performance

Classes of parallelism - 1

Parallelism at various levels is the prevailing driving force of computer design across all classes of computers, with energy and cost being the primary constraints. There are basically two kinds of parallelism in applications

- *data-level parallelism* (DLP) – it arises when there are multiple data items that can be processed at the same time
- *task-level parallelism* (TLP) – it arises when the task to be carried out can be divided into subtasks that operate independently.

Computer hardware in turn can exploit these two kinds of application parallelism in four different ways

- *instruction-level parallelism* – data-level parallelism is exploited with compiler help using ideas like pipelining, speculative execution and multiple issue
- *through special hardware* – vector architectures and graphic processor units (GPUs) exploit data-level parallelism by applying the same instruction to a collection of data in parallel
- *thread-level parallelism* – data-level and task-level parallelism may be both exploited in a tightly-coupled model that allows for interaction among concurrent threads
- *request-level parallelism* – parallelism is exploited among essentially loosely-coupled tasks specified by the programmer or the operating system.

Classes of parallelism - 2

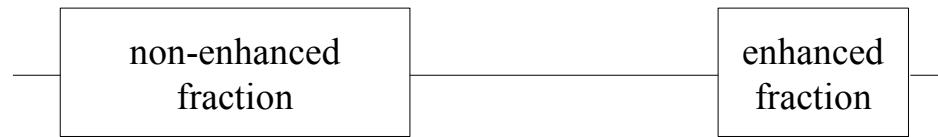
In the 1960s, Michael Flynn studied the parallel computing efforts which were made so far and found a classification that is still popular today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor and placed all computers in one of the four categories

- *single instruction – single data streams* (SISD) – it corresponds to the uniprocessor; nevertheless, instruction-level parallelism can still be exploited
- *single instruction – multiple data streams* (SIMD) – the same instruction is executed by multiple processing units using different data streams; this category comprises vector architectures, multimedia extensions to standard instruction sets and GPUs
- *multiple instruction – single data streams* (MISD) – multiple instructions are executed upon the same piece of data; no commercial multiprocessor of this type has been built yet (although if the *piece of data* is considered to represent a *data vector*, then *systolic arrays* may be reasoned to fall in this category)
- *multiple instruction – multiple data streams* (MIMD) – it targets task-level parallelism where each processor runs its own program on its own data; data-level parallelism may also be exploited, although the overhead is likely to be higher than in SIMD; processor multicores fall in this category.

Amdahl's Law - I



execution time for the entire task without using the improvement



execution time for the entire task using the improvement

Amdahl's Law - 2

The performance gain that can be obtained by improving some feature of a computer system can be estimated by the *Law of Amdahl*. Amdahl stated in 1967 that *the speed up to be gained from adopting some faster mode of execution is limited by the time fraction of all operations that can not be enhanced* and is expressed by the formula

$$\begin{aligned}\text{speedup}_{\text{overall}} &= \frac{\text{execution time for the entire task without using the improvement}}{\text{execution time for the entire task using the improvement}} = \\ &= \frac{1}{(1 - \text{frac}_{\text{enhanc}}) + \frac{\text{frac}_{\text{enhanc}}}{\text{speedup}_{\text{enhanc}}}},\end{aligned}$$

where $\text{frac}_{\text{enhanc}}$ is the time fraction in the original computer system which can be converted to take advantage of the faster mode of execution and $\text{speedup}_{\text{enhanc}}$ is the speed up to be gained locally by the adoption of the faster mode of execution.

Amdahl's Law - 3

The processor used in a web server is to be changed in order to speed up operations. The new processor is 10 times faster than the original one. Assuming that presently the processor is busy 40% of the time and is waiting for I/O 60% of the time, what is the overall speed up gained if the replacement takes place?

$$\begin{aligned} \text{frac}_{\text{enhanc}} &= 0.4 \quad \wedge \quad \text{speedup}_{\text{enhanc}} = 10 \Rightarrow \\ \Rightarrow \text{speedup}_{\text{overall}} &= \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56 . \end{aligned}$$

Thus, Amdahl's Law is a law of diminishing returns!

$$\lim_{\text{speedup}_{\text{enhanc}} \rightarrow +\infty} \text{speedup}_{\text{overall}} = \frac{1}{1 - \text{frac}_{\text{enhanc}}} .$$

Amdahl's Law - 4

Amdahl's Law is particularly useful in comparing the overall system performance of different alternatives.

A common operation required in graphics processors is *square root*. Suppose that floating point square root (FPSQRT) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQRT hardware and speed up 10 times this operation. The other alternative is just to make all floating point operations run faster by a factor of 1.6. Note that floating point operations account for half of the execution time of the application. Compare these two design alternatives.

$$\text{speedup}_{\text{FPSQRT}} = \frac{1}{0.8 + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{speedup}_{\text{FP}} = \frac{1}{0.5 + \frac{0.5}{1.6}} = \frac{1}{0.81} = 1.23 .$$

Quantitative principles of computer design

Some guidelines are useful in the design of computer systems

- *take advantage of parallelism* – parallelism is one of the most important methods for improving performance; one can resort to redundancy to increase *dependability* and/or sometimes make operations go faster by simply adding more resources – *scalability*, or by eliciting the underlying *concurrency*; this can be done at various abstraction levels: system, individual processor or low-level digital design
- *take advantage of the principle of locality* – programs tend to reuse instructions and data that they have recently used
- *focus on the common case* – in making a design trade-off, favor the frequent case over the infrequent one; the frequent case is often simpler to optimize and can be made more rewarding; it works well not only for performance, but also for resource allocation and power.

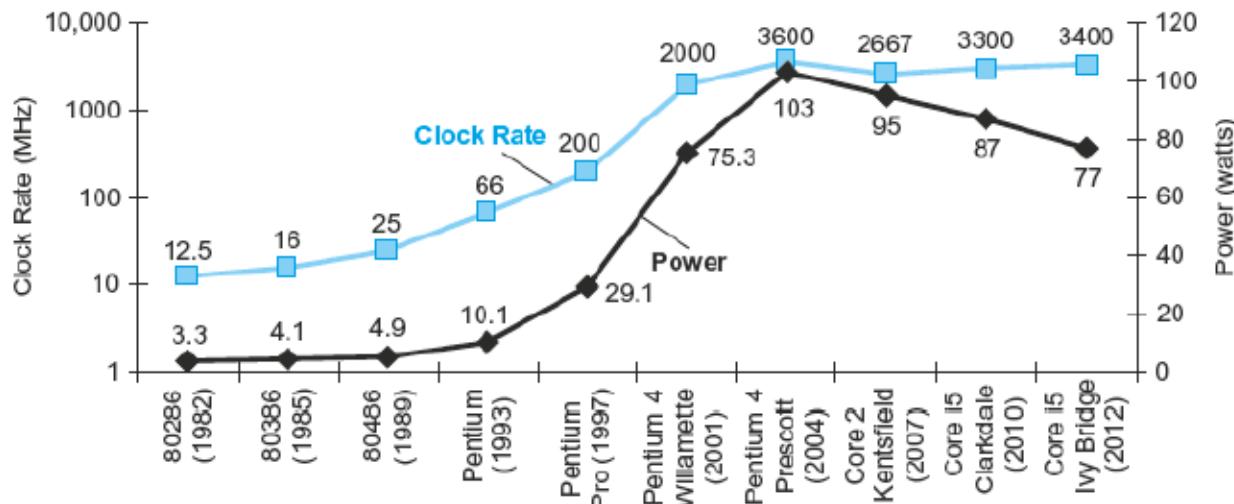
Power and energy in integrated circuits - 1

Power is the biggest challenge today facing the computer designer. First, power must be brought in and distributed around the chip and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

Both clock rate and power increased rapidly for decades and then flattened off recently. The reason is that designers have run into the practical power limit for cooling microprocessors.

Clock rate and power for Intel x86 microprocessors over eight generations

Source: Computer Organization and Design: The Hardware / Software Interface



Power and energy in integrated circuits - 2

For CMOS chips, the traditional energy consumption has been in switching transistors, usually called *dynamic energy*. The energy required per transistor is proportional to the product of the capacitive load driven by the transistor and the square of the voltage

$$\text{energy}_{\text{dynamic}} \hat{=} \text{capacitive load} \cdot \text{voltage}^2 .$$

The capacitive load is a function of the number of transistors connected to an output and of the technology, which by itself determines the capacitance of the wires and the transistors.

On the other hand, the *dynamic power* required per transistor is the product of the energy of a transition multiplied by the transitions frequency

$$\text{power}_{\text{dynamic}} \hat{=} \text{capacitive load} \cdot \text{voltage}^2 \cdot \text{switching frequency} .$$

Power and energy in integrated circuits - 3

Dynamic power and energy are greatly reduced by lowering the voltage, so voltages have dropped from 5V to just under 1V in 20 years to allow for frequency increase without affecting too much the power. The problem today is that further lowering of the voltage appears to make the transistors too *leaky*.

Besides the dynamic energy consumption, one has to consider nowadays also the *static energy* consumption because of the leakage current that flows even when the transistor is turned off. In servers, for instance, the leakage current is typically responsible for 40% of the total energy consumption. Thus, increasing the number of transistors increases the need for more power dissipation even when all transistors are turned off. Server chips can consume more than 100W and cooling the chip and the surrounding system is a major expense in warehouse scale computers.

Power and energy in integrated circuits - 4

Modern microprocessors offer several techniques to improve energy efficiency despite flat clock rates and constant supply voltages

- *keep track on operations* – most microprocessors today turn off the clock of inactive modules to save energy and dynamic power
- *dynamic voltage-frequency scaling* (DVFS) – most microprocessors today offer a few clock frequencies and voltages in which to operate for lower power and energy consumption
- *design for typical case* – microprocessors to be used in desktop computing are designed for a typical case of heavy use at high operating temperatures, relying on on-chip temperature sensors to detect when activity should be reduced automatically to avoid overheating
- *over clocking* – Intel started offering *Turbo mode* in 2008, where the chip decides by itself if it is safe to run at a higher clock rate, typically 10% over the nominal clock rate, for a short period of time, possibly on just a few cores, until the temperature starts to rise.

Dependability - I

Historically, integrated circuits were one of the most reliable components of a computer system. That conventional wisdom, however, has started to change as transistor feature sizes became smaller than 32 nm. Both transient and permanent faults are becoming more common, so computer architects must design systems to cope with these challenges.

Module reliability is a measure of continuous module operation as specified, or to put it in another way, is a measure of the time that takes a module to fail counted from a reference initial instant. Hence, the *mean time to failure* (MTTF) is a reliability measure. The reciprocal of MTTF is the rate of failures, generally reported as failures per billion hours of operation, or *failures in time* (FIT). *Service interruption* is measured by the *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is just the sum of MTTF and MTTR.

If a collection of modules has exponentially distributed lifetimes – meaning that the age of a module is not relevant for its probability to fail, and their failures are independent, then the overall failure rate of the collection is the sum of the failure rates of the individual modules.

Dependability - 2

Module availability is a measure of continuous module operation as specified with respect to the alternation between service and interruption.

For nonredundant systems with repair, *module availability* is given by

$$\text{module availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} .$$

Dependability - 3

Assume a disk subsystem with the following components and MTTFs

- 10 disks, each rated at 1 000 000 hour MTTF
- 1 ATA controller rated at 500 000 hour MTTF
- 1 power supply rated at 200 000 hour MTTF
- 1 fan rated at 200 000 hour MTTF
- 1 ATA cable rated at 1 000 000 hour MTTF .

Using the simplifying assumptions about the probability distributions of the lifetimes and the independency of failures, compute the MTTF of the entire subsystem.

$$\begin{aligned}\text{failure rate}_{\text{subsys}} &= 10 \cdot \frac{1}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000} = \\ &= \frac{23}{1000000} \text{ hours}.\end{aligned}$$

$$\text{MTTF}_{\text{subsys}} = \frac{1}{\text{failure rate}_{\text{subsys}}} = \frac{1000000}{23} \approx 43\,500 \text{ hours} .$$

Dependability - 4

Disk subsystems often have redundant power supplies to improve dependability. Assume the power supply from last example, with a MTTR of 24 hours, and compute the reliability of a redundant system with two power supplies.

$$\begin{aligned} \text{MTTF}_{\text{power supply pair}} &= \frac{\frac{\text{MTTF}_{\text{power supply}}}{2}}{\frac{\text{MTTR}_{\text{power supply}}}{\text{MTTF}_{\text{power supply}}}} = \frac{\text{MTTF}_{\text{power supply}}^2}{2 \cdot \text{MTTR}_{\text{power supply}}} = \\ &= \frac{200000^2}{2 \cdot 24} \approx 830\,000\,000 \text{ hours} . \end{aligned}$$

Dependability - 5

Amdahl's Law is also applicable beyond performance.

Consider the reliability example discussed before and assume the power supply reliability was improved from the previous 200 000 hours MTTF to the impressive 830 000 000 hours MTTF, that is, a 4 150 times improvement. How does this affect the reliability of the overall system?

$$\begin{aligned}\text{failure rate}_{\text{orig sys}} &= 10 \cdot \frac{1}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000} = \\ &= \frac{23}{1000000 \text{ hours}}.\end{aligned}$$

Therefore, the fraction of failure rate due to the power supply 5 per 1 000 000 hours out of 23 per 1 000 000 hours, or approximately 22%. Hence, one has

$$\text{reliability improv}_{\text{power supply}} = \frac{1}{0.78 + \frac{0.22}{4150}} = 1.28 .$$

Measuring performance - 1

In comparing design alternatives, one often wants to relate the *performance* of two different computer systems, say X and Y. The *sentence X is faster than Y* means that the *execution time* of some program run on X is shorter than the execution time of the same program run on Y.

But the problem is how to choose a program, or a group of programs, commonly called a *benchmark*, or a *benchmark suite*, that will produce significant results in general and that do not rely on specific features of any of the alternatives.

Furthermore, one has to consider that the applications in the real world run in computer systems are so different in size and complexity that the choice of a computer system for a specific area is not an easy task.

Measuring performance - 2

One way to proceed is to run programs that are simpler than real applications, but that at the same time may be considered representative

- *kernels* – small, key portions of real applications
- *toy programs* – small programs, usually with a length up to 100 code lines, of the type students are asked to write in programming classes
- *synthetic code* – fake programs which were written with the intent to try and match the behavior of real applications.

This approach is not popular anymore, mostly because it is possible to write compilers that take the knowledge of specific programs into account so that a specific computer system appear to run these programs faster than when it is actually running real applications.

The favored approach nowadays is to consider sets of programs whose representativeness is generally accepted by a wide audience and use them to characterize the relative performance of two computer systems, one of them being the reference computer.

Measuring performance - 3

The *Standard Performance Evaluation Corporation* (SPEC) is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers. SPEC develops benchmark suites and also reviews and publishes submitted results from member organizations and other benchmark licensees.

SPECfp2000 (Sun Ultra 5 as the reference computer)

Source: Computer Architecture: A Quantitative Approach

Benchmark	Ultra 5 exec time (s)	Opteron exec time (s)	SPECRatio	Itanium 2 exec time (s)	SPECRatio	Opteron/Itanium 22 exec time	Itanium 2 / Opteron SpecRatio
wupwise	1 600	51.5	31.06	56.1	28.53	0.92	0.92
swim	3 100	125.0	24.73	70.7	43.85	1.77	1.77
mgrid	1 800	98.0	18.37	65.8	27.36	1.49	1.49
applu	2 100	94.0	22.34	50.9	41.25	1.85	1.85
mesa	1 400	64.6	21.69	108.0	12.99	0.60	0.60
galgel	2 900	86.4	33.57	40.0	72.47	2.16	2.16
art	2 600	92.4	28.13	21.0	123.67	4.40	4.40
quake	1 300	72.6	17.92	36.3	35.78	2.00	2.00
facerec	1 900	73.6	25.80	86.9	21.86	0.85	0.85
ammp	2 200	136.0	16.14	132.0	16.63	1.03	1.03
lucas	2 000	88.8	22.52	107.0	18.76	0.83	0.83
fma3d	2 100	120.0	17.48	131.0	16.09	0.92	0.92
sixtrack	1 100	123.0	8.95	68.8	15.99	1.79	1.79
apsi	2 600	150.0	17.36	231.0	11.27	0.65	0.65
Geometric mean			20.86		27.12	1.30	1.30

Measuring performance - 4

$$\frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} = \frac{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } A_i}}{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } B_i}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{SPECRatio } A_i}{\text{SPECRatio } B_i}} =$$
$$= \sqrt[n]{\prod_{i=1}^n \frac{\frac{\text{exec time Ref}_i}{\text{exec time Ref}_i}}{\frac{\text{exec time } A_i}{\text{exec time Ref}_i}}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{exec time } B_i}{\text{exec time } A_i}} =$$
$$= \sqrt[n]{\prod_{i=1}^n \frac{\text{Performance } A_i}{\text{Performance } B_i}}$$

The processor performance equation - 1

CPU execution time for running a program can be expressed by

$$\text{CPU execution time} = \text{CPU clock cycles} \cdot \text{clock cycle time ,}$$

where the variable *CPU clock cycles* represent the total number of clock cycles for the execution of the program and the variable *clock cycle time* is period of the CPU clock.

This expression can be further expanded into

$$\text{CPU execution time} = \text{instruction count} \cdot \text{CPI} \cdot \text{clock cycle time ,}$$

where the variable *instruction count* represent the total number of instructions executed by the program and it can be obtained in a straightforward manner for non-sophisticated processor organizations if one has a listing of the program compilation into assembly language, and the variable *CPI* is simply the average number of clock cycles per instruction.

The processor performance equation - 2

CPU execution time for running a program is, therefore, made dependent of three features: instruction count for the execution of the program, average number of clock cycles per instruction and clock cycle time. Furthermore, *CPU execution time* is *equally* dependent on these three parameters: a given percent change in one of them will result in the same change for *CPU execution time*.

Unfortunately, it is difficult to change one parameter in complete isolation of the others as the basic technologies involved in the changing are interdependent

- *instruction count* – is dependent on the computer architecture and the compiler technology
- *clock cycles per instruction* – is dependent on both the computer architecture and the computer organization
- *clock cycle time* – is dependent on the underlying hardware technology and on the computer organization.

The processor performance equation - 3

CPU execution time for running a program can still be further refined if one takes into account the *CPI* for each type of instruction that is being executed

$$\text{CPU execution time} = \text{instruction count} \cdot \sum_i \left(\frac{\text{instruction count}_i}{\text{instruction count}} \cdot \text{CPI}_i \right) \cdot \text{clock cycle time} .$$

The processor performance equation - 4

Suppose the following measurements were made on running a benchmark in a given computer system

- frequency of FP operations = 25%
- average CPI for FP operations = 4.00
- average CPI for all other operations = 1.33
- frequency of FPSQR = 2%
- CPI of FPSQR = 20 .

Assume that design alternatives are being considered either to decrease the CPI for FPSQR to 2, or to decrease the average CPI for all FP operations to 2.5. Use the processor performance equation to compare these alternatives.

Since CPI is the only parameter that changes in the processor performance equation, the answer may be given by just computing the overall CPI in the original case and for the two alternatives.

The processor performance equation - 5

Original situation

$$\begin{aligned}\text{overall CPI}_{\text{orig}} &= \sum_i \frac{\text{instruction count}_i}{\text{instruction count}} \cdot \text{CPI}_i = \\ &= 4.00 \cdot 0.25 + 1.33 \cdot 0.75 = 2.00\end{aligned}$$

Enhanced FPSQR

$$\begin{aligned}\text{overall CPI}_{\text{enFPSQR}} &= \text{overall CPI}_{\text{orig}} - 0.02 \cdot (\text{CPI}_{\text{origFPSQR}} - \text{CPI}_{\text{enFPSQR}}) = \\ &= 2.00 - 0.02 \cdot (20 - 2) = 1.64\end{aligned}$$

Enhanced FP

$$\begin{aligned}\text{overall CPI}_{\text{enFP}} &= \sum_i \frac{\text{instruction count}_i}{\text{instruction count}} \cdot \text{CPI}_i = \\ &= 2.50 \cdot 0.25 + 1.33 \cdot 0.75 = 1.63\end{aligned}$$

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
 - Chapter 1: *Fundamentals of Quantitative Design and Analysis*
 - Appendix M: *Historical Perspectives and References*
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
 - Chapter 1: *Basic Concepts and Computer Evolution*
 - Chapter 2: *Performance Issues*

Dependability

1. Exponential distribution

The exponential probability distribution is used to model the behavior of a random variable T having the following properties

- T , with $T \geq 0$, represents the time interval between two events of some kind or between the first event and the origin of the reference frame
- the occurrence of one event does not affect the probability of occurrence of a second one
- the rate of events occurrence is constant
- the occurrence of successive events can not coincide in time.

Under these conditions, the probability density function $p(t)$ is expressed by

$$p(t) = \lambda e^{-\lambda t}, \text{ for } t \in [0, +\infty[,$$

where λ , called the *event rate*, is the number of events per unit of time.

The probability that an event occurs in the time interval T less than t , denoted by $P(T < t)$ is determined by

$$P(T < t) = \int_0^t p(u) du = \int_0^t \lambda e^{-\lambda u} du = 1 - e^{-\lambda t}.$$

It should be noted that

- $P(T \geq t) = 1 - P(T < t) = e^{-\lambda t}$ represents the probability that no event occurs in the time interval T less than t
- the exponential distribution is indeed *memoryless*, that is, the conditional probability that some event occurs in the time interval $[t_0, t_0 + \Delta t[$, provided no event has occurred in the time interval $[0, t_0[$, is equal to the probability that some event occurs in the time interval $[0, \Delta t[$

$$\begin{aligned} P(T < t_0 + \Delta t \mid T \geq t_0) &= 1 - P(T \geq t_0 + \Delta t \mid T \geq t_0) = \\ &= 1 - \frac{e^{-\lambda(t_0 + \Delta t)}}{e^{-\lambda t_0}} = 1 - e^{-\lambda \Delta t} = P(T < \Delta t). \end{aligned}$$

The expected value, or mean, $E(\mathbf{T})$ and variance $Var(\mathbf{T})$ of a random variable \mathbf{T} that satisfies an exponential distribution with event rate λ , are given by

$$E(\mathbf{T}) = \int_0^{+\infty} \lambda t e^{-\lambda t} dt = \frac{1}{\lambda}$$

$$Var(\mathbf{T}) = \int_0^{+\infty} \lambda [t - E(\mathbf{T})]^2 e^{-\lambda t} dt = \int_0^{+\infty} \lambda t^2 e^{-\lambda t} dt - \frac{1}{\lambda^2} = \frac{1}{\lambda^2} .$$

2. Reliability

In reliability theory, the exponential distribution is used to model the failures on the modules of a physical system. In this framework, the random variable \mathbf{T} represents the time interval to module failures and the event rate λ expresses the module *failure rate* or, when converted to failures per billion hours of module operation, the module *failures in time* (FIR). Its reciprocal, equal to the mean of the random variable \mathbf{T} , is called *mean time to failure* (MTTF).

In a system consisting of N distinct modules whose individual failure behavior can be modeled by the exponential distribution and if one assumes that module failures are independent of one another, the probability that no failures occur in the time interval $[0, t_0]$, or that the system operates as specified, is expressed by

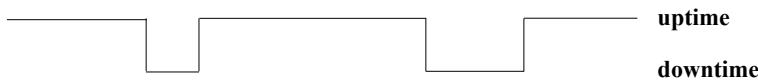
$$P(\mathbf{T}_1 \geq t_0, \mathbf{T}_2 \geq t_0, \dots, \mathbf{T}_N \geq t_0) = \prod_{i=1}^N P(\mathbf{T}_i \geq t_0) =$$

$$= \prod_{i=1}^N e^{-\lambda_i t_0} = e^{-(\lambda_1 + \lambda_2 + \dots + \lambda_N) t_0},$$

which means that under this conditions the *system*, or *overall, failure rate* is the sum of the failure rates of the constituent modules or, putting it in terms of the mean time to failure of the modules, that the *system*, or *overall, mean time to failure* is given by

$$\text{MTTF}_{\text{sys}} = \frac{1}{\sum_{i=1}^N \frac{1}{\text{MTTF}_i}} .$$

When a physical system fails to operate according to the specifications. The failed module has to be located and replaced by a properly functioning one. The time evolution of the system operation is thus described by a succession of alternate time intervals: *uptimes*, where there is proper operation, and *downtimes*, where the system is inactive or malfunctioning.



System time evolution

The system time evolution can now be modeled as a random variable \mathbf{T} equal to the summation of two independent exponentially distributed random variables: \mathbf{T}_{sys} that represents the time interval to system failures and \mathbf{T}_{rep} that represents the time interval to system repair.

Let $p_{\text{sys}}(t)$ and $p_{\text{rep}}(t)$ denote the probability density functions of the random variables \mathbf{T}_{sys} and \mathbf{T}_{rep} , respectively, then the probability density distribution $p(t)$ of the random variable \mathbf{T} , their sum, is determined by

$$\begin{aligned} p(t) &= \int_0^t p_{\text{sys}}(u) p_{\text{rep}}(t-u) du = \int_0^t \lambda_{\text{sys}} e^{-\lambda_{\text{sys}} u} \lambda_{\text{rep}} e^{-\lambda_{\text{rep}}(t-u)} du = \\ &= \lambda^2 t e^{-\lambda t} \quad \Leftarrow \quad \lambda = \lambda_{\text{sys}} = \lambda_{\text{res}} \\ &= \frac{\lambda_{\text{sys}} \lambda_{\text{res}}}{\lambda_{\text{sys}} - \lambda_{\text{res}}} \cdot (e^{-\lambda_{\text{rep}} t} - e^{-\lambda_{\text{sys}} t}) \quad \Leftarrow \quad \lambda_{\text{sys}} \neq \lambda_{\text{res}} , \end{aligned}$$

and its expected value $E(\mathbf{T})$ by

$$\begin{aligned} E(\mathbf{T}) &= \int_0^{+\infty} \lambda^2 t^2 e^{-\lambda t} dt = \frac{2}{\lambda} \quad \Leftarrow \quad \lambda = \lambda_{\text{sys}} = \lambda_{\text{res}} \\ &= \int_0^{+\infty} \frac{\lambda_{\text{sys}} \lambda_{\text{res}}}{\lambda_{\text{sys}} - \lambda_{\text{res}}} \cdot t (e^{-\lambda_{\text{rep}} t} - e^{-\lambda_{\text{sys}} t}) dt = \frac{1}{\lambda_{\text{sys}}} + \frac{1}{\lambda_{\text{res}}} \quad \Leftarrow \quad \lambda_{\text{sys}} \neq \lambda_{\text{res}} . \end{aligned}$$

Calling the mean of the random variable \mathbf{T}_{rep} *mean time to repair* (MTTR) and the mean of the random variable \mathbf{T} *mean time between failures* (MTBF), one gets the following relation that can be applied both to the whole system or to a single module

$$\begin{aligned} \text{MTBF}_{\text{sys}} &= \text{MTTF}_{\text{sys}} + \text{MTTR}_{\text{sys}} \\ \text{MTBF}_{\text{mod}} &= \text{MTTF}_{\text{mod}} + \text{MTTR}_{\text{mod}} . \end{aligned}$$

As a measure of how reliable a system, or a module, is, one defines its *availability*, a figure of merit that describes the proportion of its uptime to its downtime during normal operation.

For nonredundant systems with repair, *availability* is expressed by the ratio

$$\begin{aligned} \text{availability}_{\text{sys}} &= \frac{\text{MTTF}_{\text{sys}}}{\text{MTBF}_{\text{sys}}} = \frac{\text{MTTF}_{\text{sys}}}{\text{MTTF}_{\text{sys}} + \text{MTTR}_{\text{sys}}} \\ \text{availability}_{\text{mod}} &= \frac{\text{MTTF}_{\text{mod}}}{\text{MTBF}_{\text{mod}}} = \frac{\text{MTTF}_{\text{mod}}}{\text{MTTF}_{\text{mod}} + \text{MTTR}_{\text{mod}}} . \end{aligned}$$

Resource redundancy is the most popular method to deal with availability improvements. The rationale is to include in the design of the system extra modules that are prepared to replace modules proved to be defective on-line. In this way, the MTTF of the concerned modules is extended and so is the overall MTTF.

Suppose that in the design of a system a given module is replicated so that its availability is increased. What is the improvement gain that was obtained?

The reasoning rests in the following premises

- in order for the module to fail, both replicas must have a failure
- the module availability, when two modules are operative, is given by

$$\text{availability}_{2\text{ mod}} = \frac{\frac{\text{MTTF}_{\text{mod}}}{2}}{\frac{\text{MTTF}_{\text{mod}}}{2} + \text{MTTR}_{\text{mod}}}$$

- the module availability, when only one module is operative, is given by

$$\text{availability}_{1\text{ mod}} = \frac{\text{MTTF}_{\text{mod}}}{\text{MTTF}_{\text{mod}} + \text{MTTR}_{\text{mod}}}$$

- the module availability of the redundant configuration is given by

$$\text{availability}_{\text{redund 2}} = 1 - (1 - \text{availability}_{2\text{ mod}}) \cdot (1 - \text{availability}_{1\text{ mod}}) .$$

This result can be expressed in terms of the MTTF and MTTR of single modules if the substitutions are made to the expression above, yielding

$$\begin{aligned} \text{availability}_{\text{redund 2}} &= 1 - \left(1 - \frac{\frac{\text{MTTF}_{\text{mod}}}{2}}{\frac{\text{MTTF}_{\text{mod}}}{2} + \text{MTTR}_{\text{mod}}} \right) \cdot \left(1 - \frac{\text{MTTF}_{\text{mod}}}{\text{MTTF}_{\text{mod}} + \text{MTTR}_{\text{mod}}} \right) = \\ &= \frac{\frac{\text{MTTF}_{\text{mod}}^2}{2} + \frac{3 \text{MTTF}_{\text{mod}} \cdot \text{MTTR}_{\text{mod}}}{2}}{\frac{\text{MTTF}_{\text{mod}}^2}{2} + \frac{3 \text{MTTF}_{\text{mod}} \cdot \text{MTTR}_{\text{mod}}}{2} + \text{MTTR}_{\text{mod}}^2} = \\ &= \frac{\frac{\text{MTTF}_{\text{mod}}^2}{2 \text{MTTR}_{\text{mod}}} + \frac{3}{2} \cdot \text{MTTF}_{\text{mod}}}{\frac{\text{MTTF}_{\text{mod}}^2}{2 \text{MTTR}_{\text{mod}}} + \frac{3}{2} \cdot \text{MTTF}_{\text{mod}} + \text{MTTR}_{\text{mod}}} , \end{aligned}$$

which means that

$$\begin{aligned} \text{MTTF}_{\text{mod - redund 2}} &= \text{MTTF}_{\text{mod}} \cdot \frac{\text{MTTF}_{\text{mod}} + 3 \text{MTTR}_{\text{mod}}}{2 \text{MTTR}_{\text{mod}}} \approx \\ &\approx \frac{\frac{\text{MTTF}_{\text{mod}}}{2}}{\frac{\text{MTTR}_{\text{mod}}}{\text{MTTF}_{\text{mod}}}} , \text{ if } \text{MTTF}_{\text{mod}} \gg 3 \text{MTTR}_{\text{mod}} . \end{aligned}$$

In the general case, for a redundancy degree of $K \in \mathbb{N}$, one gets

$$\text{availability}_{\text{redund } K} = 1 - \prod_{k=1}^K (1 - \text{availability}_{k\text{ mod}}) ,$$

or in terms of the MTTF and MTTR of single modules

$$\begin{aligned}
 \text{MTTF}_{\text{mod - redund K}} &= \text{MTTF}_{\text{mod}}^{K-1} \cdot \frac{\text{MTTF}_{\text{mod}} + O[K(K+1)/2 \cdot \text{MTTR}_{\text{mod}}]}{K! \cdot \text{MTTR}_{\text{mod}}^{K-1}} \approx \\
 &\approx \frac{\frac{\text{MTTF}_{\text{mod}}}{K!}}{\left(\frac{\text{MTTR}_{\text{mod}}}{\text{MTTF}_{\text{mod}}}\right)^{K-1}}, \text{ if } \text{MTTF}_{\text{mod}} \gg O[K(K+1)/2 \cdot \text{MTTR}_{\text{mod}}].
 \end{aligned}$$

For large values of K the above approximation is often not valid. A more accurate way to determine the module MTTF with K redundancy is through the K redundancy availability

$$\text{MTTF}_{\text{mod - redund K}} = \frac{\text{MTTR}_{\text{mod}} \cdot \text{availability}_{\text{redund K}}}{(1 - \text{availability}_{\text{redund K}})}.$$

Finally, one needs to know what was the reliability improvement gained by a system where the availability of some constituent module was increased in some way. The improvement gain can be computed through the application of Amdahl's Law

$$\text{reliability improvement} = \frac{1}{\text{frac sys non improv failure rate} + \frac{\text{frac sys improv failure rate}}{\left(\frac{\text{MTTF}_{\text{mod - improv}}}{\text{MTTF}_{\text{mod - non improv}}}\right)}}.$$

universidade de aveiro



Arquitecturas de Alto Desempenho

Instruction-Level Parallelism (Principles)

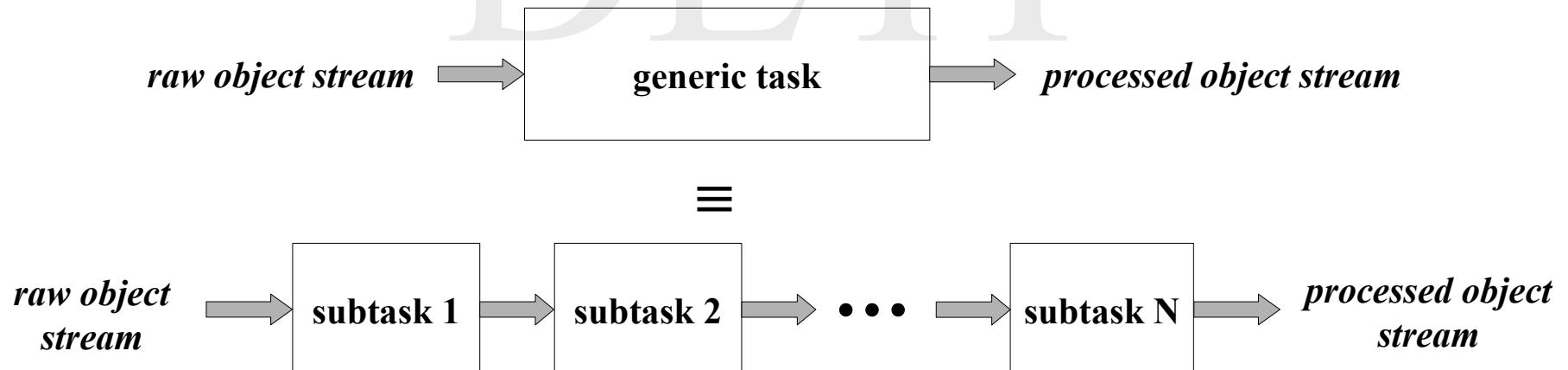
António Rui Borges

Summary

- *What is pipelining?*
- *Instruction-level parallelism*
- *Simplified RISC instruction set*
- *Non-pipelined implementation of RISC instruction set*
- *Classical 5-stage pipeline for a RISC processor*
- *Major hurdles of pipelining*
 - *Performance of pipelines with stalls*
 - *Structural hazards*
 - *Data hazards*
 - *Control hazards*
- *Implementation of classical 5-stage pipeline*
- *Exceptions*
- *Multicycle operations in classical 5-stage pipeline*
- *MIPS R4000 pipeline*
- *Suggested reading*

What is pipelining? - I

Pipelining is an implementation technique where the execution of a generic task on the objects of a stream is converted into a tandem of independent subtasks which operate simultaneously on successive objects of the stream. Each of the individual subtasks, called *pipe stages* or *segments*, is performed in sequence and represents a definite fraction of the whole task. Their combined ordered execution is equivalent to the execution of the original task on every object of the stream.



What is pipelining? - 2

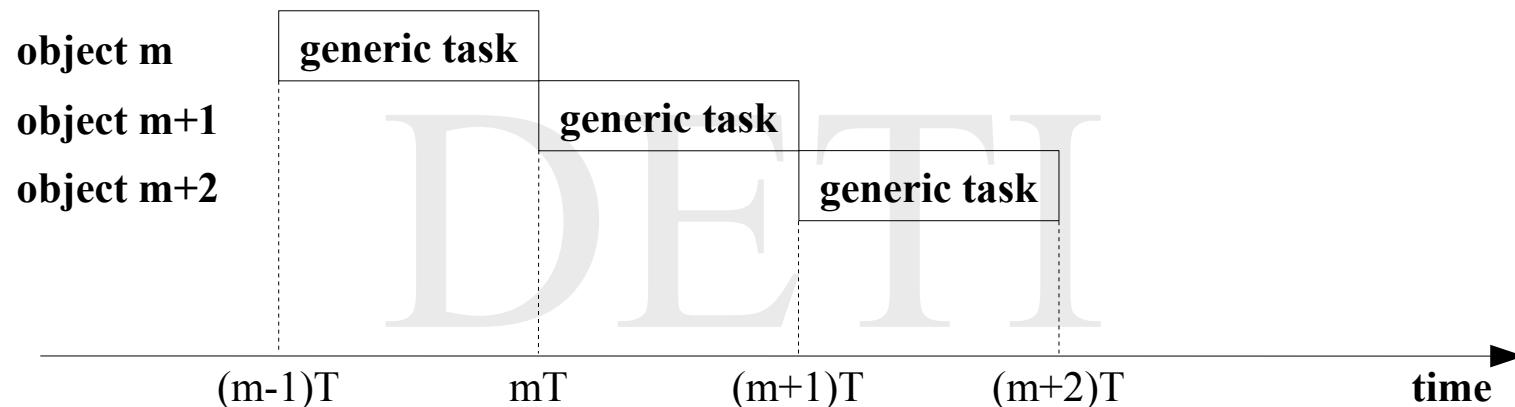
The concept of pipelining was invented in the context of the automobile industry by Ford Motor Company in the beginning of the 1900s, inspired by the operating procedures of the Chicago cattle slaughterhouses. The first factory which introduced a fully developed moving assembly line, started operations by the end of 1913 in Detroit for the production of Ford Model T.

The goal of a moving assembly line was to speed up the process of building a car from its parts and, consequently, lower its market price and make it available to a much wider consumer audience. In fact, the time to assemble a car passed from about 12h and 30m, prior to the introduction of the moving assembly line, to just 1h and 33m afterwards.

The impact of moving assembly lines was so great for the automobile industry as whole that Ford Motor Company was producing 50% of all cars sold in the US and about 40% of the cars sold in the British Commonwealth by 1920.

What is pipelining? - 3

non-pipelined version

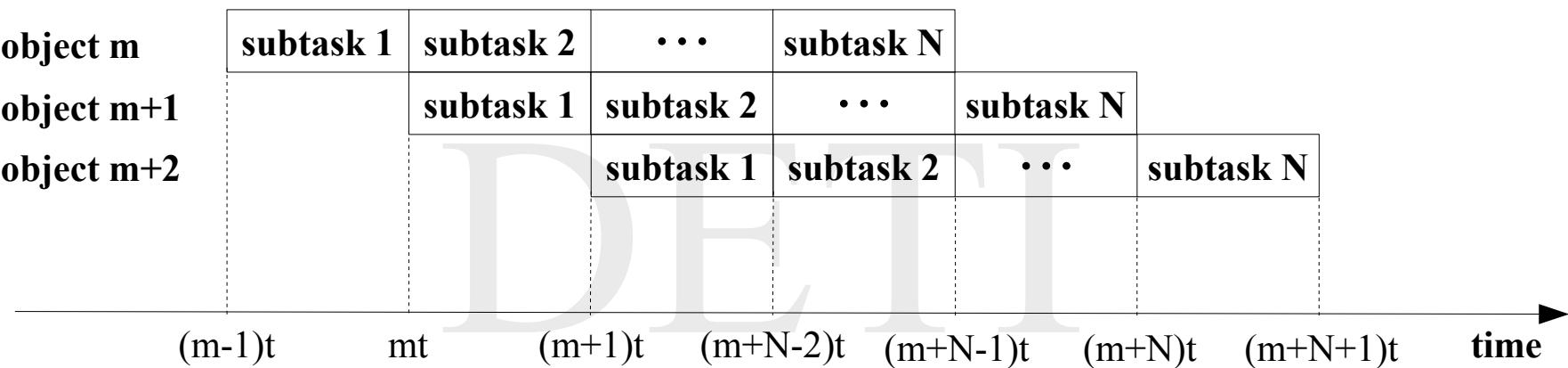


$$\text{throughput} = \frac{1}{T}$$

$$\text{execution time for a } M \text{ object stream} = M \cdot T$$

What is pipelining? - 4

N-stage pipelined version



t_n , with $n = 1, 2, \dots, N$ — execution time for subtask n

throughput = $\frac{1}{t}$, where $t = \max(t_1, t_2, \dots, t_N)$

execution time for a M object stream = $(N - 1 + M) \cdot t$

What is pipelining? - 5

The *speed up* obtained from the execution of a N -stage pipeline implementation over the non-pipelined execution of the same task is expressed by

$$\begin{aligned}\text{speed up}_{N\text{-stage pipeline}} &= \frac{\text{execution time of the non-pipelined implementation}}{\text{execution time of the } N\text{-stage pipelined implementation}} = \\ &= \frac{M \cdot T}{(N - 1 + M) \cdot t} = \frac{M \cdot N \cdot t^*}{(N - 1 + M) \cdot t} \approx N \cdot \frac{t^*}{t},\end{aligned}$$

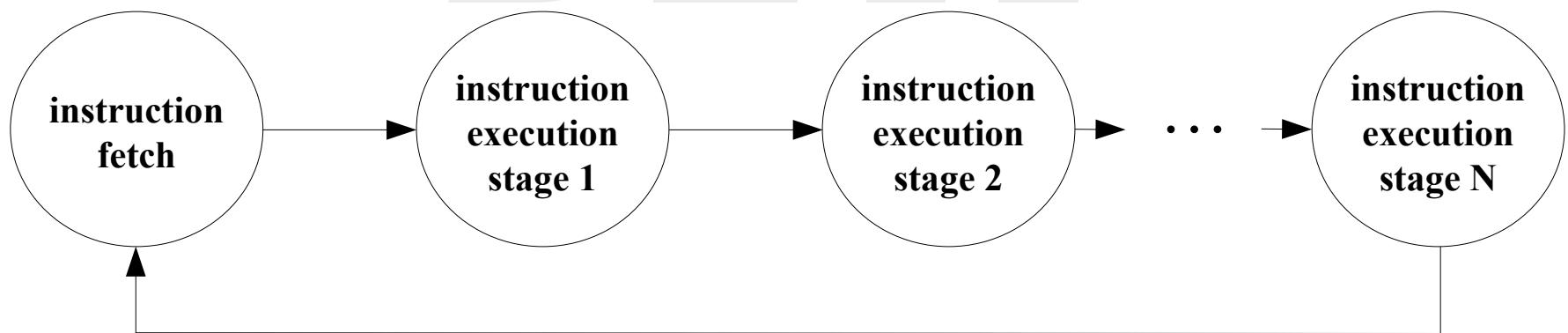
where $T = M \cdot t^*$ and $N \ll M$.

Ideally, if the pipe stages are almost perfectly balanced and the overhead involved in pipelining is negligible, then the ratio t^*/t approaches unity and the speed up is approximately equal to the number stages used on the partition of the original task into subtasks.

Instruction-level parallelism

All processors since 1985 have adopted pipelining as a means for overlapping the execution of instructions and for improving performance. This potential overlap of instructions during their own execution is called *instruction-level parallelism* (ILP) because the instructions are in some sense processed in parallel through the activity decomposition that takes place.

A common approach for doing this is by decomposing the *instruction execution* phase of the processor cycle into several stages.



Simplified RISC instruction set - 1

Part of the core architecture of a typical RISC (Reduced Instruction Set Computer), MIPS64, is going to be used to illustrate the basic concepts of pipelining.

RISC architectures are characterized by a few key properties, which simplify dramatically their implementation

- the only operations that affect memory are the *load* and *store* instructions that move data from memory to a register of the register bank, or from a register of the register bank to memory; instructions that transfer less than the contents of a full register are often also available
- the only operations on data are the *arithmetic* and *logic* instructions that apply to data on registers of the register bank; they change in principle the whole register
- the instruction formats are few in number, having typically the same size.

These properties also lead to dramatic simplifications in the implementation of pipelining, which is one of the reasons why the instruction sets are so designed.

Simplified RISC instruction set - 2

A brief description of the 64-bit version of MIPS instruction set follows. All instructions are 32 bits in length. The data types consist of 8-bit *bytes*, 16-bit *half words*, 32-bit *words* and 64-bit *double words*. The extended 64-bit instructions are generally denoted by having a D at the start, or at the end, of the mnemonic. The general-purpose register bank contains 32 64-bit registers, of which *register 0* is read-only and has value zero.

The core instruction set has three classes of instructions

- *load and store instructions* – these instructions take as operands a register source, the *base register*, and an immediate 16-bit field, the *offset*; the sum of the contents of the base register and the sign-extended offset is used as a memory address, the *effective address*; the instructions LD and SD load or store the entire 64-bit register contents

opcode	rs	rt	offset	0
31	25	20	15	

I-type instruction

Simplified RISC instruction set - 3

- *arithmetic / logic instructions* – these instructions take either two registers or a register and a sign-extended immediate value, operate on them and store the result in another register; typical *arithmetic* instructions include add (DADD) and subtract (DSUB) for the 64-bit extensions; *logical* instructions, however, do not differentiate between 32-bit and 64-bit extensions; *immediate* versions use the same mnemonics with suffix I; there are signed and unsigned forms of the *arithmetic* instructions: the unsigned forms do not generate overflow exceptions and have a suffix U at the end (DADDU, DSUBU, DADDIU)

opcode	rs	rt	rd	shamt	funct
31	25	20	15	10	5

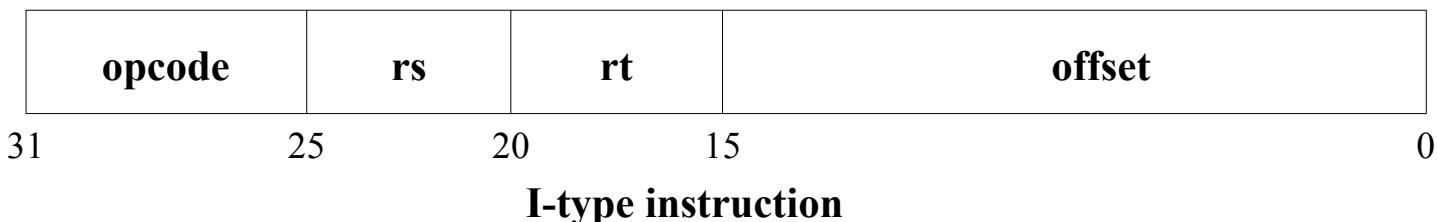
R-type instruction

opcode	rs	rt	immediate
31	25	20	15

I-type instruction

Simplified RISC instruction set - 4

- *branch instructions* – these instructions are conditional modifications of the strict sequentiality of instruction execution; the branch *condition* is typically specified either by a set of *condition bits*, also called *condition code*, a limited set of comparisons between a pair of registers (*greater*, *greater or equal*, *less*, *less or equal*, *equal* and *non-equal*) or between a register and zero (*equal* and *non-equal*); the branch *destination* is obtained by adding a sign-extended offset, shifted left by two bits to make it a word offset, to the current contents of the *program counter* (PC).



Non-pipelined implementation of a RISC instruction set - 1

A simple non-pipelined implementation of part of the MIPS64 core instruction set is presented and will be used as a framework to fully understand the inner workings of pipelining. It is not the most economical or the highest performant implementation, but it is designed to lead naturally to pipelining.

Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; they are introduced just to simplify pipelining. This implementation will focus only on the integer subset of MIPS64 core instruction set that consists of the load / store, integer ALU and branch operations. They will be implemented in at most 5 clock cycles.

The operation at each clock cycle is as follows

1. *Instruction fetch cycle (IF)*

The contents of the program counter (PC) is used as an address to fetch the next instruction from memory, its value is stored in the instruction register (IR). The PC is next updated to point the next instruction by adding 4 to the current contents.

Non-pipelined implementation of a RISC instruction set - 2

2. Instruction decode / register fetch cycle (ID)

The instruction is decoded leading to the operations

- the contents of the registers corresponding to the register source specifiers are read from the register bank
- the equality test is carried out on the contents of the registers as they are read, to be later used in a possible branch
- the offset field is sign-extended in case it will be needed
- the possible branch target address is computed by adding the incremented PC contents to the 2 bit left-shifted sign-extended offset field.

The *branch* instruction is completed at the end of this stage by storing the branch target address into the PC if the test condition yields true, or by leaving it unchanged if the test condition yields false.

Non-pipelined implementation of a RISC instruction set - 3

3. Execution / effective address cycle (EX)

The ALU takes the operands computed in the previous cycle and, depending on the instruction type, performs one of the actions below

- *memory reference* – the base register and the offset are added together to form the effective address
- *register-to-register ALU instruction* – the operation specified by the opcode is carried out on the values read from the register bank
- *register-immediate ALU instruction* – the operation specified by the opcode is carried out on the first value read from the register bank and the sign-extended immediate value.

Non-pipelined implementation of a RISC instruction set - 4

4. Memory access (MEM)

When the instruction is a *load*, the effective address computed in the previous cycle is the address of the memory location whose data is to be read; when the instruction is a *store*, the second value read from the register bank is to be written into the memory location whose address is the effective address.

The *store* instruction is completed at the end of this stage.

5. Write-back cycle (WB)

The result from an instruction of one of the types *register-to-register*, *register-immediate*, or *load*, is written into a register of the register bank.

Non-pipelined implementation of a RISC instruction set - 5

In this implementation, *branch* instructions are executed in 2 clock cycles, *store* instructions in 4 clock cycles and all other instructions in 5 clock cycles. Assuming that a given benchmark has a branch frequency of 12% and a store frequency of 10%, what is the average CPI for running this application on the non-pipelined implementation just described?

$$\begin{aligned}\text{overall CPI} &= \sum_i \frac{\text{instruction count}_i}{\text{instruction count}} \cdot \text{CPI}_i = \\ &= 0.12 \cdot 2 + 0.10 \cdot 4 + 0.78 \cdot 5 = 4.54 .\end{aligned}$$

Classical 5-stage pipeline for a RISC processor - 1

The non-pipelined implementation can be converted into a pipelined implementation with almost no structural changes by fetching a new instruction every clock cycle.

<i>Instruction number</i>	<i>Clock number</i>								
	1	2	3	4	5	6	7	8	9
m	IF	ID	EX	MEM	WB				
m+1		IF	ID	EX	MEM	WB			
m+2			IF	ID	EX	MEM	WB		
m+3				IF	ID	EX	MEM	WB	
m+4					IF	ID	EX	MEM	WB

Although each instruction takes 5 clock cycles to complete, during each cycle the hardware will be processing some part of five consecutive instructions.

Classical 5-stage pipeline for a RISC processor - 2

However, for this transformation to work, one has to determine what happens at every processor clock cycle and make sure that no two operations are using the same data path resource at the same time, that is, one needs to check carefully that the overlap of instructions in the pipeline is not causing such a conflict.

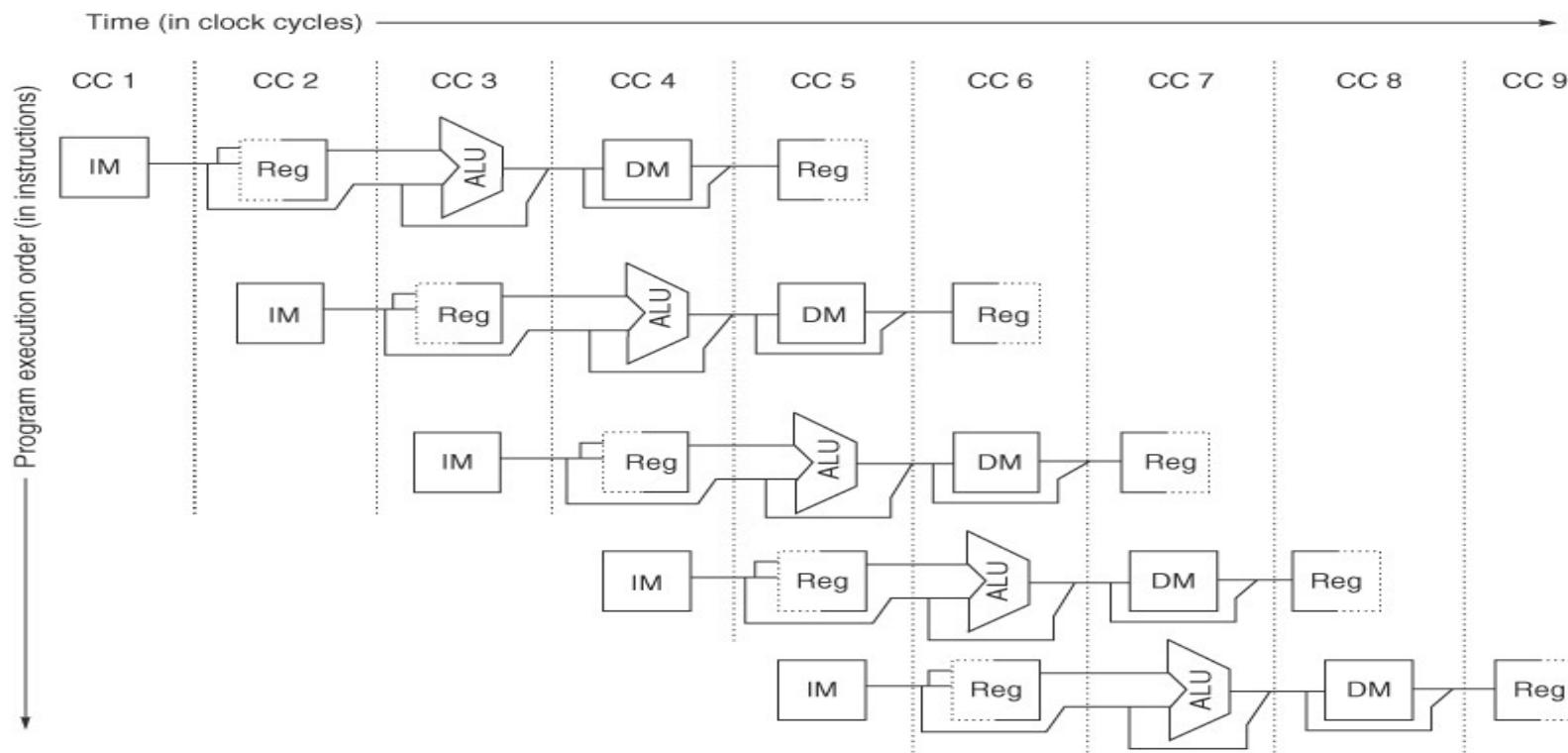
The number of details to consider are directly proportional to the number of pipeline stages and the number of instructions that are in overlapping execution. So it is important to devise a graphical representation of the execution that makes apparent what is happening.

A popular way of doing this is by describing the flow of operations as a series of the data path resources in use at the different stages of the pipeline.

Classical 5-stage pipeline for a RISC processor - 3

Pipeline seen as a series of data path resources shifted in time

Source: Computer Architecture: A Quantitative Approach



CC – clock cycle

DM – data memory

ALU – arithmetic/logic unit

IM – instruction memory

Reg – register bank

Classical 5-stage pipeline for a RISC processor - 4

A few observations are in order

- there is a need for separate *instruction* and *data memories*, which means the implementation of separate instruction and data caches for eliminating the conflict of a single memory that would arise between the *instruction fetch* (IF) and *data memory access* (MEM) stages
- the *register bank* is accessed in two stages: for reading, in the *instruction decode / register fetch* stage (ID), and for writing, in the *write-back* stage (WB); one needs to perform two reads and one write every clock cycle – to handle reading and writing at the same register, register write is performed in the first half of the clock cycle and register read in the second half
- to fetch an instruction every clock cycle, the *program counter* (PC) must be updated every clock cycle in the *instruction fetch* (IF) stage in preparation for the next instruction; furthermore, the potential branch target address must be computed during the *instruction decode / register fetch* stage (ID); still, since the branch, when the test condition yields true, also changes the PC in the ID stage, there is a conflict that will be dealt with later on.

Classical 5-stage pipeline for a RISC processor - 5

One must also ensure that instructions in different stages of the pipeline are not interfering with one another. The solution is to add registers for temporary storage of relevant data between successive stages, so that at the beginning of a clock cycle all the results from a given stage are stored into the registers that are used as inputs for the next stage. The registers are named for the two stages that are separated by that register: for instance, the registers between the IF and the ID stages are called IF/ID.

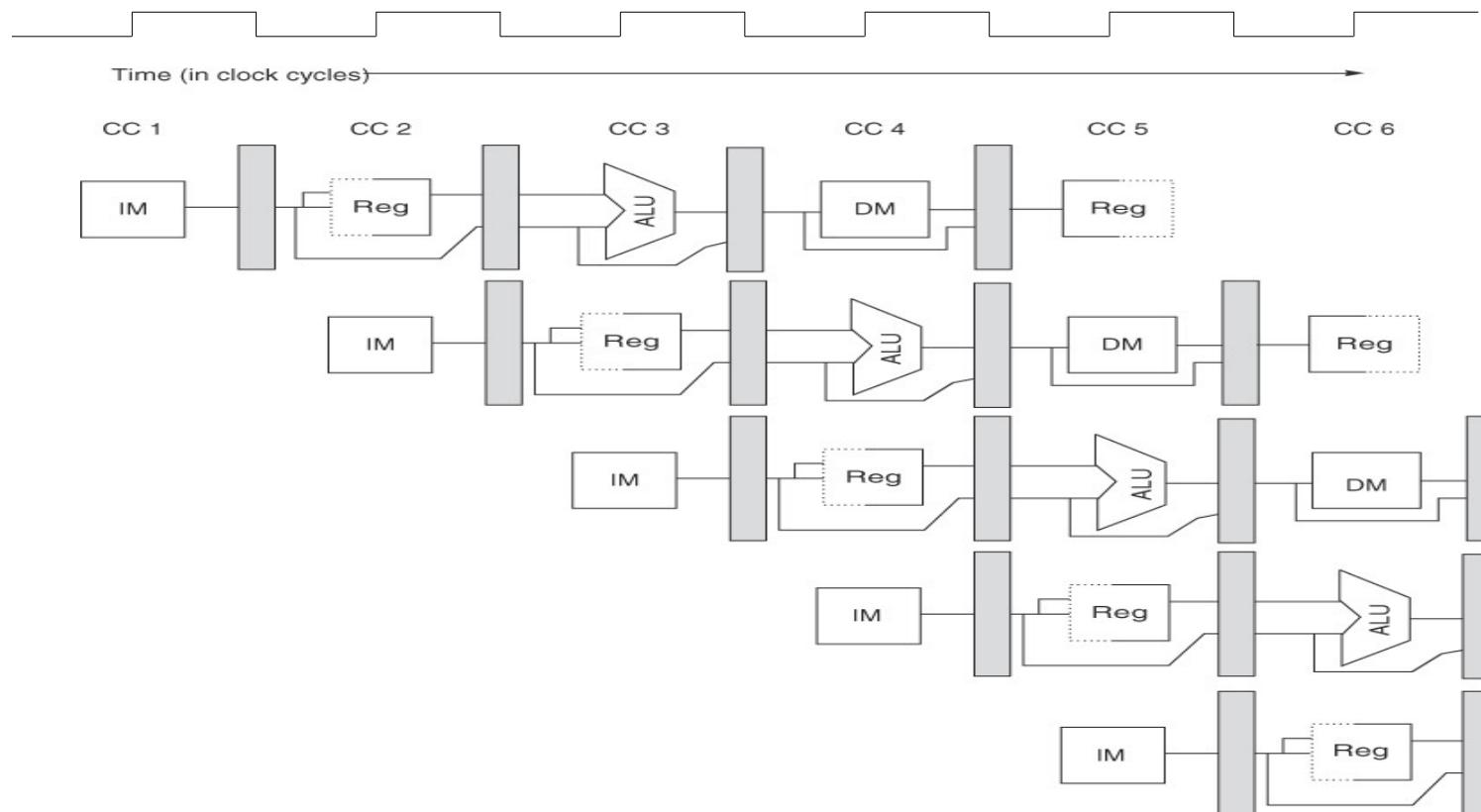
Notice that no registers are required at the end of the WB stage. All instructions must update in some particular manner the computer state (the register bank, the memory or the program counter) so a separate register at the end of the pipeline would be redundant to the state being updated.

The program counter itself may be thought of as a pipeline register: one feeding the IF stage of the pipeline. Unlike the other registers, however, the program counter is part of the visible architectural state; its contents must be saved when an exception is serviced, while the contents of the other pipeline registers is discarded.

Classical 5-stage pipeline for a RISC processor - 6

Pipeline with registers between successive stages for temporary storage

Source: Adapted from Computer Architecture: A Quantitative Approach



Classical 5-stage pipeline for a RISC processor - 7

Pipelining increases the instruction throughput, but does not reduce the execution time of an individual instruction. In fact, it slightly increases it due to the required overhead to control the pipeline. Notice that the increase in instruction throughput means that a program runs faster and has a lower execution time, even though no single instruction runs faster.

Consider the non-pipelined processor previously described. Assume it has a 1ns clock cycle and that it runs a benchmark with a branch frequency of 20% and a store frequency of 10%. Suppose that due to the overhead to control the pipeline, the clock cycle of the pipelined version of the processor is 1.2ns. What is the ideal speed up in the instruction execution rate gained from pipelining?

$$\begin{aligned}\text{average inst exec time}_{\text{nonpipelined}} &= \text{overall CPI}_{\text{nonpipelined}} \cdot \text{clock cycle time} = \\ &= (0.2 \cdot 2 + 0.1 \cdot 4 + 0.7 \cdot 5) \cdot 1.0 = 4.3 \text{ ns}\end{aligned}$$

$$\text{speed up} = \frac{\text{average inst exec time}_{\text{nonpipelined}}}{\text{average inst exec time}_{\text{pipelined}}} = \frac{4.3}{1.2} = 3.6 .$$

Major hurdles of pipelining - 1

Pipelining would work as described if the instructions were independent from one another. In reality, this is not so. There are situations, called *hazards*, which prevent the next instruction in the stream to be executed during its designated clock cycle. Thus, reducing the performance from the ideal speed up gained by pipelining.

There are three classes of hazards

- *structural hazards* – they arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
- *data hazards* – they arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of the instructions in the pipeline
- *control hazards* – they arise from the pipelining of branch instructions and other instructions that affect the PC.

Major hurdles of pipelining - 2

Hazards in pipelines may require to *stall* the pipeline, that is, in order to avoid the hazard it is often needed that some instructions in the pipeline are forced to remain in the same stage, while others are allowed to proceed to the next stage. When an instruction is stalled, all the instructions that were fetched later than the stalled instruction, are also stalled, and all the instructions fetched earlier must continue, since otherwise the hazard would never disappear. As a result, no new instructions are fetched during a stall and an execution delay takes place.

Performance of pipelines with stalls

$$\begin{aligned}\text{speed up} &= \frac{\text{average inst exec time}_{\text{nonpipelined}}}{\text{average inst exec time}_{\text{pipelined}}} = \\ &= \frac{\text{overall CPI}_{\text{nonpipelined}}}{\text{overall CPI}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} = \\ &= \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + \text{pipeline stall cycles per instruction}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}}.\end{aligned}$$

Structural hazards - 1

When a processor is pipelined, the overlapped execution of instructions requires both the same level of pipelining of all the functional units and the duplication of resources, to allow all possible combination of instructions in the pipeline. If some combination of instructions cannot be accommodated due to resource conflicts, the processor is said to incur in a *structural hazard*.

The most common instance of structural hazards arises when some functional unit is not fully pipelined. Then, a sequence of instructions where some of them use that unit, cannot proceed through the pipe stages at the rate of one instruction per clock cycle. Another common cause is when some resource is not replicated enough to allow all combinations of instructions to execute at the nominal rate.

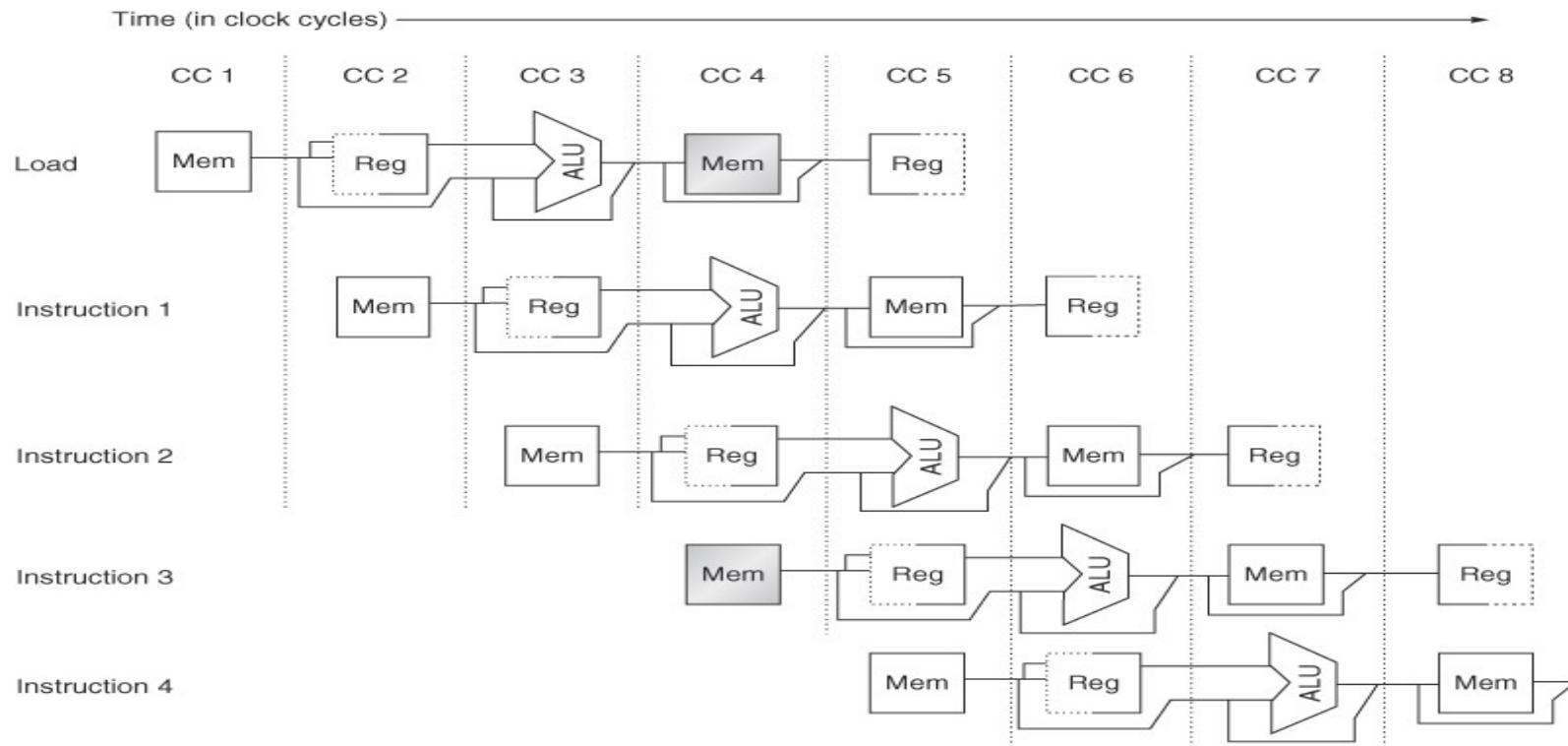
When this type of hazard is encountered, the pipeline will *stall* the latest coming instruction until the required unit is available. The stall will generate what is usually called a *bubble*, since this condition may be thought of flowing through the pipe stages taking space, but not producing any useful work.

Structural hazards - 2

Some processors have a single memory to access both instructions and data. This will generate a potential conflict between the instruction fetch (IF) and the data memory access (MEM) stages.

Processor with only one memory port

Source: Computer Architecture: A Quantitative Approach



Structural hazards - 3

Processor with only one memory port: the effect of a *load* instruction

Source: Adapted from Computer Architecture: A Quantitative Approach

<i>Instruction</i>	<i>Clock number</i>									
	1	2	3	4	5	6	7	8	9	10
<i>load</i>	IF	ID	EX	MEM	WB					
m+1		IF	ID	EX	MEM	WB				
m+2			IF	ID	EX	MEM	WB			
m+3				stall	IF	ID	EX	MEM	WB	
m+4						IF	ID	EX	MEM	WB
m+5							IF	ID	EX	MEM
m+6								IF	ID	EX

<i>Instruction</i>	<i>Clock number</i>									
	1	2	3	4	5	6	7	8	9	10
<i>load</i>	IF	ID	EX	MEM	WB					
m+1		IF	ID	EX	MEM	WB				
m+2			IF	ID	EX	MEM	WB			
<i>bubble</i>				IF	ID	EX	MEM	WB		
m+3					IF	ID	EX	MEM	WB	
m+4						IF	ID	EX	MEM	WB
m+5							IF	ID	EX	MEM
m+6								IF	ID	EX

Structural hazards - 4

Suppose that data reference instructions (*load* and *store*) constitute 40% of the instruction mix for a given benchmark and that the ideal CPI for the pipelined processor, ignoring the structural hazard, is 1. Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the processor without the hazard. Disregarding any other performance losses, which processor runs the benchmark faster?

$$\text{average inst exec time}_{\text{without haz}} = \text{overall CPI}_{\text{without haz}} \cdot \text{clock cycle time}_{\text{ideal}} = \\ = 1 \cdot \text{clock cycle time}_{\text{ideal}}$$

$$\text{average inst exec time}_{\text{with haz}} = \text{overall CPI}_{\text{with haz}} \cdot \text{clock cycle time}_{\text{ideal}} = \\ = (1 + 0.4 \cdot 1) \cdot \frac{\text{clock cycle time}_{\text{ideal}}}{1.05} \approx \\ \approx 1.3 \cdot \text{clock cycle time}_{\text{ideal}} .$$

Structural hazards - 5

If all factors are equal, a processor without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards?

The primary reason is to reduce the processor cost, since pipelining all functional units, and / or replicating them, may prove to be too costly. For example, processors that require an instruction and a data cache access every clock cycle need twice as much total memory bandwidth. Likewise, designing a fully-pipelined floating point multiplier consumes lots of gates and space may not be available.

If the structural hazard is rare, it may not be worth the cost to avoid it.

Data hazards - 1

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. Due to this, data hazards may arise. *Data hazards*, in fact, occur when the pipeline forces an actual order for read / write operations to operands different from the one perceived by sequentially executing the instructions on a non-pipelined processor.

Consider the pipelined execution of the code

DADD	R1, R2, R3
DSUB	R4, R1, R5
AND	R6, R1, R7
OR	R8, R1, R9
XOR	R10, R1, R11

Data hazards - 2

All the instructions after DADD use the result of the DADD instruction.

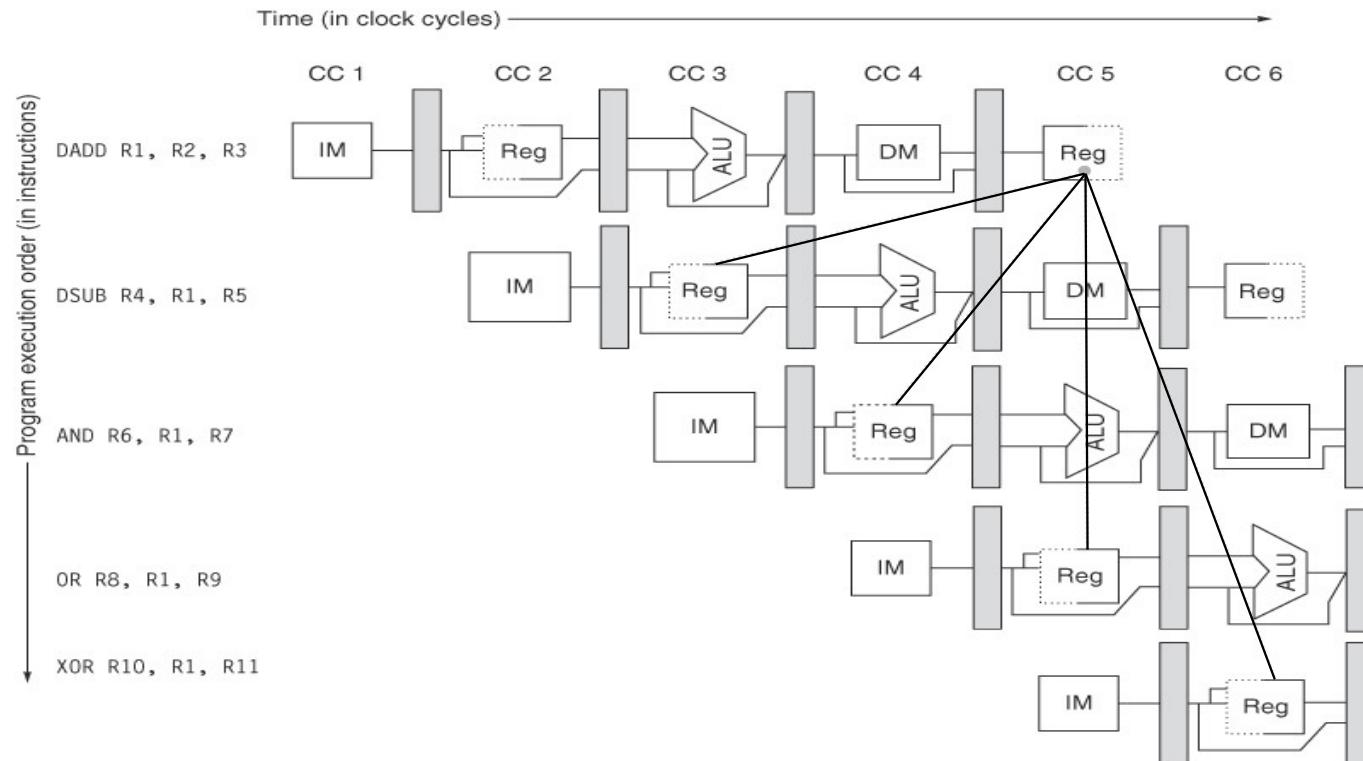
The DADD updates the value of R1 in the WB stage, but DSUB reads R1 prior to this in its ID stage. Unless precautions are taken to prevent it, DSUB will read a wrong value. The value read is not even deterministic. It seems logic to assume that DSUB will always get the value assigned to R1 by some instruction prior to DADD, but this may not be the case. If an interrupt is serviced between the DADD and the DSUB instructions, the WB stage of DADD will complete and the value of R1 at that point will be the result of DADD.

The AND instruction is also affected by this hazard. The writing of R1 does not complete until the first half of clock cycle 5. Thus, AND that reads the register at the second half of clock cycle 4, will receive a wrong result. The OR and XOR instructions, however, execute properly: the former will read R1 at the second half of clock cycle 5, after the writing has taken place; the latter reads R1 only at clock cycle 6.

Data hazards - 3

The effect of the DADD instruction on the instructions that follow

Source: Computer Architecture: A Quantitative Approach



Data hazards - 4

The problem can be solved through the use of a simple hardware technique called *forwarding*, *bypassing*, or *short-circuiting*. The key insight in *forwarding* is that the result is not really needed by DSUB, or by AND, until after DADD actually produces it. If the result can be moved from the register where DADD stores it to where DSUB and AND need it, then the introduction of stalls may be avoided.

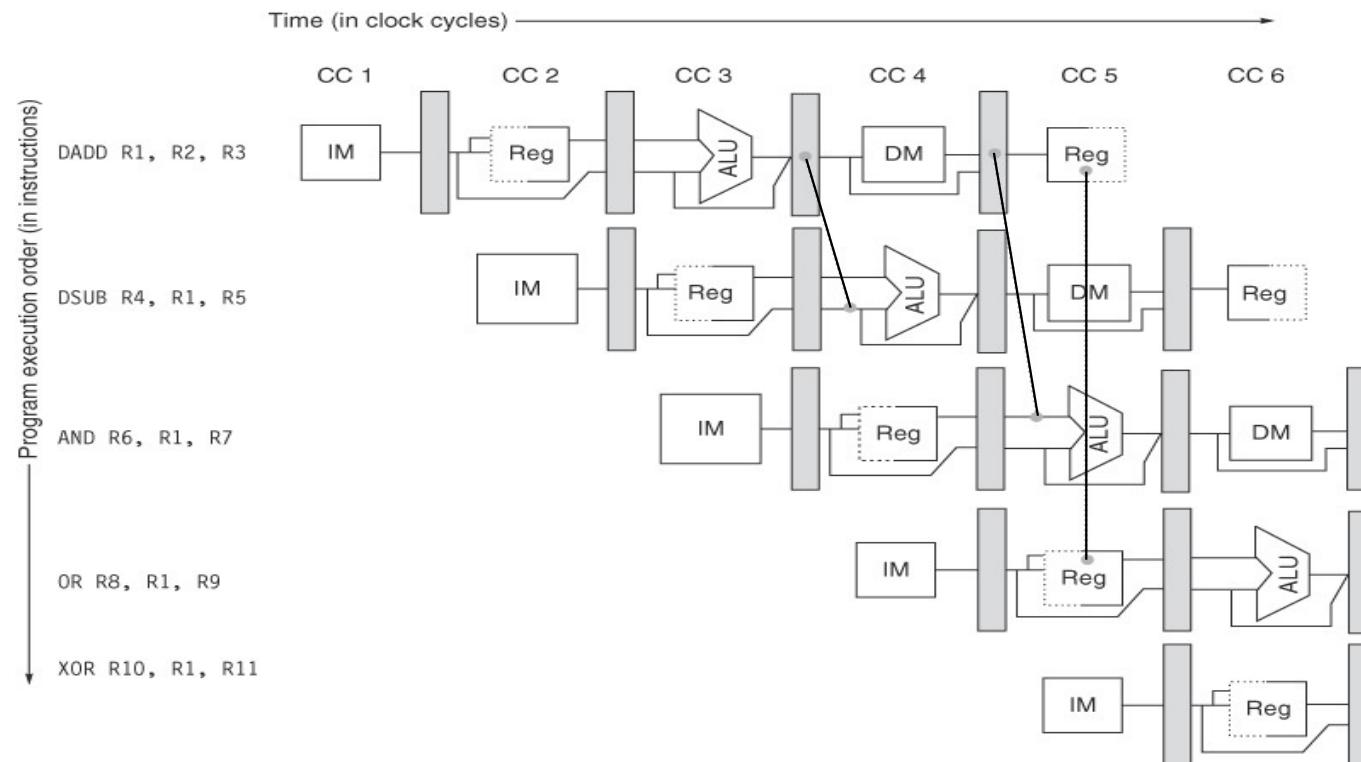
Using this observation, *forwarding* works as follows

- the ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs
- if the forwarding hardware detects that a previous ALU operation has modified the register corresponding to a source for the current ALU operation, control logic selects the forward result as the ALU input rather than the value read from the register bank.

Data hazards - 5

The effect of the DADD instruction on the instructions that follow solved by forwarding to avoid a data hazard

Source: Computer Architecture: A Quantitative Approach



Data hazards - 6

Forwarding can be generalized to include passing a result to the functional unit that requires it. A result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Consider the pipelined execution of the code

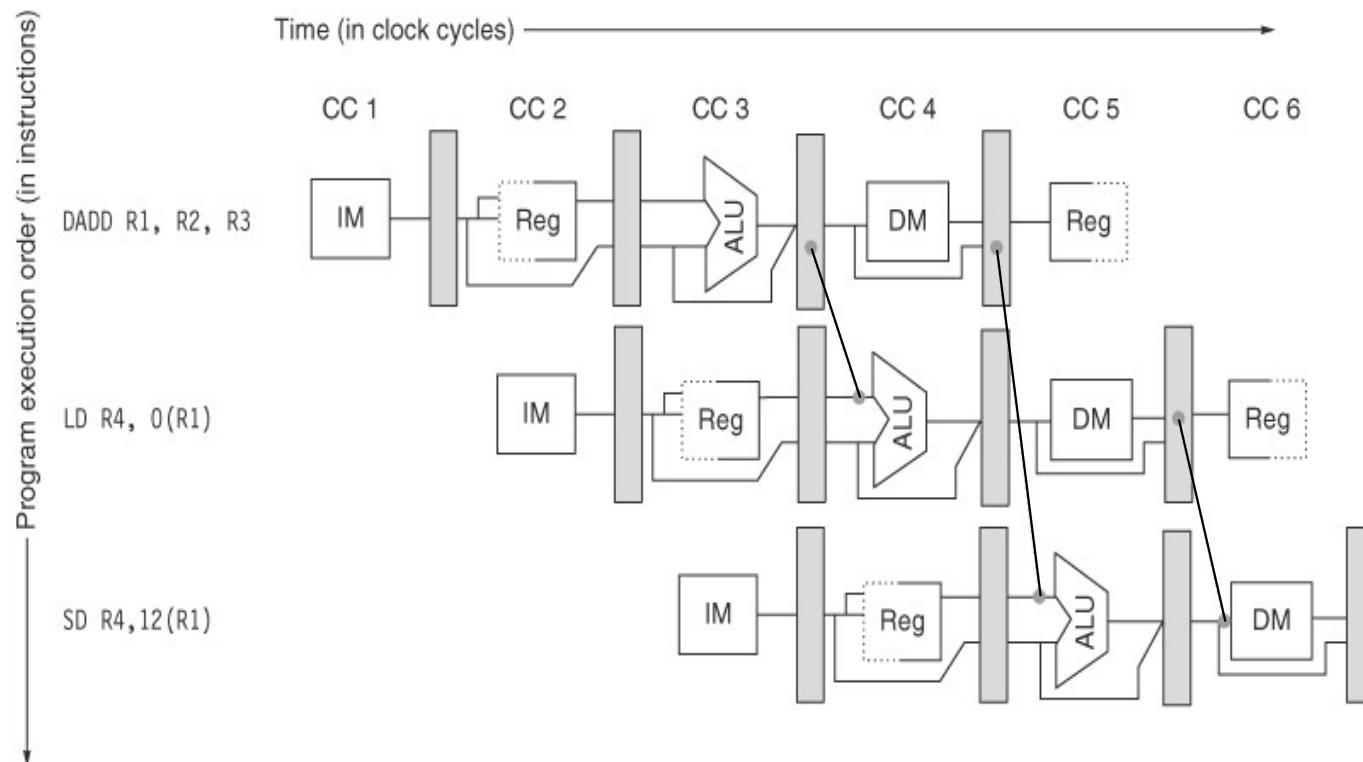
DADD	R1, R2, R3
LD	R4, 0 (R1)
SD	R4, 12 (R1)

To prevent a stall in this sequence, one needs to forward the values of the ALU output and the data memory output from the pipeline registers to the ALU and the data memory inputs.

Data hazards - 7

Forwarding of operand required by store during MEM

Source: Computer Architecture: A Quantitative Approach



Data hazards - 8

Unfortunately, not all potential data hazards can be handled by forwarding.

Consider the pipelined execution of the code

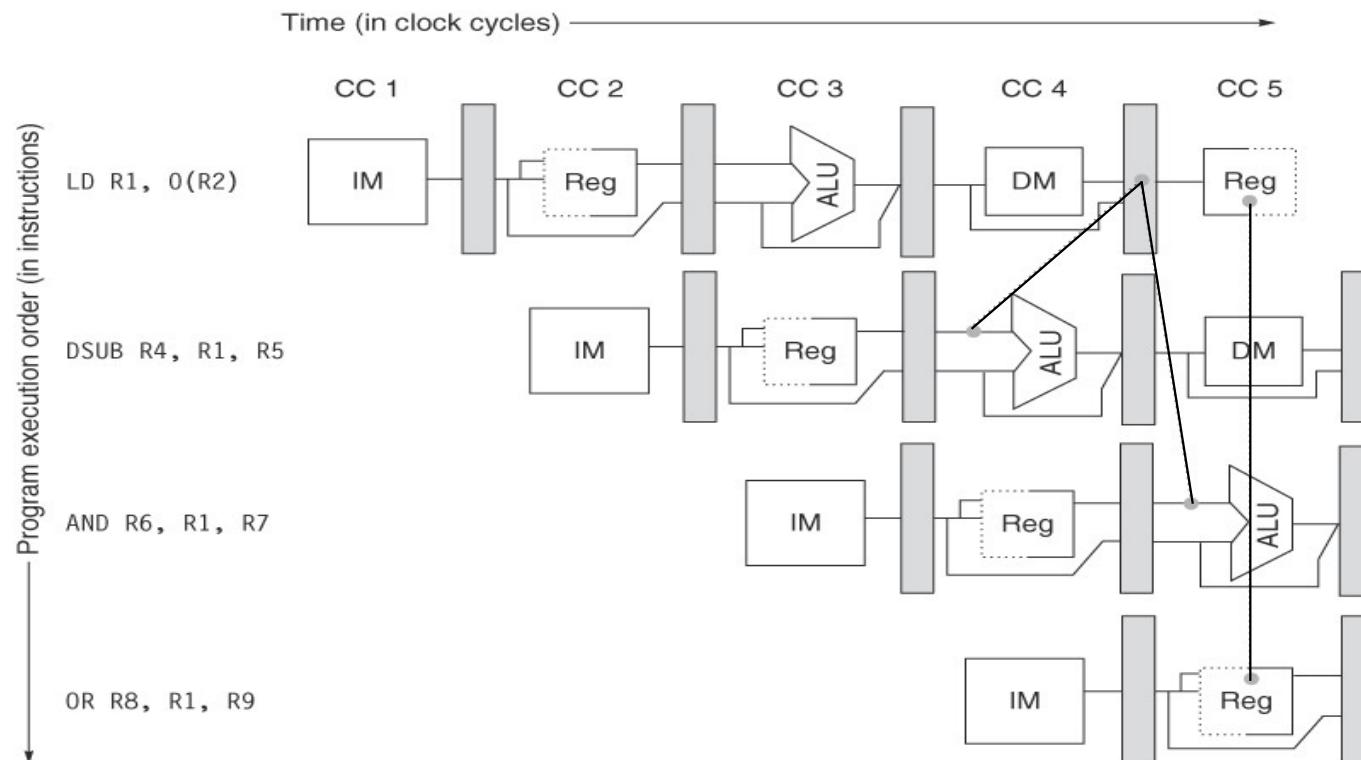
LD	R1, 0 (R2)
DSUB	R4, R1, R5
AND	R6, R1, R7
OR	R8, R1, R9

The LD instruction does not have the data until the end of clock cycle 4, while the DSUB instruction requires it at the beginning of this cycle. Thus, the hazard from using the result of a load instruction cannot be completely eliminated. The result can be forwarded immediately to the ALU for use by the AND instruction, which begins 2 clock cycles after LD. Likewise, the OR instruction has no problem since it gets the value through the register file.

Data hazards - 9

Data hazard that cannot be solved by forwarding

Source: Computer Architecture: A Quantitative Approach



Data hazards - 10

To solve the problem, one needs to add special hardware, called *pipeline interlock*, to preserve the correct execution pattern. In general, the *pipeline interlock* detects a hazard and stalls the pipeline until the hazard disappears. In this case, the *pipeline interlock* stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it and makes it available.

<i>Instruction</i>	<i>Clock number</i>								
	1	2	3	4	5	6	7	8	9
LD R1, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Control hazards - 1

When a branch is executed, it may or may not change the PC. It is said that if a branch instruction changes the PC to its target address, it is a *taken* branch; if it falls through, it is an *untaken* branch. When an instruction is a taken branch, the PC is not changed until the end of the ID stage.

The simplest method to deal with branches is to redo the fetch of the instruction following the branch, once the branch instruction is detected (during the ID stage). The first IF cycle is essentially a stall, because it never performs useful work. One should notice that if a branch is untaken, there is a penalty, the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched.

<i>Instruction</i>	<i>Clock number</i>								
	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
<i>branch</i>	IF	ID	EX	MEM	WB				
<i>successor</i>		IF	IF	ID	EX	MEM	WB		
<i>successor+1</i>				IF	ID	EX	MEM	WB	
<i>successor+2</i>					IF	ID	EX	MEM	WB

Control hazards - 2

A higher-performance alternative, called *predicted-untaken* scheme, is to treat every branch as *untaken*, simply allowing the hardware to continue as if the branch instruction were not executed. The complexity of this scheme arises from the need to ascertain when the state have been changed by an instruction and to revert such a change. The pipeline looks as if nothing out of the ordinary is happening in this case. If the branch is *taken*, however, the fetched instruction is turned into a *no-op* instruction and the fetch stage is restarted at the target address.

<i>Instruction</i>	<i>Clock number</i>							
	1	2	3	4	5	6	7	8
<i>untaken branch</i>	IF	ID	EX	MEM	WB			
<i>successor</i>		IF	ID	EX	MEM	WB		
<i>successor+1</i>			IF	ID	EX	MEM	WB	
<i>successor+2</i>				IF	ID	EX	MEM	WB

<i>Instruction</i>	<i>Clock number</i>							
	1	2	3	4	5	6	7	8
<i>taken branch</i>	IF	ID	EX	MEM	WB			
<i>successor</i>		IF	<i>idle</i>	<i>idle</i>	<i>idle</i>			
<i>target</i>			IF	ID	EX	MEM	WB	
<i>target+1</i>				IF	ID	EX	MEM	WB

Control hazards - 3

One may think the other way around and treat every branch as *taken*. This scheme is obviously called the *predicted-taken* scheme. It is only relevant for processors that use implicit set condition codes or more powerful, and hence slower, branch conditions – the branch target address is known before the branch outcome and, because of that, the *predicted-taken* scheme might make sense.

In either of the predicted schemes, the compiler plays an important role since it can improve performance by organizing the code so that the most frequent path matches the hardware choice.

Control hazards - 4

A last scheme, called *delayed branch* scheme, was heavily used in early RISC processors. The execution cycle with a branch delay of one is defined as

branch instruction
sequential successor
branch target if taken

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken and can not be itself a branch instruction.

<i>Instruction</i>	<i>Clock number</i>							
	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>
<i>untaken branch</i>	IF	ID	EX	MEM	WB			
<i>branch delay</i>		IF	ID	EX	MEM	WB		
<i>successor</i>			IF	ID	EX	MEM	WB	
<i>successor+1</i>				IF	ID	EX	MEM	WB

<i>Instruction</i>	<i>Clock number</i>							
	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>
<i>taken branch</i>	IF	ID	EX	MEM	WB			
<i>branch delay</i>		IF	ID	EX	MEM	WB		
<i>target</i>			IF	ID	EX	MEM	WB	
<i>target+1</i>				IF	ID	EX	MEM	WB

Control hazards - 5

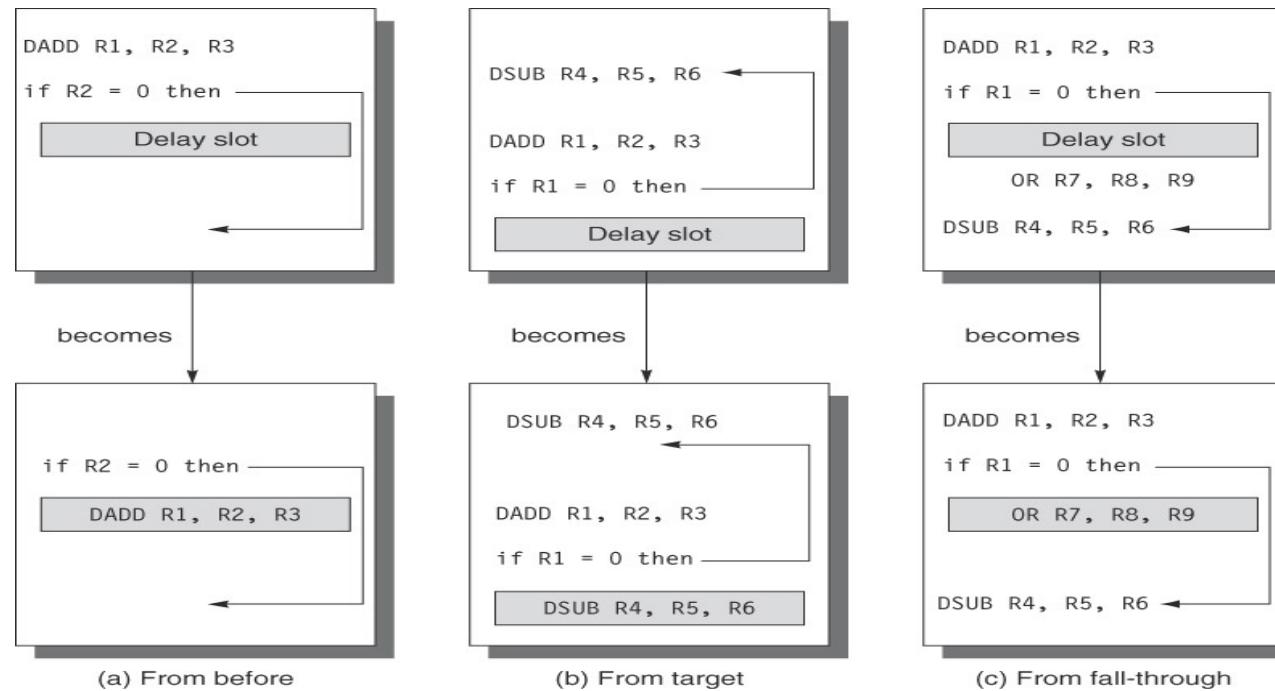
The job of the compiler is to make the sequential successor instruction valid and useful. Several alternatives are possible

- *from before* – insert in the *branch delay slot* an instruction supposed to be executed before the branch instruction and that is independent from it (the *ideal* situation since no side effects result from its application)
- *from target* – insert in the *branch delay slot* the instruction that the branch, if *taken*, points to and make the branch point to its successor (notice that when the branch is *not taken*, an extra instruction, not originally intended, is executed)
- *from fall-through* – insert in the *branch delay slot* the instruction that will be executed next if the branch is *not taken* (then this instruction is still being executed when the branch is *taken*).

Control hazards - 6

Branch delay slot compiler optimizations

Source: Computer Architecture: A Quantitative Approach



One has to bear in mind that the last two optimizations are only possible if they do not imply as side effect something which makes the program run incorrectly when the branch takes the unexpected direction!

Control hazards - 7

To improve the ability of the compiler to fill the branch delay slot, most processors with conditional branches have introduced a *cancelling* or *nullifying branch* instruction. It looks like the usual *delayed branch* instruction. When the branch is *taken*, the instruction in the *branch delay slot* is executed; however, when the branch is *not taken*, the instruction in the *branch delay slot* is turned into a *no-op* instruction and nothing happens.

The effective speed up of these schemes to deal with control hazards when the code is executed, can be expressed by

$$\begin{aligned} \text{speed up} &= \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + \text{pipeline stall cycles per instruction}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} \\ &= \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + (\text{branch frequency} \cdot \text{branch penalty})_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} . \end{aligned}$$

Control hazards - 8

As pipelines go deeper and the potential penalty of branches increases, delayed branches and similar schemes are deemed to be insufficient. It is necessary to be more aggressive in predicting branches.

The problem is approached in two different ways

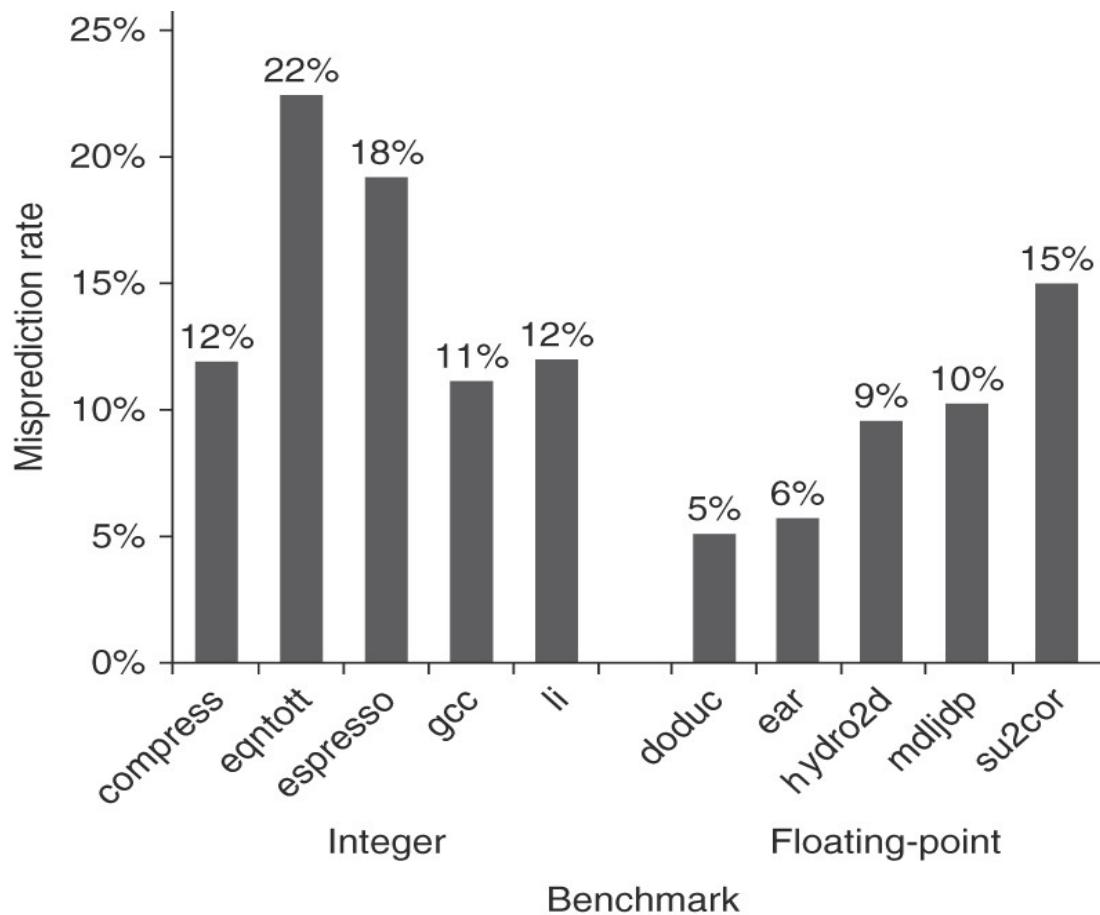
- *low-cost static schemes* – they rely on information available at compiler time and on profiling earlier runs of the same code; the key observation that makes this effort worthwhile is that branch behavior is often bimodally distributed, that is, each particular branch tends to be often highly biased towards *taken* or *not taken*
- *dynamic schemes* – they consist of methods to predict the branch direction during runtime.

In general, the misprediction rate for integer programs is higher than for floating-point programs, not only their branch frequency is higher, but also the branching direction is more random.

Control hazards - 9

Misprediction rate for SPEC92 using a profile-based predictor

Source: Computer Architecture: A Quantitative Approach



Control hazards - 10

The simplest dynamic branch-prediction scheme uses a *branch-prediction buffer* (BPB) or a *branch-prediction table* (BPT). A *branch-prediction buffer* is a small memory indexed by the lower part of the address of the branch instruction. The memory has a bit per position that states whether the branch was recently *taken* or *not taken*.

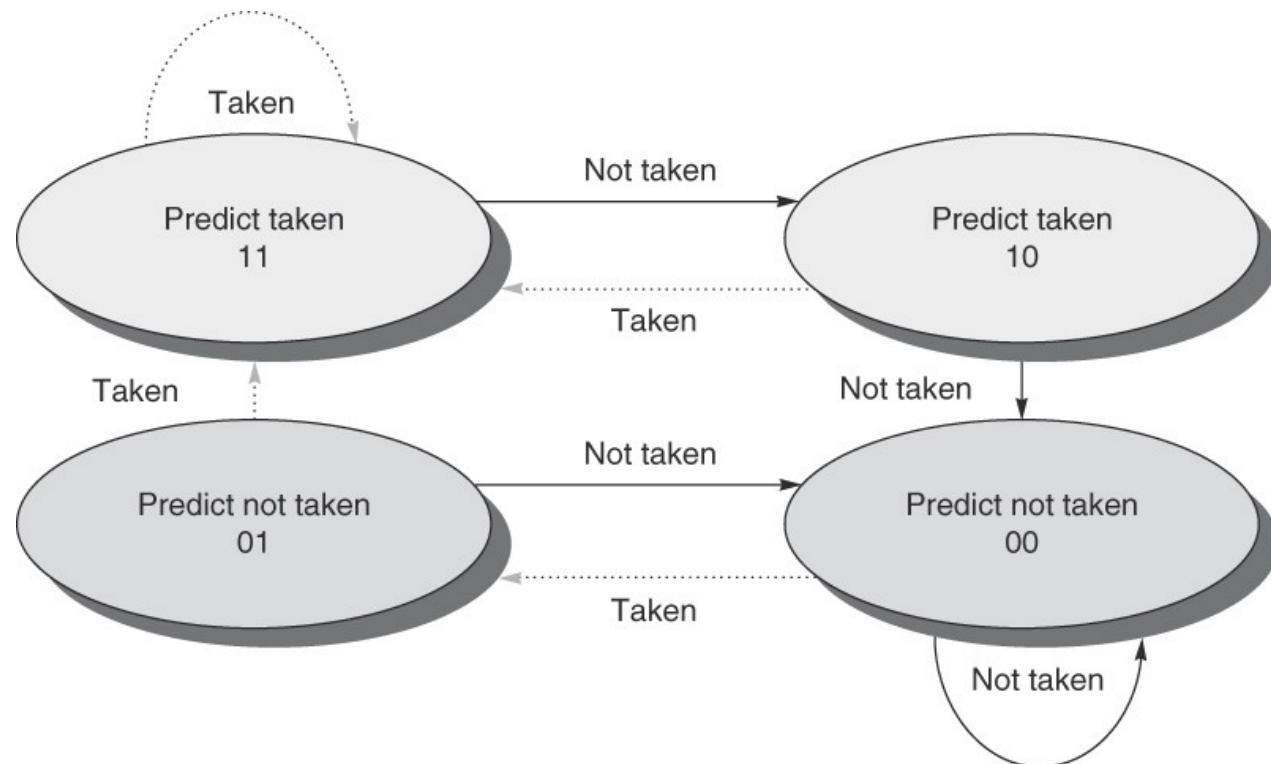
With such a device, one does not really know if the prediction is correct – it may have been set or reset by another branch instruction that shares the same low-order address bits, or now the branch behavior may be opposite to what was before. If one thinks a little about this, it is not determinant. A prediction is just a hint about the future one assumes to be true and, therefore, fetching proceeds in the predicted direction. If it is proved to be false later on, the prediction bit is complemented and fetching is restarted.

This simple 1-bit prediction scheme has a performance shortcoming: even when the branch is almost always taken, it will be likely to predict incorrectly twice, rather than once, when it was not taken. This can be remedied by using a 2-bit prediction scheme instead.

Control hazards - 11

The 2-bit prediction scheme introduces hysteresis in the prediction process.

2-bit prediction scheme state diagram
Source: Computer Architecture: A Quantitative Approach

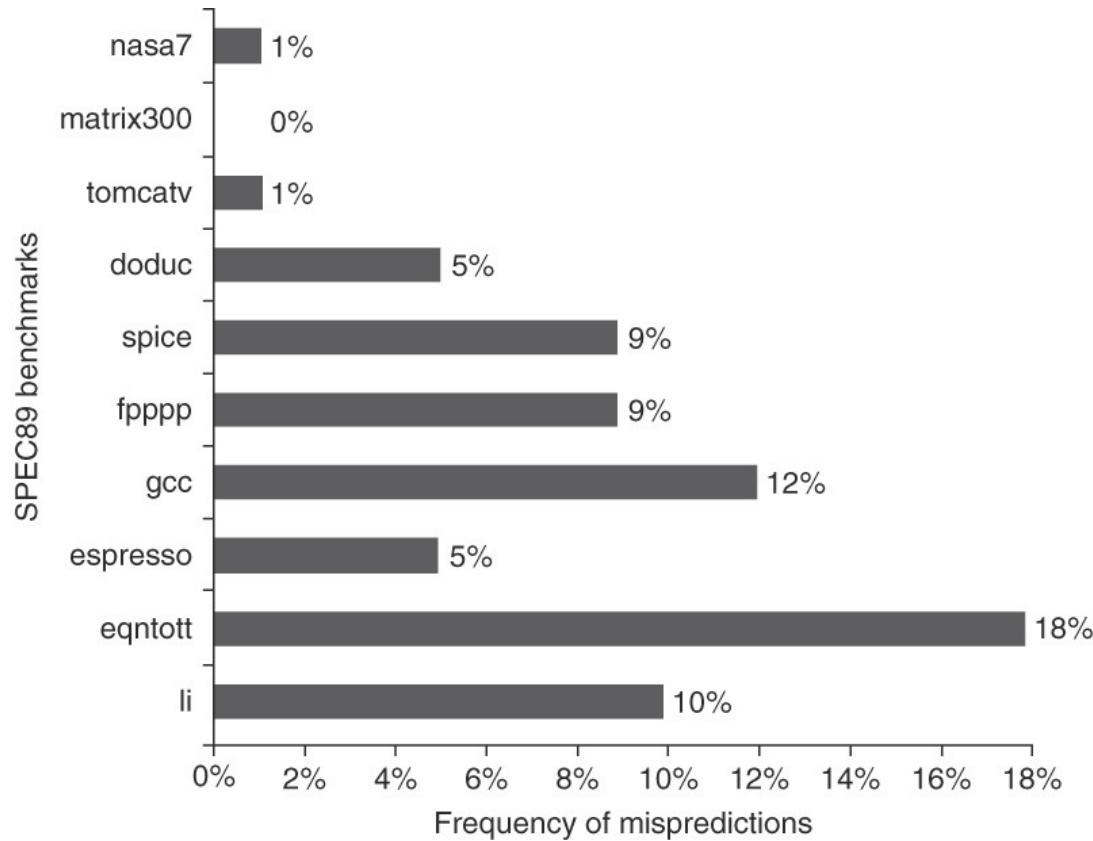


Control hazards - 12

The improvement over a profile-based predictor is real, but not spectacular.

Misprediction rate for SPEC89 using a 4096-entry 2-bit prediction buffer

Source: Computer Architecture: A Quantitative Approach



Control hazards - 13

Studies show that the hit rate of the buffer is not the major limiting factor. In fact, the behavior of a 4096-entry buffer is very similar to the behavior of a buffer with unlimited number of entries. Increasing the number of bits of the predictor without changing the predictor structure also has little impact.

Current approaches favor the use of both local and global information to improve prediction accuracy.

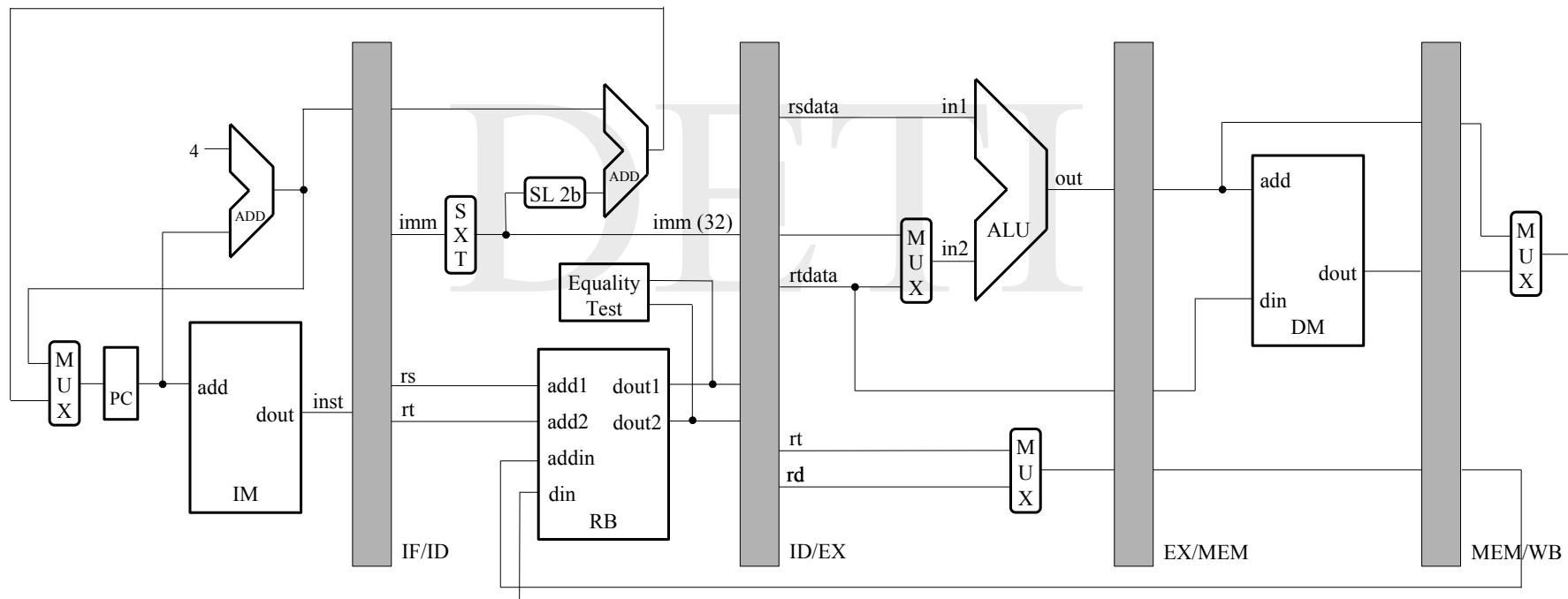
In the simplest case, a *correlating predictor* may consist of a 2-bit predictor for each branch, reporting its past history in different global situations.

More recently, *tournament predictors* have been used. A *tournament predictor* uses multiple predictors, tracking which yields the best result for each branch and taking the winner suggestion as the next prediction.

Implementation of classical 5-stage pipeline - 1

5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



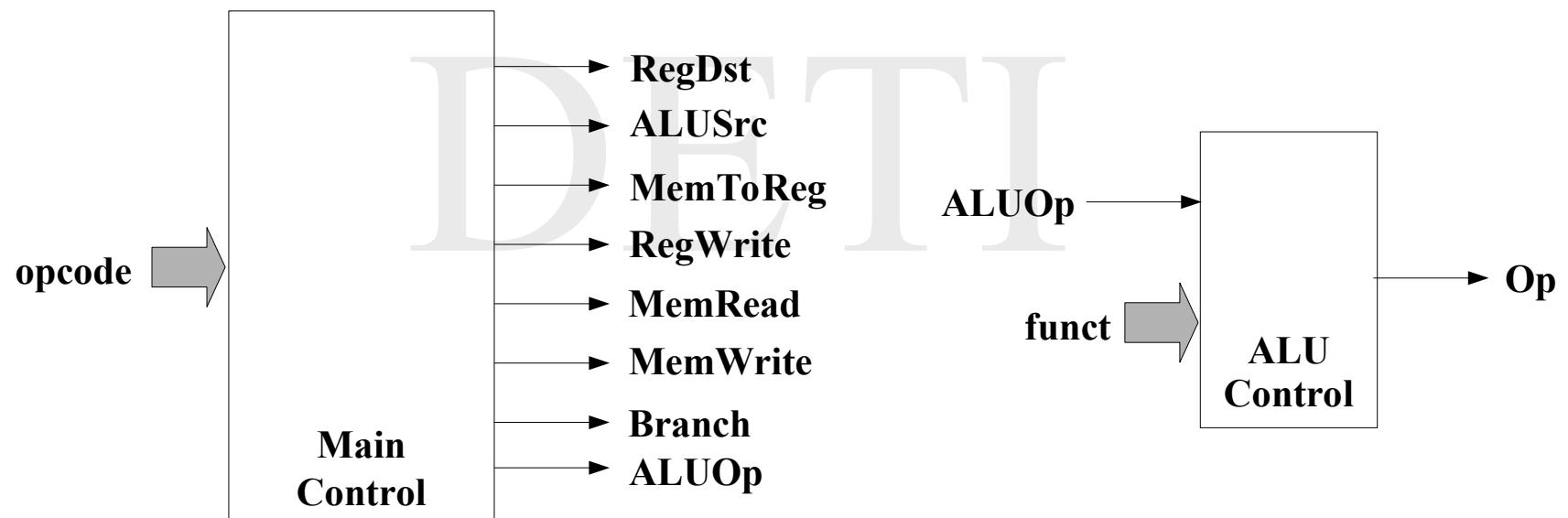
Implementation of classical 5-stage pipeline - 2

Classes of instructions to be considered

	R-type instruction					
arithmetic / logic instructions	31	25	20	15	10	5
	0	rs	rt	rd	shamt	funct
	I-type instruction					
	31	25	20	15		0
	opcode	rs	rt		immediate	
	31	25	20	15		0
load and store instructions	I-type instruction					
	31	25	20	15		0
	23 / 2B	rs	rt		offset (immediate)	
	31	25	20	15		0
branch instructions	I-type instruction					
	31	25	20	15		0
	4	rs	rt		offset (immediate)	

Implementation of classical 5-stage pipeline - 3

Pipeline Control Unit



Implementation of classical 5-stage pipeline - 4

Main Control Functional Description

Source: Adapted from Computer Organization and Design: The Hardware/Software Interface

Instruction Type	OpCode	Branch	Execution / Effective Address Stage				Memory Access Stage		Write Back Stage	
			RegDst	ALUSrc	ALUOp1	ALUOp2	MemRead	MemWrite	RegWrite	MemToReg
R-format	0x00	0	1	0	1	0	0	0	1	0
lw	0x23	0	0	1	0	0	1	0	1	1
sw	0x2B	0	x	1	0	0	0	1	0	x
beq	0x04	1	x	0	0	0	0	0	0	x

Signal Name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	the register destination number for the write register comes from the rt field (bits 20:16)	the register destination number for the write register comes from the rd field (bits 15:11)
RegWrite	none	the register whose number is in addin is written with the value in din
ALUSrc	the ALU second operand comes from the contents of the register whose number is in rt field (bits 20:16)	the ALU second operand is the immediate field (bits 15:0) after sign-extending it to 32 bits
PCSrc	the PC is replaced by the output of the adder that computes the value of PC+4	the PC is replaced by the output of the adder that computes the branch target address
MemRead	none	the contents of the memory location referenced by add is placed at dout
MemWrite	none	the contents of the memory location referenced by add is replaced by the value at din
MemToReg	the value fed to the data input of the write register comes from the ALU	the value fed to the data input of the write register comes from the dout of data memory

Implementation of classical 5-stage pipeline - 5

ALU Control Functional Description

Source: Adapted from Computer Organization and Design: The Hardware/Software Interface

Instruction	ALUOp	Function	Desired ALU action	Operation
lw	00	xxxxxx	add	0010
sw	00	xxxxxx	add	0010
add	10	100000	add	0010
subtract	10	100010	subtract	0110
and	10	100100	and	0000
or	10	100101	or	0001
slt	10	101010	set on less than	0111

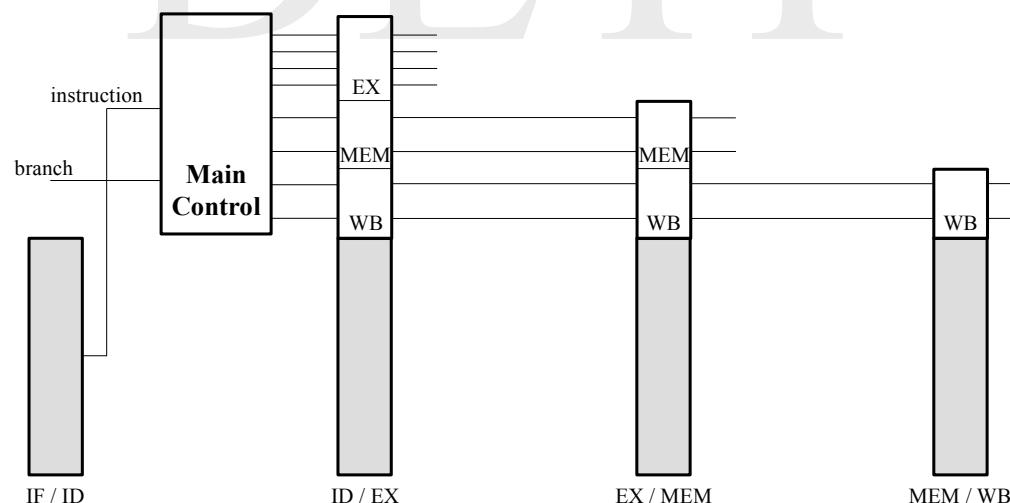
Implementation of classical 5-stage pipeline - 6

In order control circuitry to work properly, the control signals must be set to the right values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

Since control signals start to be applied at the EX stage, the control information can be created during the ID stage.

Deployment of control signals through the different pipe stages

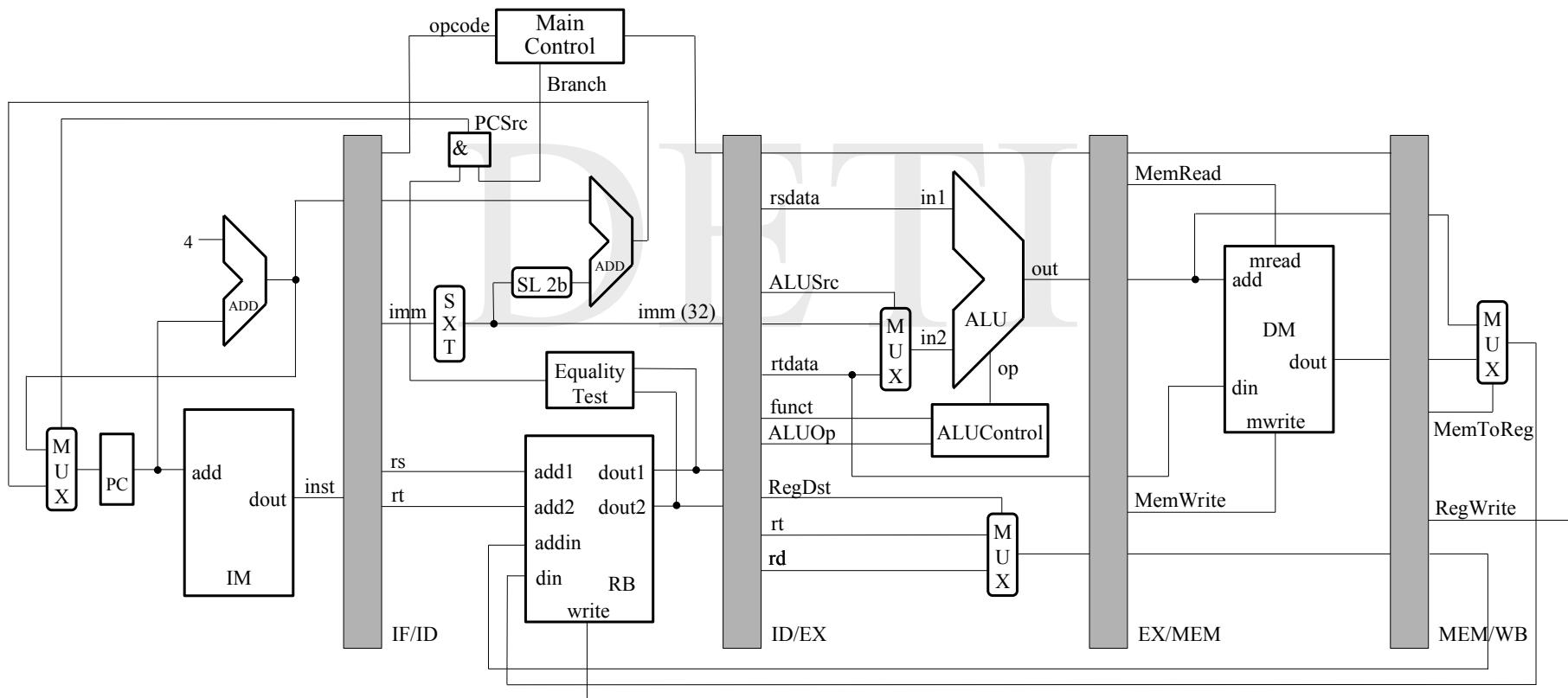
Source: Adapted from Computer Organization and Design: The Hardware/Software Interface



Implementation of classical 5-stage pipeline - 7

Integrating control in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Implementation of classical 5-stage pipeline - 8

Data hazards that can be solved by *forwarding* for this specific 5-stage pipeline implementation, may be classified in the following classes

- *ID hazard* – the contents of a register of the register bank is required by a *beq* instruction for equality test and has been modified by a prior instruction and its updated value is presently stored in a ID/EX, EX/MEM or MEM/WB pipeline register
- *EX hazard* – the contents of a register of the register bank is required by an instruction and has been modified by a prior instruction and its updated value is presently stored in a EX/MEM or MEM/WB pipeline register
- *MEM hazard* – the contents of a register of the register bank is required by an instruction and has been modified by a prior instruction and its updated value is presently stored in a MEM/WB pipeline register.

Implementation of classical 5-stage pipeline - 9

Detecting ID data hazards

```
if (Branch && ID/EXE.RegWrite && (ID/EXE.RegDest ≠ 0) &&  
    (ID/EXE.RegDest == IF/ID.RegRs) )  
    fwdIDA = 01  
else if (Branch && EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&  
    (EX/MEM.RegDest == IF/ID.RegRs) )  
    fwdIDA = 10  
else if (Branch && MEM/WB.RegWrite) &&  
    (MEM/WB.RegDest ≠ 0) &&  
    (MEM/WB.RegDest == IF/ID.RegRs) )  
    fwdIDA = 11  
else fwdIDA = 00
```

Implementation of classical 5-stage pipeline - 10

Detecting ID data hazards (cont.)

```
if (Branch && ID/EXE.RegWrite && (ID/EXE.RegDest ≠ 0) &&  
    (ID/EXE.RegDest == IF/ID.RegRt) )  
    fwdIDB = 01  
else if (Branch && EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&  
    (EX/MEM.RegDest == IF/ID.RegRt) )  
    fwdIDB = 10  
else if (Branch && MEM/WB.RegWrite) &&  
    (MEM/WB.RegDest ≠ 0) &&  
    (MEM/WB.RegDest == IF/ID.RegRt) )  
    fwdIDB = 11  
else fwdIDB = 00
```

Implementation of classical 5-stage pipeline - 11

Detecting EX data hazards

```
if (EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
    (EX/MEM.RegDest == ID/EX.RegRs) )
    fwdEXA = 10
else if (MEM/WB.RegWrite && (MEM/WB.RegDest ≠ 0) &&
         (MEM/WB.RegDest == ID/EX.RegRs) )
    fwdEXA = 11
else fwdEXA = 00

if (EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
    (EX/MEM.RegDest == ID/EX.RegRt) && !ID/EX.ALUSrc )
    fwdEXB = 10
else if (MEM/WB.RegWrite && (MEM/WB.RegDest ≠ 0) &&
         (MEM/WB.RegDest == ID/EX.RegRt) && !ID/EX.ALUSrc )
    fwdEXB = 11
else if (ID/EX.ALUSrc )
    fwdEXB = 01
else fwdEXB = 00
```

Implementation of classical 5-stage pipeline - 12

Detecting MEM data hazards

```
if ( EX/MEM.MemWrite && (EX/MEM.RegDest ≠ 0) &&  
    MEM/WB.RegWrite &&  
    (EX/MEM.RegDest == MEM/WR.RegDest) )  
    fwdMEM = 1  
else fwdMEM = 0
```

Implementation of classical 5-stage pipeline - 13

Selection values for the forwarding multiplexors at the ID stage

MUX Selection	Source	Explanation
fwdIDA= 00	IF/ID	The first ALU operand comes from the register file
fwdIDA= 01	ID/EX	The first ALU operand is forwarded from the last ALU result
fwdIDA= 10	EX/MEM	The first ALU operand is forwarded from first to last ALU result
fwdIDA= 11	MEM/WB	The first ALU operand is forwarded from data memory or second to last ALU result
fwdIDB = 00	IF/ID	The second ALU operand comes from the register file
fwdIDB = 01	ID/EX	The second ALU operand is forwarded from the last ALU result
fwdIDB = 10	EX/MEM	The second ALU operand is forwarded from first to last ALU result
fwdIDB = 11	MEM/WB	The second ALU operand is forwarded from data memory or second to last ALU result

Implementation of classical 5-stage pipeline - 14

Selection values for the forwarding multiplexors at the EX stage

MUX Selection	Source	Explanation
fwdEXA= 00	ID/EX	The first ALU operand comes from the register file
fwdEXA= 10	EX/MEM	The first ALU operand is forwarded from the last ALU result
fwdEXA= 11	MEM/WB	The first ALU operand is forwarded from data memory or first to last ALU result
fwdEXB = 00	ID/EX	The second ALU operand comes from the register file
fwdEXB = 01	ID/EX	The second ALU operand is the immediate field sign-extended to 32 bits
fwdEXB = 10	EX/MEM	The second ALU operand is forwarded from the last ALU result
fwdEXB = 11	MEM/WB	The second ALU operand is forwarded from data memory or first to last ALU result

Implementation of classical 5-stage pipeline - 15

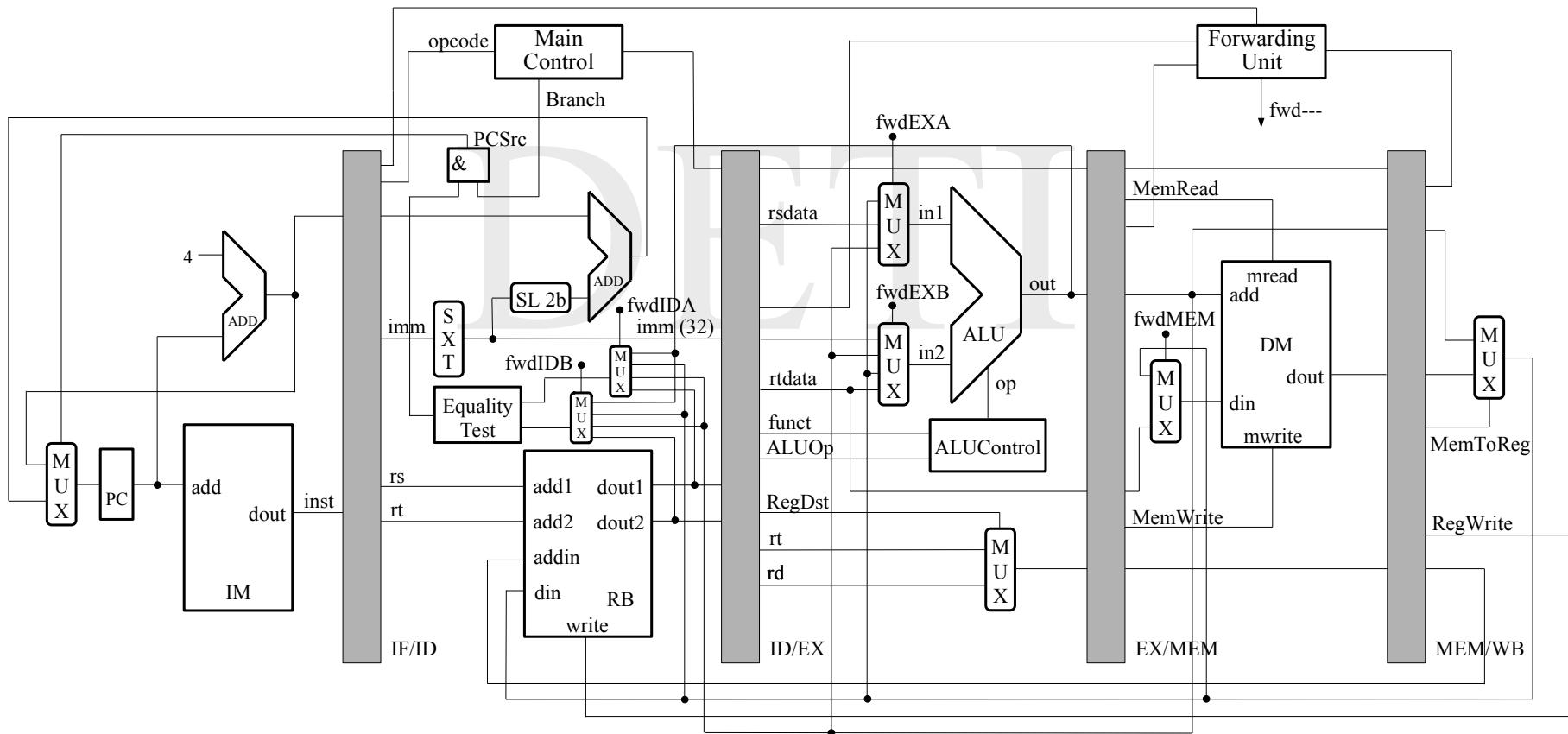
Selection values for the forwarding multiplexors at the MEM stage

MUX Selection	Source	Explanation
fwdMEM = 0	EX/MEM	The value to be possibly written in data memory comes from the contents of register rt
fwdMEM = 1	MEM/WB	The value to be written in data memory is the value that has just been produced

Implementation of classical 5-stage pipeline datapath - 16

Integrating forwarding in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Implementation of classical 5-stage pipeline - 17

Not all the *data hazards* for this specific 5-stage pipeline implementation can be solved by *forwarding*. As it was seen, if an instruction immediately following a load tries to read the same register which is written by it, there is no way for this instruction to proceed until the new value is effectively retrieved from memory. So the progress of this instruction in the pipeline must be *stalled*.

Furthermore, and as far as *control hazards* are concerned, if a *delayed branch* scheme is assumed, no new abnormal situations that would require the stalling of instruction progress, need to be considered.

When the instruction that follows the `load` is a ALU instruction, one stall cycle is required; when it is a `beq` instruction, two stall cycles are required.

Implementation of classical 5-stage pipeline - 18

Detecting the need for the insertion of stall cycles

```
if ( ID/EX.MemRead && !MemWrite &&  
    ( (RegDest == IF/ID.RegRs) ||  
      (RegDest == IF/ID.RegRt && !MemRead) )  
    stall the pipeline  
else if (Branch && EX/MEM.MemRead &&  
    ( (EX/MEM.RegDest == IF/ID.RegRs) ||  
      (EX/MEM.RegDest == IF/ID.RegRt) )  
    stall the pipeline
```

Implementation of classical 5-stage pipeline - 19

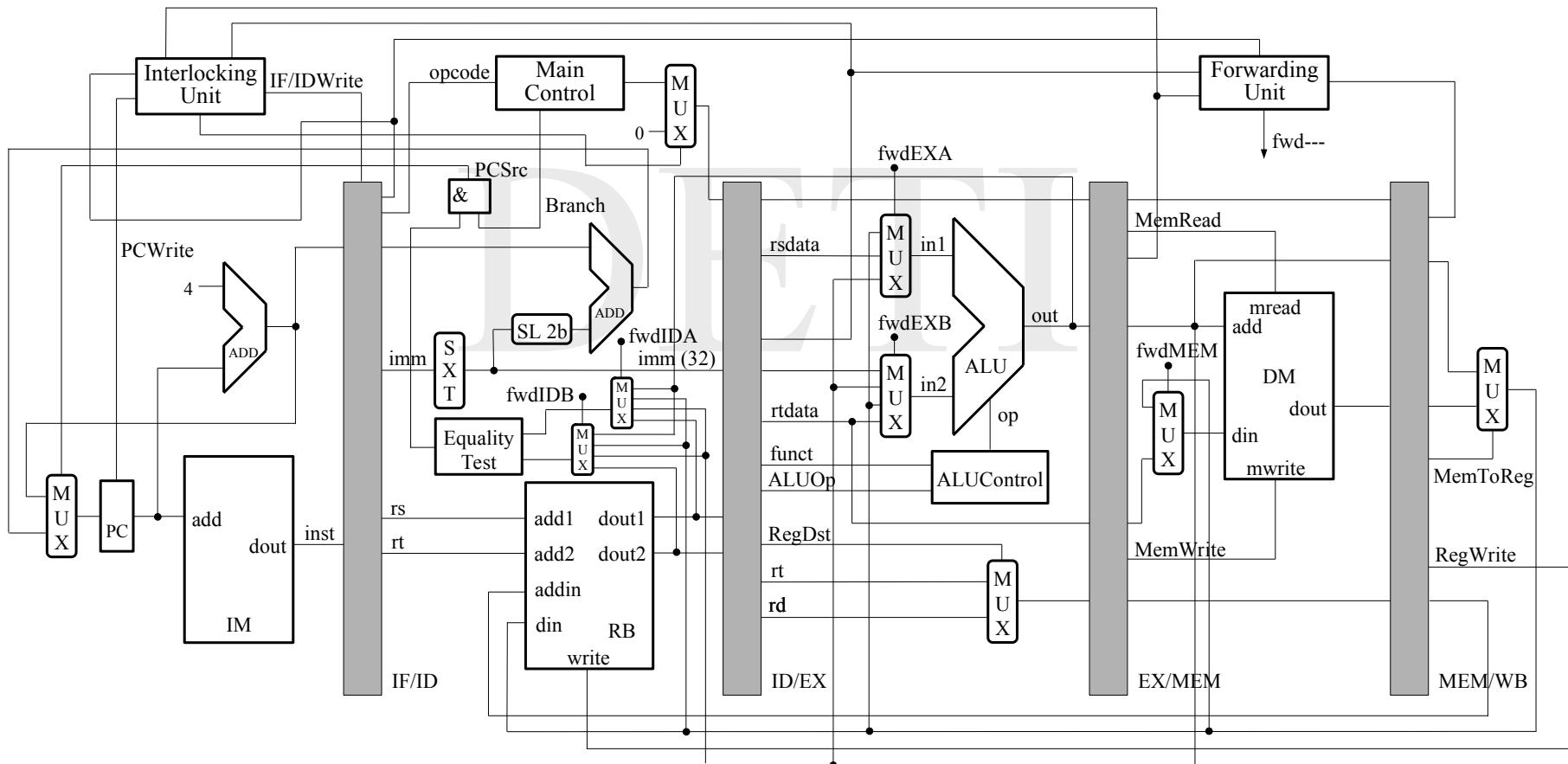
In order for the instruction at the ID stage to be stalled, the instruction at the IF stage must also be stalled. A simple way to achieve this is to prevent both the PC and the IF/ID pipeline registers from changing. Provided the contents of these registers is preserved, the instructions processed at the IF and ID stages are kept the same.

On the other hand, the instructions at the other stages must proceed as if nothing had happened. This means that a *no-op* instruction must be generated and inserted in the EX stage at the next clock cycle. This can be achieved by disabling the control signals when they are written to the ID/EX pipeline registers.

Implementation of classical 5-stage pipeline datapath - 20

Integrating interlocking in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Exceptions - 1

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make go fast. One of the hardest parts of control is allowing for *exceptions* or *interrupts* – events that cause the breaking of the strict sequentiality of instruction execution, other than branches and jumps.

Typical exceptions include

- requests of attention by a controller associated to a I/O device
- invoking a system call from the user level
- tracing instruction execution or defining breakpoints
- integer overflow or floating-point anomaly
- page fault in a virtual memory organization
- misaligned memory access
- memory protection violation
- trying to execute an undefined or unimplemented instruction
- hardware malfunction
- power failure.

Exceptions - 2

Individual exceptions have important characteristics that determine what action the hardware is required to perform. These requirements can be classified in five semi-independent categories

- *synchronous* vs. *asynchronous* – unlike asynchronous events, synchronous events occur at the same place every time the program is executed with the same data and memory allocation; asynchronous events, in contrast, may occur anywhere within the program and can be usually handled after the completion of the current instruction
- *user requested* vs. *coerced* – user requested exceptions, being predictable, are not really in some sense true exceptions; they are only treated as such because the same mechanism, which is used to save and restore the program state, is also applied to them; since the only function of the instruction that triggers this kind of exception is to cause the exception itself, they can always be handled after the instruction is completed; coerced exceptions, in contrast, are caused by some hardware event that the program does not control and are totally unpredictable

Exceptions - 3

- *maskable* vs. *non-maskable* – some exceptions allow the program to choose the moment the hardware responds to them
- *within* vs. *between instructions* – an event may prevent the completion of an instruction in execution, when it is triggered by it, because some malfunction or anomaly has occurred at the software / hardware level; the corresponding exceptions are usually synchronous and hard to implement because the instruction must be stopped and, eventually, restarted later; asynchronous exceptions that occur within instructions, in contrast, arise from catastrophic situations and always cause program termination
- *resume* vs. *terminate* – exceptions that do not require the program to resume after they are handled, are easier to implement because there is no need to restart the program.

Exceptions - 4

How common exceptions stand in the 5-category classification scheme

Source: Adapted from Computer Architecture: A Quantitative Approach

Exception Type	Synchronous vs. Asynchronous	User requested vs. Coerced	Maskable vs. Non-maskable	Within vs. Between instructions	Resume vs. Terminate
I/O device request	asynchronous	coerced	maskable	between	resume
operating system invocation	synchronous	user requested	maskable	between	resume
tracing instruction execution or defining breakpoints	synchronous	user requested	maskable	between	resume
integer overflow or FP anomaly	synchronous	coerced	maskable	within	resume
page fault (virtual memory)	synchronous	coerced	non-maskable	within	resume
misaligned memory access	synchronous	coerced	non-maskable	within	terminate
memory protection violation	synchronous	coerced	non-maskable	within	terminate
undefined or unimplemented instruction execution	synchronous	coerced	non-maskable	within	terminate
hardware malfunction	asynchronous	coerced	non-maskable	within	terminate
power failure	asynchronous	coerced	non-maskable	within	terminate

Exceptions - 5

As in non-pipeline organizations, the difficult task is implementing exceptions occurring within instructions, typically at the EX or the MEM stages, which have to be resumed. The implementation supposes that another program, in principle the operating system, is invoked to save the state of the executing program, correct the cause of the exception and restore the state of the program, before the instruction that caused the exception is executed again – a mechanism that has obviously to be totally transparent to the executing program.

If the pipeline has the ability to handle the exception, save the state and restart without affecting program execution, the processor is said to be *restartable*. While early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it rests at the base of making operational virtual memory organization.

Exceptions - 6

For the operating system to be able to handle an exception, it must know the reason that has triggered it, in addition to the instruction that has caused it or that would next be executed if the exception was not serviced. There two main methods to communicate the reason for an exception

- *cause register* – it is a status register that holds a field describing the cause for an exception that has occurred, its value being set by the hardware upon its detection; when the exceptions are serviced by the same entry point address, this is the means the operating system has to determine the cause of the exception
- *vectored exceptions* – there are multiple entry point addresses for service of the exceptions; in general, each entry point address is associated with a particular exception so the identification of the cause by the operating system becomes trivial.

Exceptions - 7

When an exception is serviced, the pipeline control must take the following steps to save the program state and allow the program to resume later on, if this is to be the case.

1. The current value of PC is saved and is replaced by the address of the entry point corresponding to the exception for the next IF cycle. The processor is placed at a privileged mode where a wider instruction set is available and all maskable exceptions of the same or lower priority level are disabled.
2. All succeeding instructions currently in the pipeline, and the instruction itself if the exception is within, have to be turned into *no-op* instructions for the remaining pipe stages. All preceding instructions, if any, on the other hand, must be allowed to complete so that the program state is consistent at the time of the processing of the exception.

Exceptions - 8

3. After the exception service routine in the operating system starts executing, it immediately updates the saved PC value to the correct value, which depends on the cause of the exception (within or in between, and if the former case is true, on the pipe stage that has triggered it).

One should also notice that, if a *delayed branch* scheme is assumed, it is no longer possible to recreate the state of the processor with the storage of a single PC value – the instructions in the pipeline may not be sequentially related. In order for the exception service routine to be able to update the saved PC to the correct value, one needs as many PC addresses as the length of the *branch delay slot* plus 1.

4. Finally, after the termination of the exception service routine, special instructions of the *return from exception* type restart the former instruction stream by restoring the PC value and the former mode of execution.

Exceptions - 9

If the pipelined can be stopped so that the instructions just preceding the faulting instruction are completed and itself, together with those succeeding it, can be restarted from scratch, the pipeline is said to have *precise exceptions*.

Ideally, the faulting instruction should not change the state of execution and, therefore, for correctly handling some exceptions, it is required that the faulting instruction produces no effects. For other exceptions, however, such as floating-point exceptions, the faulting instruction in some processors writes its result before the exception can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is the same as one of the source operands.

To overcome this problem, many recent high-performance processors have introduced two modes of operation: one mode has precise exceptions and the other, to be more performant, does not. Indeed, the precise exception mode must be slower because it has to allow a lot less overlapping among floating-point instructions.

Exceptions - 10

With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in simultaneous execution.

Typical *within* exceptions that may occur in the classical 5-stage pipeline

Source: Computer Architecture: A Quantitative Approach

Pipeline Stage	Exceptions
IF	page fault on instruction fetch misaligned memory access memory protection violation
ID	undefined or unimplemented instruction
EX	arithmetic exception
MEM	page fault on data fetch misaligned memory access memory protection violation
WB	none

Exceptions - 11

Consider the instruction sequence

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

This pair of instructions can cause a data page fault and an arithmetic exception in the same clock cycle. The way to deal with this problem is to respond to the data page exception and then restart execution. The arithmetic exception will reoccur and only then can be handled independently.

In general, things are not so straightforward. Exceptions may occur *out of order*, that is, an instruction may trigger an exception before a preceeding one, which is in simultaneous execution, also triggers its own. To visualize this, take for instance the situation where the *load* instruction generates a page fault on data fetch at the MEM stage and the *add* instruction generates a page fault on instruction fetch at the IF stage.

To develop a *precise exceptions* pipeline implementation, the exception triggered by the *load* instruction must be handled first. How can that be done?

Exceptions - 12

The pipeline cannot handle exceptions as they occur in time, since by doing this the exceptions risk to be processed out of the non-pipelined order.

Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction. The status vector keeps moving along the pipe stages with the instruction. As soon as there is an exception indication in the status vector, any control signal that may cause a data value to be written, is deactivated (this means the register write and the memory write signals).

When the instruction enters the WB stage, the status vector is checked. If any exception is posted, it is handled in the order it would occur in time in a non-pipelined implementation.

Multicycle operations in classical 5-stage pipeline - 1

It is impractical to assume that all FP operations and integer multiplications and divisions will complete in one clock cycle, or even in two. Doing so would mean extending the clock period to allow the execution of the operations within it, or using an enormous amount of logic in the implementation of the functional units, or both together. Instead, the FP pipeline will contemplate a longer latency for the operations.

The best way to grasp the idea is imagining the FP instructions as having the same pipeline as the core integer instructions, with two important differences

- the EX cycle may be repeated as many times as needed to complete the operation – the number of repetitions can vary with the operation
- there may be multiple functional units.

A stall will occur if the instruction to be issued will cause either a structural hazard for the required functional unit, or a data hazard.

Multicycle operations in classical 5-stage pipeline - 2

For the sake of discussion, four separate functional units will be considered in the pipeline implementation

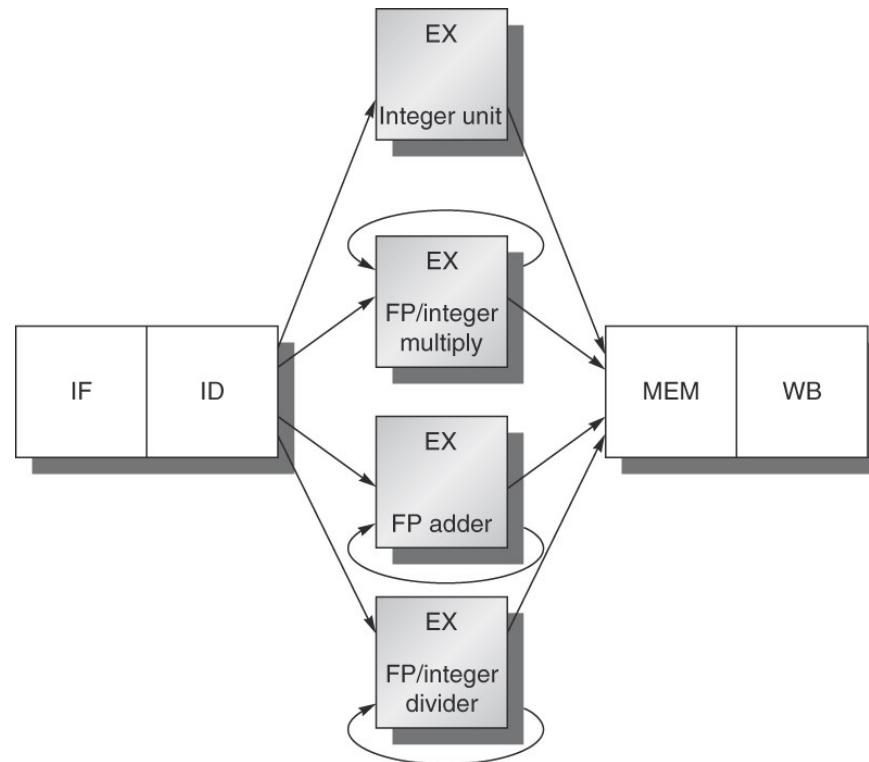
- the main integer unit, which handles loads, stores and the basic integer ALU operations
- the FP and integer multiplier
- the FP adder, which handles addition, subtraction and data type conversions
- the FP and integer divider.

Assuming these functional units are not pipelined, no new instruction using a specific functional unit may be issued if a previous instruction using the same unit is still in operation, or which is the same, has not yet left the EX stage. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline up to this point will be stalled.

Multicycle operations in classical 5-stage pipeline - 3

The classical 5-stage pipeline with three additional non-pipelined FP functional units

Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 4

The structure of the whole pipeline may be generalized to allow pipelining of some FP functional units and enable multiple ongoing operations. To ease the way the description is made, two definitions are introduced for the EX functional units

- *latency* – is the number of intervening clock cycles between an instruction that produces a result and an instruction that uses the result
- *initiation or repetition interval* – is the number of clock cycles that must elapse between issuing two operations of the same type.

Integer ALU operations have a latency of zero since the results can be used on the next clock cycle. *Loads*, on the other hand, have a latency of one since the results can be used after one intervening clock cycle. Furthermore, since most operations consume their operands at the beginning of the EX stage, latency is usually the number of stages after EX that an instruction needs to produce the result: zero stages for the integer ALU operations and one stage for the load operations. The exception is *stores* where the latency is minus one.

Multicycle operations in classical 5-stage pipeline - 5

Latencies and repetition intervals for the operations in the functional units

Source: Computer Architecture: A Quantitative Approach

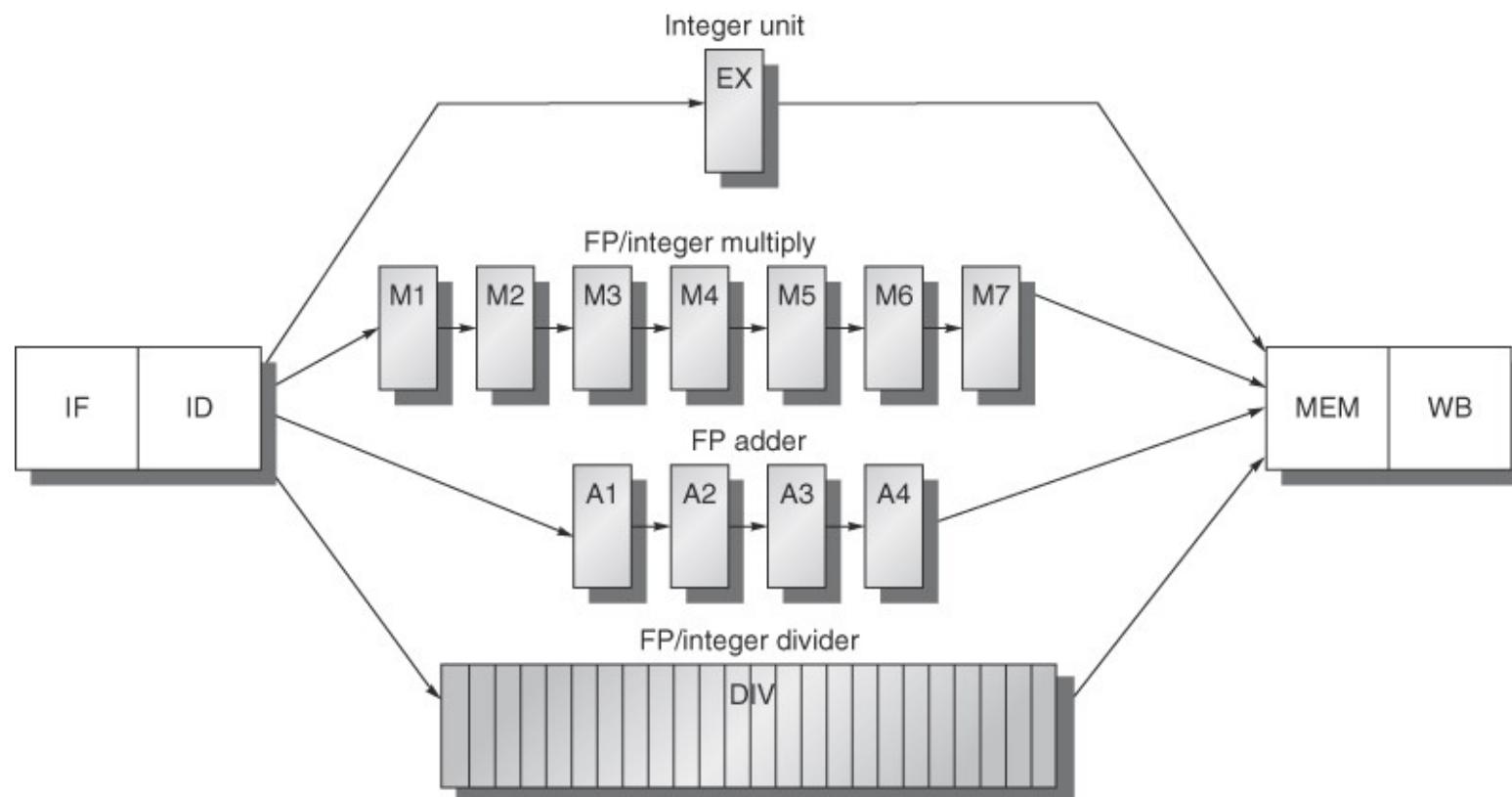
Functional Unit	Latency	Repetition Interval
Integer ALU	0	1
Data memory (Integer and FP loads)	1	1
FP addition	3	1
FP multiplication	6	1
Integer multiplication		
FP division	24	25
Integer division		

Notice that only the FP addition and the FP multiplication / Integer multiplication units are pipelined, The FP division / Integer division unit, on the other hand, is not pipelined and requires 24 clock cycles to produce a result.

Multicycle operations in classical 5-stage pipeline - 6

The classical 5-stage pipeline supporting multiple outstanding FP operations

Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 7

The following observations are in order

- since the FP division / Integer division unit is not pipelined, *structural hazards* can occur; they have to be detected and the issuing instructions needing to use the unit have to be stalled
- since now not all instructions have the same execution time, the number of register writes in the same clock cycle may be larger than one
- *data hazards*, where an instruction tries to write a register before it has been written by another instruction, are possible – which implies that the non-pipelined execution order is no longer maintained
- the fact that instructions can complete in a different order from the one they were issued, will give rise to problems when dealing with exceptions
- *data hazards*, where an instruction tries to read a register before it has been written by another instruction, will be more frequent.

Multicycle operations in classical 5-stage pipeline - 8

Data hazards, where an instruction tries to read a register before it has been written by another instruction, follow a pattern that is fundamentally the same found for the integer pipeline.

<i>Instruction</i>	<i>Clock number</i>																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4, 0 (R2)	IF	ID	EX	MEM	WB												
MUL.D F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2, 0 (R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Notice that the *store* instruction has been stalled an extra cycle to prevent the structural hazard that would arise from the access conflict to the MEM stage. However, since only one of the instructions accesses data memory, they can be made to proceed to the MEM stage in the same clock cycle, if extra hardware is added.

Multicycle operations in classical 5-stage pipeline - 9

If one assumes that the FP register bank has a single write port, sequences of FP operations, as well as a FP *load* instruction together with other FP operations, can give rise to access conflicts to the register write port.

Instruction	Clock number										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
. . .		IF	ID	EX	MEM	WB					
. . .			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
. . .					IF	ID	EX	MEM	WB		
. . .						IF	ID	EX	MEM	WB	
L.D F0,0(R2)							IF	ID	EX	MEM	WB

In clock cycle 11, all three instructions reach the WB stage and need to write to the FP register bank. With a single write port, the processor must serialize instruction completion. The number of write ports could be increased to solve the problem, but this approach may not be very attractive since the extra write ports may be seldom used.

Multicycle operations in classical 5-stage pipeline - 10

Instead, one can choose to detect the structural hazard and to schedule the access to the write port.

One solution is to track the use of the write port at the ID stage and to stall, if it is necessary, the current instruction before it issues, just as it is done for any other structural hazard. The implementation of this approach needs a shift register, called the *reservation register*, whose contents moves in the opposite direction to the instruction flow in the pipeline at every clock cycle. Its purpose is to signal the cycles where instructions already issued will write at the FP register bank. Thus, when the instruction at the ID stage will have to write to a register of the FP register bank, the corresponding stage of the shift register is checked. If the bit is set, meaning that another instruction already issued will also be doing it in that clock cycle, the instruction is stalled. Otherwise, that bit of the shift register is set and the instruction progresses.

This approach has the advantage of keeping all interlock detection and stall insertion restricted to the ID stage. The cost is the extra shift register and all the logic for determining the potential write conflict.

Multicycle operations in classical 5-stage pipeline - 11

Another solution is to stall a conflicting instruction as it tries to enter either the MEM or the WB stage. Any of the conflicting instructions may be chosen to stall. A simple, though sometimes suboptimal, heuristic is to give priority to the instruction exiting the unit with the highest latency, since it is the one most likely to have caused another instruction to be stalled.

This approach has the advantage of postponing the conflict detection until the moment where it becomes trivial to assert it. The disadvantage is that it makes pipeline control more complex as stalls can now arise from two places.

Multicycle operations in classical 5-stage pipeline - 12

Data hazards, where an instruction tries to write to a register before it has been written by another instruction, are apparently not dramatic and could be discarded. The rationale is that they should never occur in a well written code sequence because no compiler would generate two writes to the same register without an intervening read. However, if the sequence is unexpected, they can indeed occur and produce wrong results.

```
BNZ      R1, foo  
DIV.D   F0, F2, F4  
. . .  
foo:   L.D    F0, 0 (R3)
```

If the branch is taken (assuming a delayed branch scheme), the `L.D` instruction will reach the WB stage before the `DIV.D` instruction can complete. Thus, originating an inconsistency in the program state from this point on.

Multicycle operations in classical 5-stage pipeline - 13

There are two possible ways to handle this hazard. The first approach is to delay the issue of the second writing instruction until the first enters the MEM stage. The second is to stamp out the first instruction by detecting the hazard and disabling its ability to change the register – the second instruction can then be immediately issued.

Due to its rarity to appear in the code, either approach is acceptable.

Multicycle operations in classical 5-stage pipeline - 14

A critical problem caused by long running instructions is illustrated by the code sequence below

DIV.D	F0, F2, F4
ADD.D	F10, F10, F8
SUB.D	F12, F12, F14

Although there are no data dependancies, the first instruction will complete after the next two. A situation that is known as *out of order completion*.

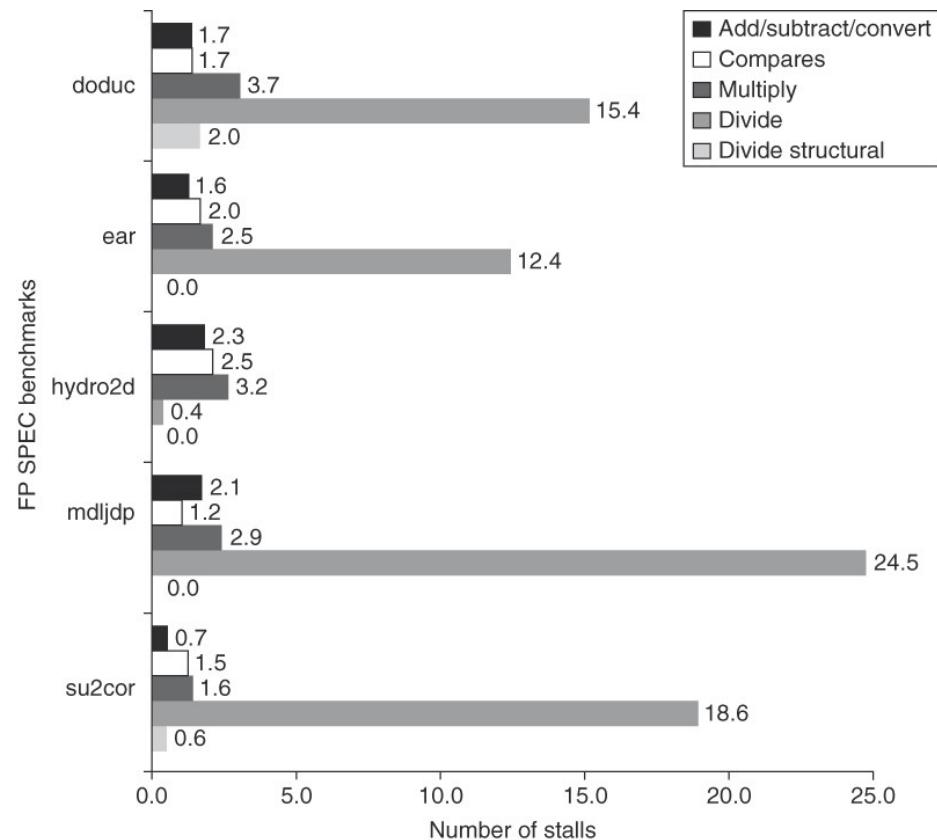
This condition may lead to *imprecise exceptions*. To understand how, consider that after the completion of the ADD and the SUB instructions, the DIV instruction will generate an exception. Since both ADD and SUB have changed each one of their operands, the program state has been modified and cannot be recovered, not even with software help. So restarting instruction DIV is not possible!

There are ways to deal with this problem, but they will not be discussed here.

Multicycle operations in classical 5-stage pipeline - 15

Stalls per FP operation for each major type of FP operation for SPEC89 FP benchmark

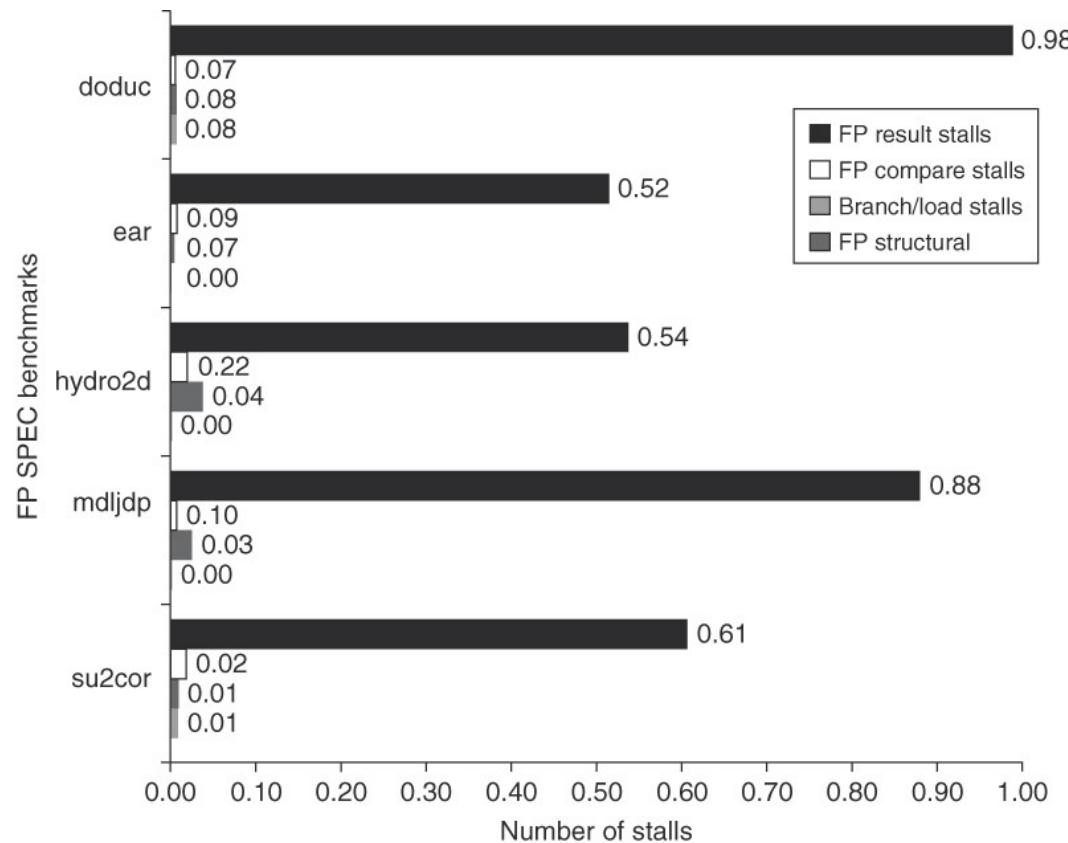
Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 16

Stalls per instruction for the MIPS FP pipeline for SPEC89 FP benchmark

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 1

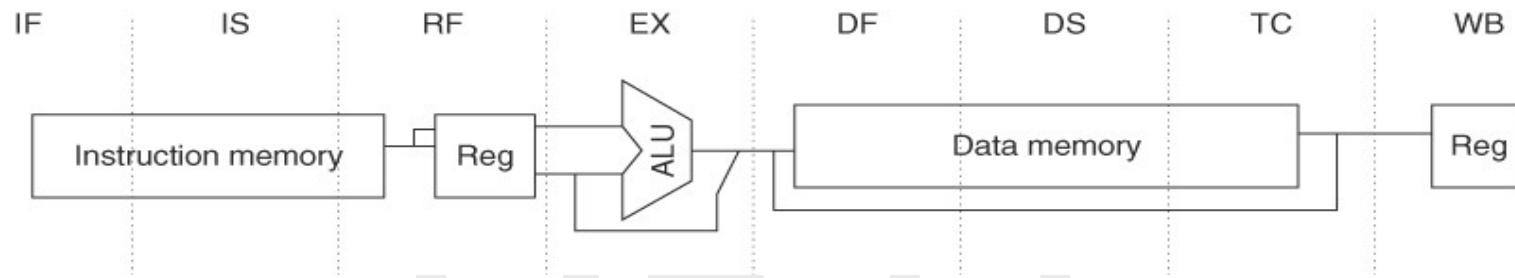
The MIPS R4000 processor family implements MIPS64 instruction set, but implements a deeper pipeline than the classical 5-stage pipeline that has been described, both for integer and FP programs.

The deeper pipeline allows the processor to achieve a higher clock rate by making a decomposition into eight stages, instead of five. Since memory access, even by the use of caches, is particularly time critical, the extra pipe stages are concerned with memory access decomposition. This type of deeper pipelining is sometimes called *superpipelining*.

MIPS R4000 pipeline - 2

R4000 eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



IF – first half of instruction fetch

IS – second half of instruction fetch

RF – instruction cache hit detection + instruction decode and register fetch + hazard checking

EX – execution, including ALU operation, effective address and branch target computation and condition evaluation

DF – first half of data fetch

DS – second half of data fetch

TC – data cache hit detection

WB – register write back

MIPS R4000 pipeline - 3

Although instruction and data memory accesses occupy multiple clock cycles, they are fully pipelined and, therefore, a new instruction can start on every clock cycle. In fact, as it can be noticed, since cache hit detection takes place on the last stage of memory access, the pipeline tries to use data even before hit detection is completed. If a miss occurs, the pipeline is backed up a cycle when the correct data are available.

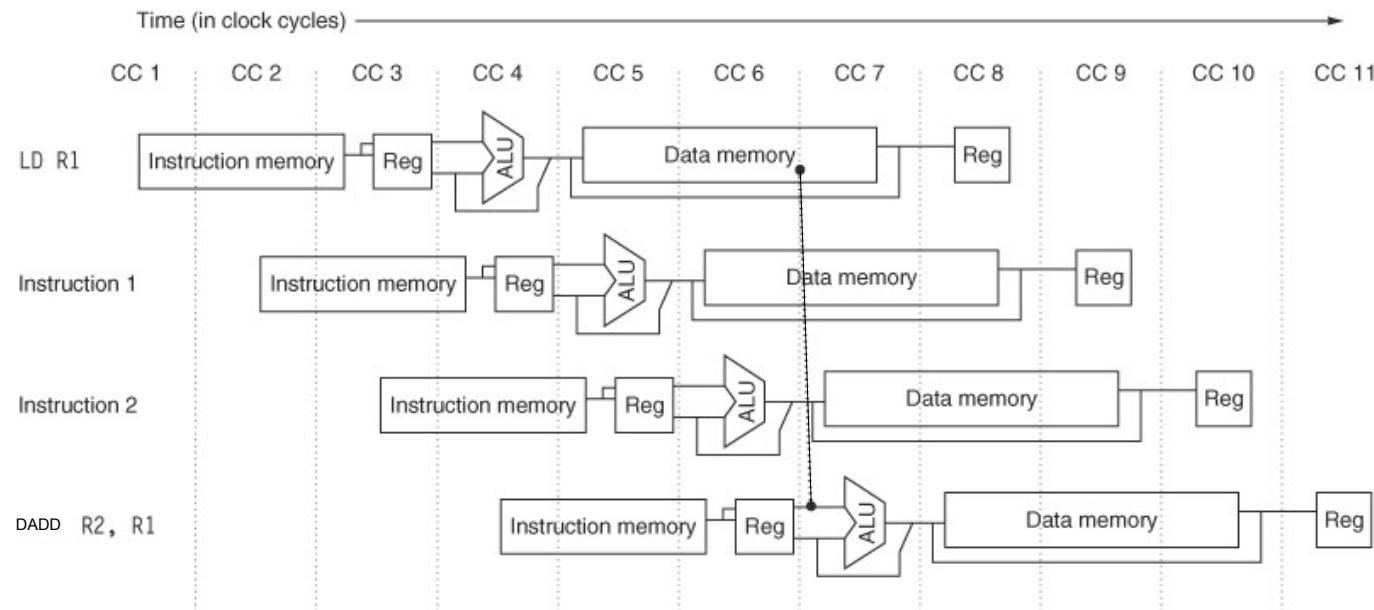
One should notice that this longer latency pipeline not only increases the required *forwarding* logic, but also increases the *load* and *branch* delays. The *load* delay is two cycles long, since data are available at the end of the DS stage. The *branch* delay is 3 cycles long, since the branching condition is computed here during the EX stage. R4000 architecture, however, uses a single cycle *delayed branch* scheme, together with a *predicted-untaken* strategy, which produces no idle cycles, when the branch is not taken, and two idle cycles, when the branch is taken.

Pipeline *interlocks* enforce both the 2-cycle branch stall penalty, when the branch is taken, and any data hazard that arises from the occurrence of a load instruction.

MIPS R4000 pipeline - 4

Two cycle load instruction delay on the eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 5

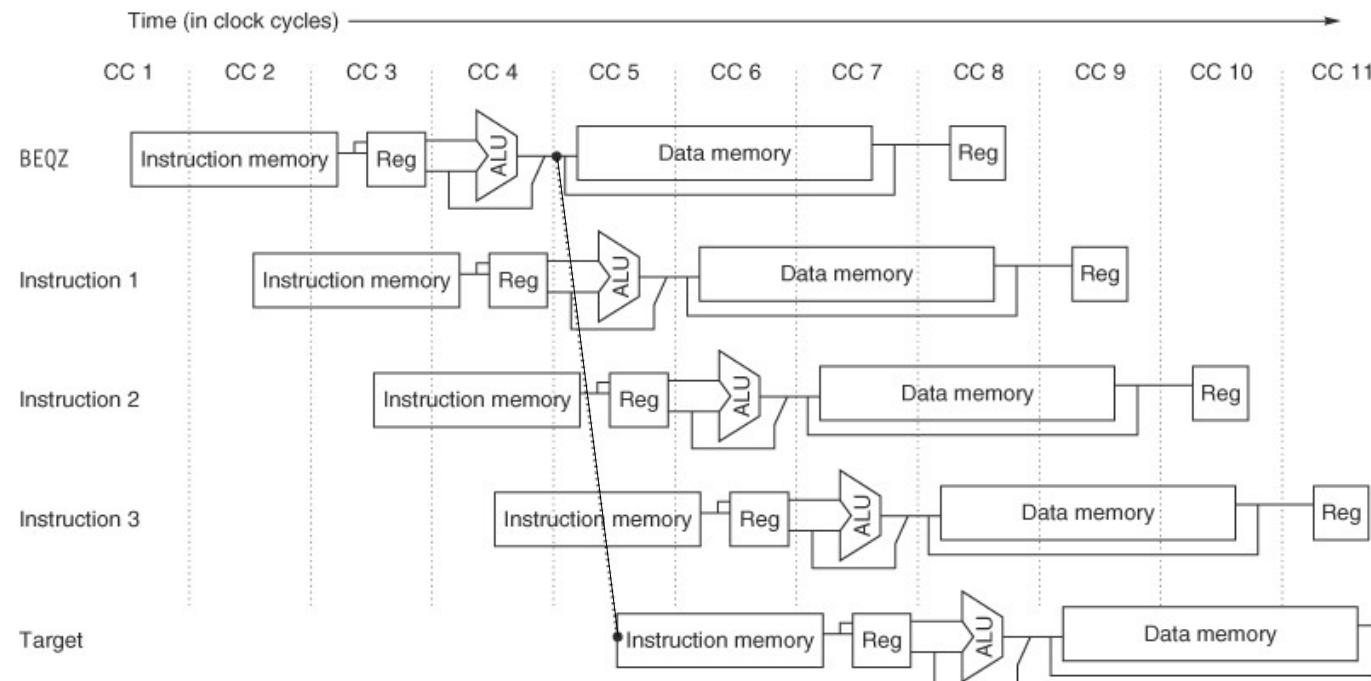
Two cycle stall produced by a load instruction, followed by the immediate use of the loaded value, on the eight-stage pipeline organization

<i>Instruction</i>	<i>Clock number</i>								
	1	2	3	4	5	6	7	8	9
LD R1, 0(R10)	IF	IS	RF	EX	DF	DS	TC	WB	
DAAD R2, R1, R2		IF	IS	RF	stall	stall	EX	DF	DS
DSUB R3, R1, R3			IF	IS	stall	stall	RF	EX	DF
OR R4, R1, R4				IF	stall	stall	IS	RF	EX

MIPS R4000 pipeline - 6

Three cycle basic branch delay on the eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 7

Delayed branch behavior both for the *taken* and *untaken* cases on the eight-stage pipeline organization

<i>Instruction</i>	<i>Clock number</i>								
	1	2	3	4	5	6	7	8	9
branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
delay slot		IF	IS	RF	EX	DF	DS	TC	WB
idle cycle			<i>stall</i>						
idle cycle				<i>stall</i>	<i>stall</i>	<i>stall</i>	<i>stall</i>	<i>stall</i>	<i>stall</i>
branch target					IF	IS	RF	EX	DF

<i>Instruction</i>	<i>Clock number</i>								
	1	2	3	4	5	6	7	8	9
branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
delay slot		IF	IS	RF	EX	DF	DS	TC	WB
branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
branch instruction + 3				IF	IS	RF	EX	DF	DS

MIPS R4000 pipeline - 8

The MIPS R4000 floating-point unit consists of three nominal functional units: a floating-point adder, a floating-point multiplier and a floating-point divider. The adder logic is used in the final stages of a multiply or a divide operation.

Double precision operations may take from 2 cycles (for a negate) to 112 cycles (for a square root).

Each functional unit can be thought of as having up to eight different processing stages. There is a single copy of each of these processing stages and different instructions may use a particular stage zero or more times and combine them in their own order.

MIPS R4000 pipeline - 9

The eight processing stages in the R4000 floating point pipelines

Source: Computer Architecture: A Quantitative Approach

<i>Processing stage</i>	<i>Elemental functional unit</i>	<i>Description</i>
A	FP adder	mantissa ADD stage
D	FP divider	divide pipeline stage
E	FP multiplier	exception test stage
M	FP multiplier	multiplier first stage
N	FP multiplier	multiplier second stage
R	FP adder	rounding stage
S	FP adder	operand shift stage
U		unpack floating point numbers

MIPS R4000 pipeline - 10

Latencies, repetition intervals and stage composition for floating point operations in R4000

Source: Computer Architecture: A Quantitative Approach

FP instruction	Latency	Repetition interval	Processing Stages
add – subtract	4	3	U – S+A– A+R – R+S
multiply	8	4	U – E+M – M – M – M – N – N+A– R
divide	36	35	U – A– R – D ²⁸ – D+A – D+R – D+A – D+R – A– R
square root	112	111	U – E – (A+R) ¹⁰⁸ – A– R
negate	2	1	U – S
absolute value	2	1	U – S
compare	3	2	U – A– R

MIPS R4000 pipeline - 11

Effect of a FP multiply instruction issued at clock cycle 0 on a FP add instruction issued between clock cycles 1 to 7

Source: Adapted from Computer Architecture: A Quantitative Approach

<i>Instruction</i>	<i>Decision</i>	<i>Clock number</i>										
		<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>
multiply	issue	U	E+M	M	M	M	N	N+A	R			
add	issue		U	S+A	A+R	R+S						
	issue			U	S+A	A+R	R+S					
	issue				U	S+A	A+R	R+S				
	stall					stall	stall	U	S+A	A+R	R+S	
	stall						stall	U	S+A	A+R	R+S	
	issue							U	S+A	A+R	R+S	
	issue								U	S+A	A+R	R+S

MIPS R4000 pipeline - 12

Effect of a FP divide instruction issued at clock cycle 0 on a FP add instruction issued between clock cycles 26 to 36

Source: Adapted from Computer Architecture: A Quantitative Approach

<i>Instruction</i>	<i>Decision</i>	<i>Clock number</i>										
		26	27	28	29	30	31	32	33	34	35	36
divide	issue at cc 0	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R
add	issue		U	S+A	A+R	R+S						
	issue		U	S+A	A+R	R+S						
	stall			stall	stall	stall	stall	stall	stall	U	S+A	
	stall				stall	stall	stall	stall	stall	U	S+A	
	stall					stall	stall	stall	stall	U	S+A	
	stall						stall	stall	stall	U	S+A	
	stall							stall	stall	U	S+A	
	issue									U	S+A	
	issue											U

MIPS R4000 pipeline - 13

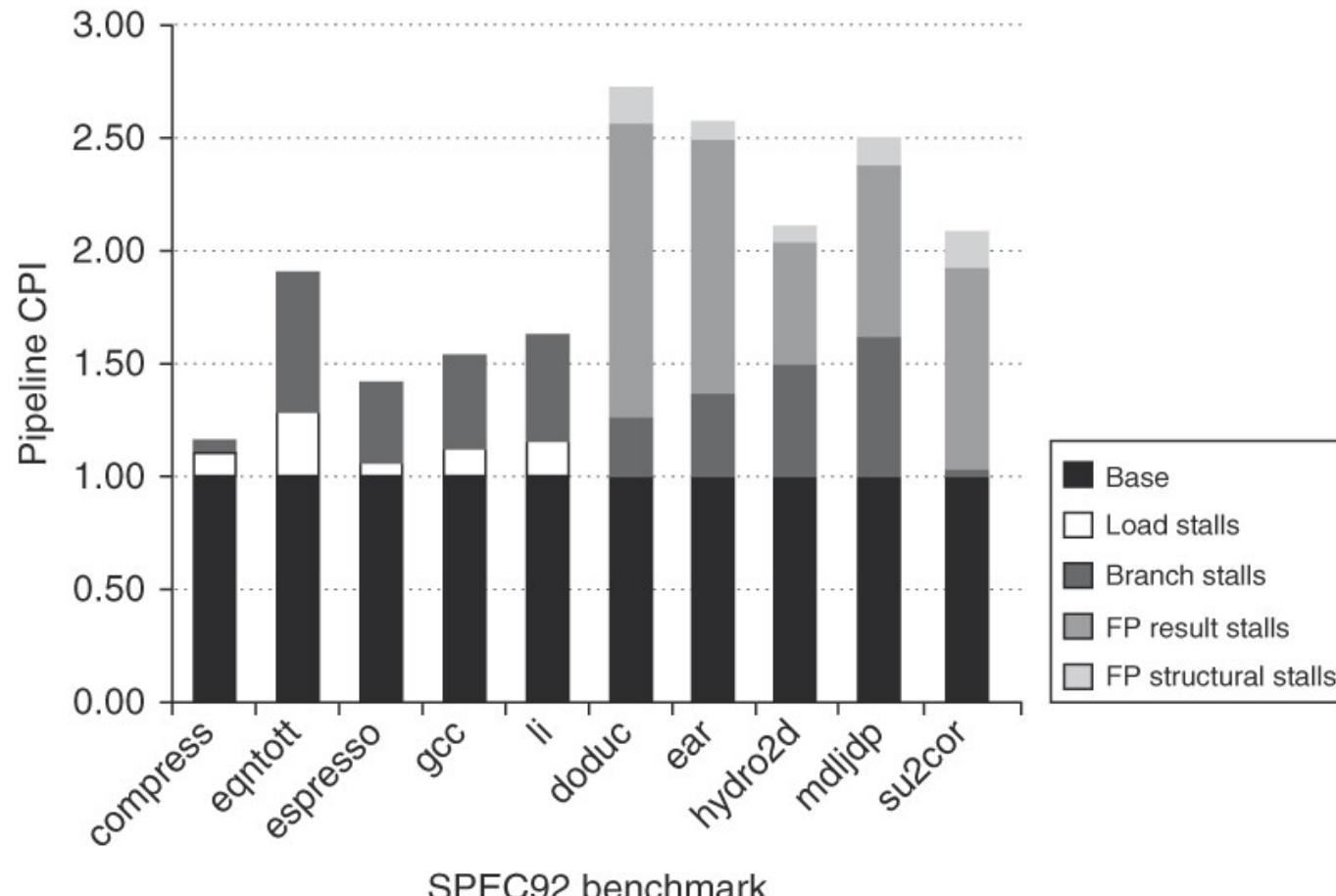
There are four major causes for pipeline stalls or losses which prevent that instruction issuing at the nominal rate be attained

- *load stalls* – delays arising from the use of the load value one or two clock cycles after load execution
- *branch stalls and losses* – two clock cycle delays after every *taken* branch and one clock cycle delay if the branch delay slot cannot be filled with an useful instruction
- *FP result stalls* – delays arising because an operand is required and is not yet computed
- *FP structural stalls* – delays arising because the required processing stages in the FP pipeline are not available when needed.

MIPS R4000 pipeline - 14

The pipeline CPI for 10 of the SPEC92 benchmarks assuming a perfect cache

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 15

The R4000 pipeline has much longer branch delays than the classical 5-stage pipeline. The longer branch delay substantially increases the clock cycles spent on branches, specially for integer programs with a high branch frequency.

An interesting effect for FP programs is that the latency of the FP functional units leads to more result stalls than those produced by the structural hazards, which are due both to the repetition interval limitations and to conflicts from the use of specific processing stages in different FP instructions.

Thus, reducing the latency of FP operations should be the first task to be taken care of on an optimization effort, rather than increasing pipelining or replicating the functional units processing stages.

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
 - Appendix A: *Instruction Set Principles* (Sections 1 to 8)
 - Appendix C: *Pipelining: Basic and Intermediate Concepts* (Sections 1 to 7)
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
 - Chapter 12: *Instruction sets: Characteristics and Functions*
 - Chapter 13: *Instruction sets: Addressing Modes and Formats*
 - Chapter 14: *Processor Structure and Function*
 - Chapter 15: *Reduced Instruction Set Computers*

universidade de aveiro



Arquitecturas de Alto Desempenho

Memory Hierarchy Design

António Rui Borges

Summary

- *Why memory management is so important*
 - Principle of locality
 - Memory hierarchy
- *Cache*
 - Cache principles
 - Cache performance
 - Cache optimization
- *Main memory*
 - Asynchronous dynamic RAMs
 - Synchronous dynamic RAMs
- *Suggested reading*

Why memory management is so important - 1

Since the early days of the computing era, programmers dream about *unlimited* amounts of *fast* memory to store and run their programs, that is, they dream about having as much memory as they deem necessary to store code and data which may then be accessed at the maximum rate the processor can operate.

Although *main memory* capacity has increased steadily over the years, the fact is programs tend to expand in size and complexity filling all the space that is available – *Parkinson Law*.

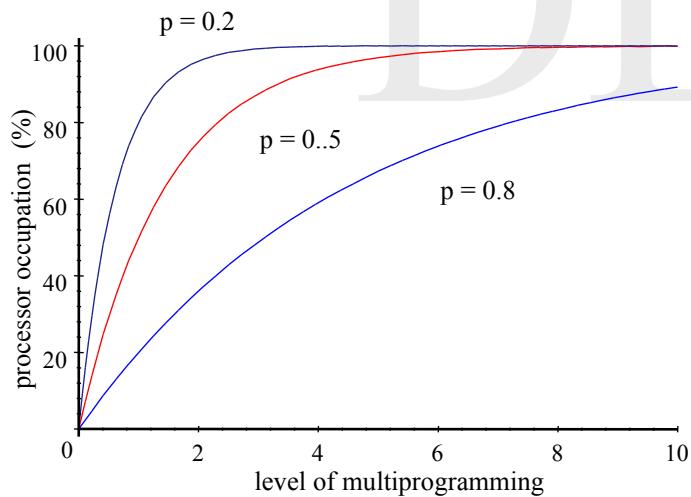
This is true not only because programs have become more complex as the performance of computer systems improves, dealing with problems that could not have been previously tackled, but also because it is critical in a multitasking environment to store together in main memory the addressing spaces of many processes so that the processor is kept fully occupied and the associated *response* and/or *turn-around* times are minimized.

Why memory management is so important - 2

$$\text{fraction of processor occupation} = 1 - p^n \quad (\text{simplified model})$$

p - fraction of the time a process waits blocked for the completeness of I/O operations, synchronization or any other cause

n - number of processes which currently coexist in main memory



N. of processes in MP	% of occupation (P)
4	59
8	83
12	93
16	97

$$p = 0,8$$

Principle of locality - I

It is not possible, however, to satisfy literally programmers' dream.

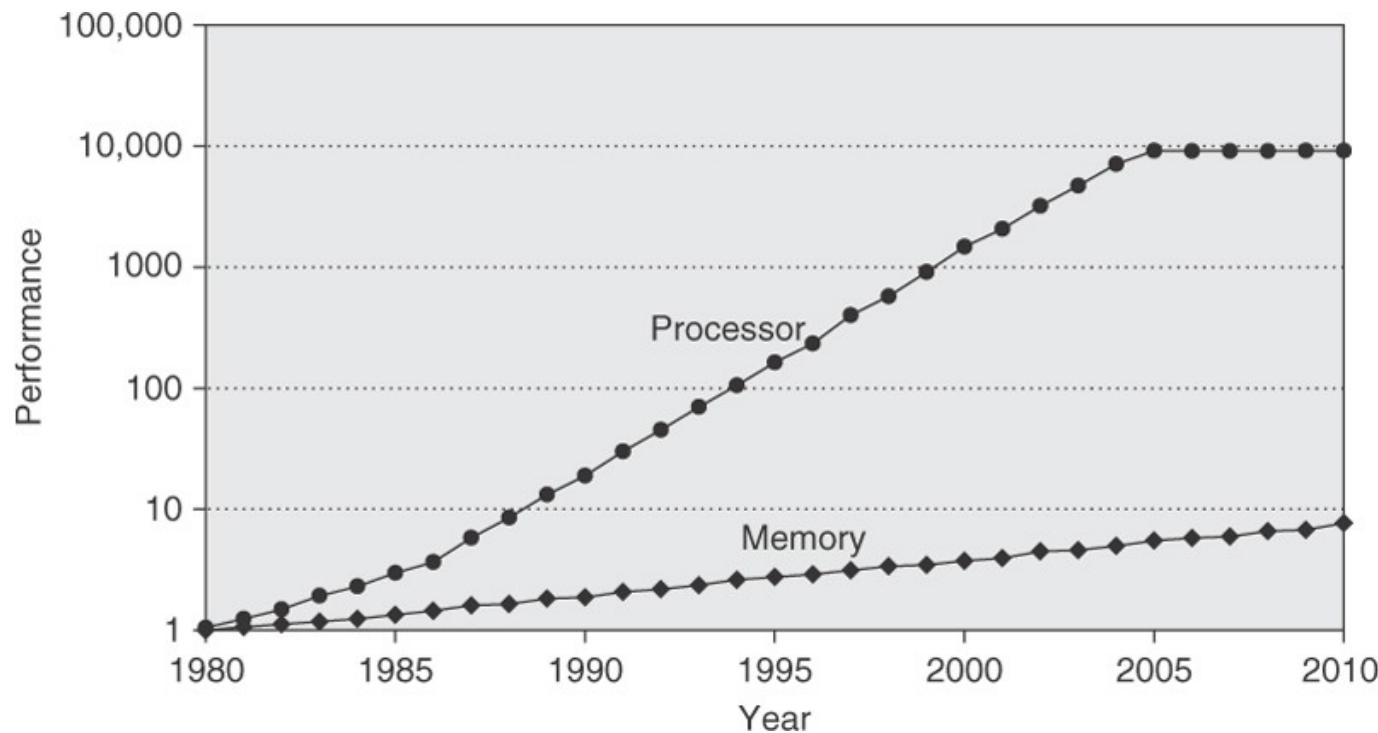
It can be shown that, for a given implementation technology and a given power budget, simpler hardware can be made faster. As complexity grows, signal propagation time delays become longer due to the increase of the required amount of interconnection circuitry and the decrease of the intensity of the driving electrical currents.

Futhermore, the problem has turned more severe as time went by. Instead of being reduced, the processor-DRAM performance gap has augmented gradually over the past decades and has even worsened since the introduction of multicore processors, where the aggregate peak bandwidth is proportional to the number of processors in the core.

Principle of locality - 2

Over time performance variation of single processor vs. memory

Source: Computer Architecture: A Quantitative Approach



processor curve – average number of memory requests per second

memory curve – inverse of DRAM access latency

Principle of locality - 3

To deal with this problem, the approach followed by memory systems designers is based on an observational fact derived from the tracing of program execution and known as the *principle of locality*. It simply states that, for relatively large periods of time, a program tends to reference a very definite fraction of its addressing space. Thus, one may speak of

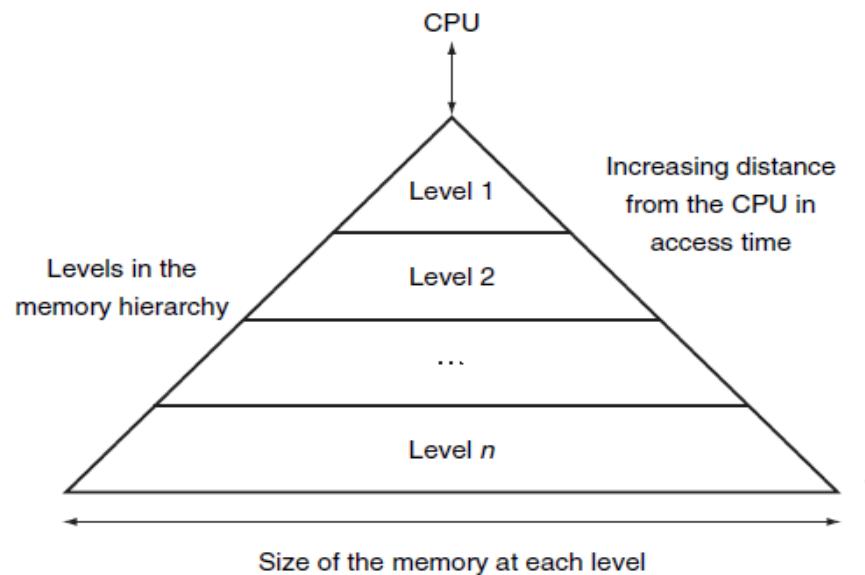
- *spatial locality* – when a memory word is referenced once, another one, which is stored nearby, may be referenced soon
- *temporal locality* – when a memory word is referenced once, it may be referenced again soon.

It is not too difficult to see why programs have this kind of behavior. Try to assert the reasons why it is so!

Memory hierarchy - 1

To take advantage of the principle of locality, computer memory is implemented as a memory *hierarchy*, that is, it consists of multiple levels of memory with different sizes and access speeds. As a rule, the faster memory modules have a higher price per bit than the slower ones and a smaller storage capacity.

The goal is to present the programmer with as much memory as it is available in the cheapest technology (lowest level), while providing access at the speed offered by the most expensive one (highest level).



Source: Computer Organization and Design: The Hardware/Software Interface

Memory hierarchy - 2

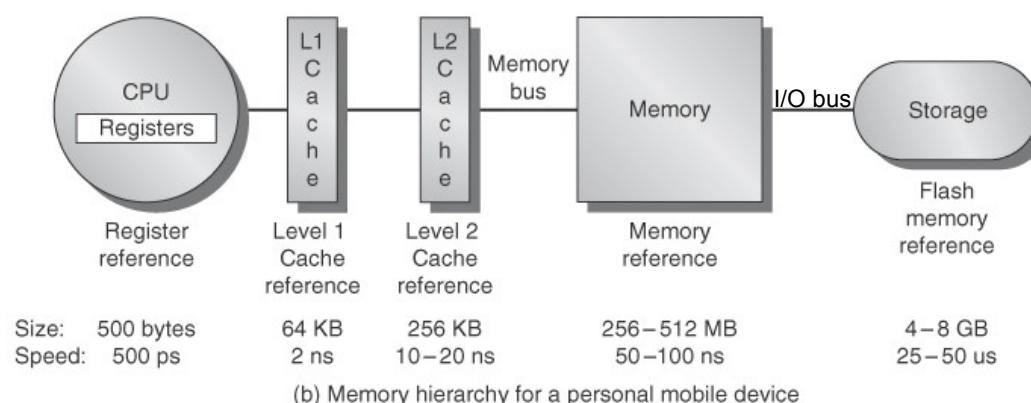
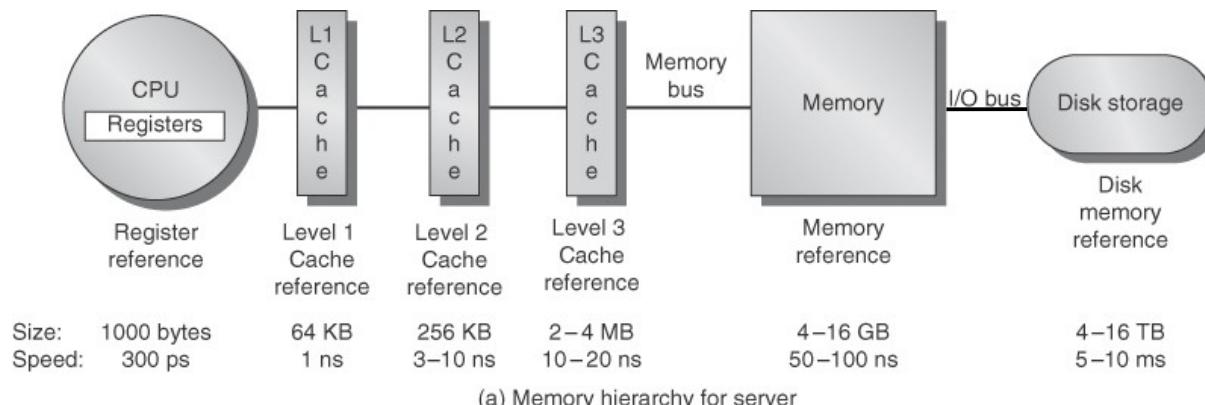
Memory hierarchy may be divided into distinct levels

- *register bank* – internal memory to the processor, access is controlled by the instructions
- *cache* – external memory to the processor, access is controlled by *instruction fetch* and *data transfer* instructions; it presently consists of several levels of static RAM, all of them placed inside the processor integrated circuit; the first level is even located inside the processor chip and is split into instruction and data units
- *main memory* – external memory to the processor; it implements the *stored-program concept* defining the device where instructions and data of an executing program are mostly stored; it consists of dynamic RAM
- *swapping area* – non-volatile memory located in mass storage; it works as an extension of the main memory to implement an operating system controlled memory organization, usually a *virtual memory paged-architecture*; it consists mostly of HDD or flash memory devices.

Memory hierarchy - 3

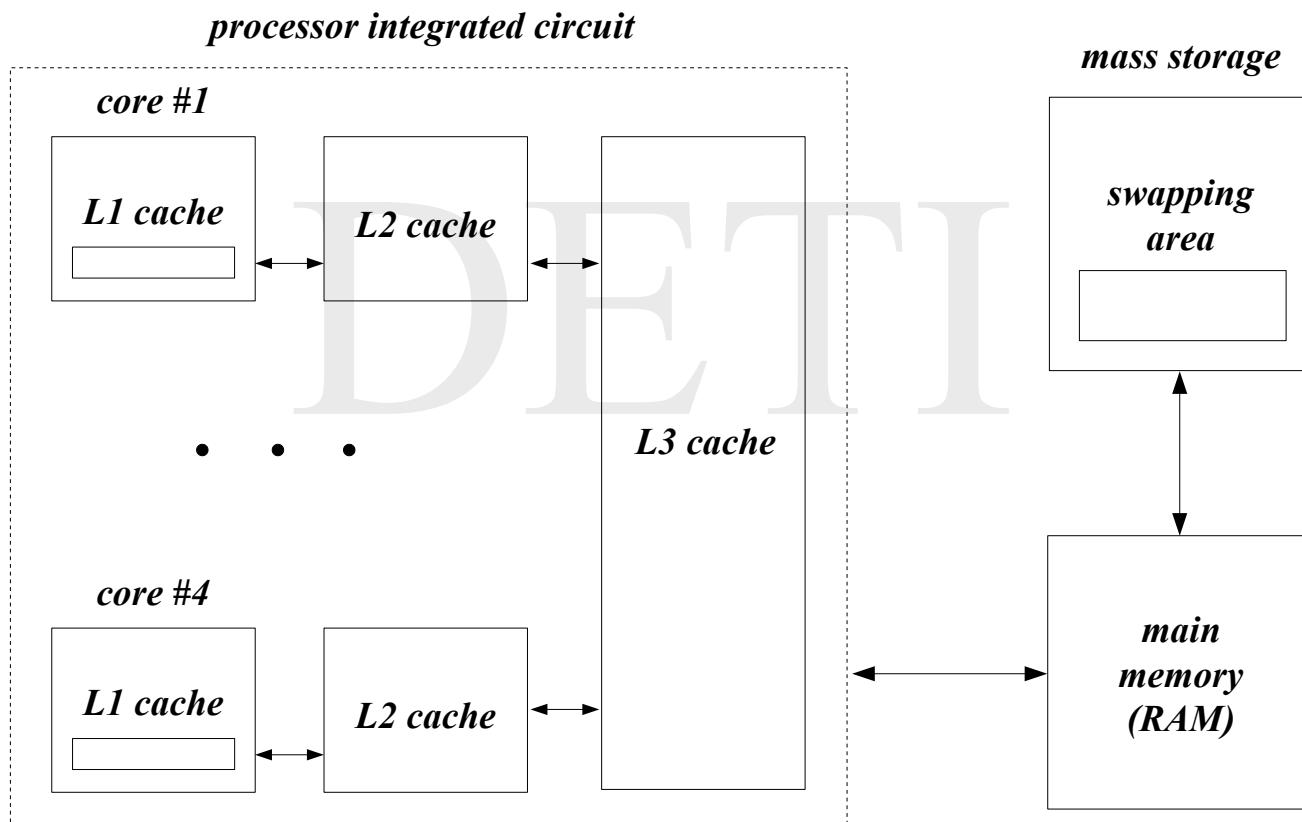
Levels of a typical memory hierarchy

Source: Computer Architecture: A Quantitative Approach



Memory hierarchy - 4

Typical memory hierarchy for a 4-core processor system



Memory hierarchy - 5

Technical specifications of a typical memory hierarchy

Source: adapted from Computer Architecture: A Quantitative Approach

<i>Name</i>	<i>Typical size</i>	<i>Implementation technology</i>	<i>Access time (ns)</i>	<i>Bandwidth (MB/s)</i>	<i>Managed by</i>	<i>Backed by</i>
register bank	less 1KB	multiport custom design – CMOS	0,15 – 0,30	10^5 – 10^6	compiler	cache
cache	32 KB – 8 MB	on/off-chip CMOS SRAM	0,5 – 15	10^4 – 4×10^4	hardware	main memory
main memory	less 512 KB	CMOS DRAM	30 – 200	5×10^3 – 2×10^4	operating system	swapping area
swapping area	greater 1 TB	semiconductor / magnetic	3×10^4 - 5×10^6	50 – 500	operating system	long term storage

Memory hierarchy - 6

In a memory hierarchy, the higher a level is, the closer to the processor it is located. Memory hierarchies take advantage of *temporal locality*, by keeping more recently accessed instructions and data closer to the processor, and of *spatial locality*, by moving blocks consisting of multiple contiguous memory words to higher levels of the hierarchy.

In most systems, memory constitutes a true hierarchy, that is, in order for data to be present at level i , it must be present first at level $i+1$, and all data is present at the lowest level.

The underlying concepts to building memory systems affect many other aspects beyond computer architecture. These include how the operating system manages memory and I/O, how compilers generate code and even how applications use the computer resources.

Because all programs spend much of their time accessing memory, the memory subsystem is a determinant factor for performance. Therefore, programmers need nowadays to understand that memory is hierachic to improve the performance of their programs.

Memory hierarchy - 7

Although a memory hierarchy may consist of several levels, data transfer usually takes place only between two adjacent levels at a time, so one can concentrate on what happens between any pair of levels to get a general view of operations.

The minimum amount of data that can either be present or missing in a two-level hierarchy is called a *block*. We say that a *hit* occurs when the data requested by the processor appears in some block at the upper level; otherwise, the request is called a *miss* and the lower level is then accessed to retrieve the block containing the requested data.

The *hit rate* or *hit ratio* is the fraction of memory references to data found in the upper level over all memory references. On the other hand, the *miss rate* or *miss ratio* is its complement to 1.

Since performance is the major issue, the time required to service hits and misses is relevant. The *hit time*, being the time to access the upper level, comprises also the time to assert whether the access is a hit or a miss. The *miss penalty*, then, is the time to replace a block at the upper level by the block containing the data requested by the processor.

Cache principles - 1

Cache is the name traditionally given to the level(s) of memory hierarchy located between the processor and the main memory. Nowadays, however, the term has a wider meaning: it refers to any storage device which is managed in a manner that takes advantage of the principle of locality.

When dealing with the cache, the term *line* is specifically used to refer to the minimum amount of information that is transferred between any pair of cache levels or stored at the lowest cache level. *Block* is reserved to refer to the data itself either stored in a cache line or in main memory.

Assuming a single level cache, the answer to the following questions will help to enlighten the way how a cache works

- where is a line located in the cache? (*line placement*)
- how is it found if it were present there? (*line identification*)
- which line should be changed on a miss? (*line replacement*)
- what happens in a write operation? (*write strategy*).

Cache principles - 2

Since main memory capacity is much larger than cache size, many memory blocks will overlap at the same location within the cache over time. There are several ways for doing this, but one should bear in mind that the major goals are to keep hardware simple and the whole storing/accessing procedure efficient.

In this sense, the block size should not be completely arbitrary. It is important that the number of stored bytes be a power of 2 so that a *memory address* may be trivially split into a *block address* and an *offset*.

In general, cache can be organized as

- *direct mapped* – when there is a single place where a *block* may be placed
- *fully associative* – when a *block* can be placed anywhere
- *set associative* – when there are an aggregate of places, called a *set*, where a *block* may be placed.

Direct mapped to fully associative organizations form a continuum of levels of set associativity: the set size for *direct mapping* is one and for *fully associativity* is equal to the number of lines in the cache.

Cache principles - 3

memory address

block address (s bits)	offset (w bits)
------------------------	-----------------

Main memory / cache characterization

address length = $s + w$ bits

number of addressable units in main memory = 2^{s+w} bytes

number of blocks in main memory = 2^s

number of lines in the cache = $m = 2^r$

cache size = 2^{r+w} bytes

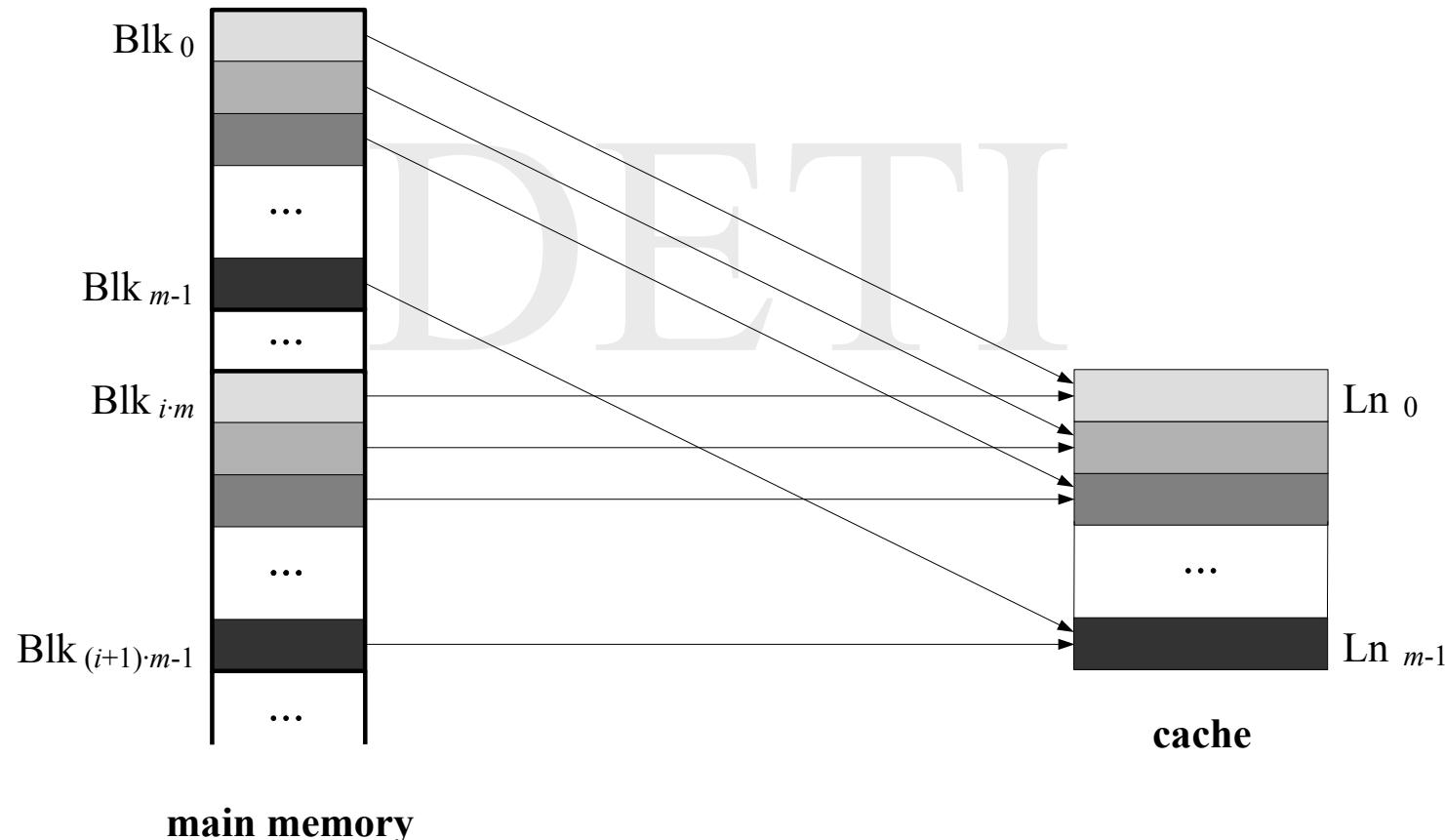
number of sets in the cache = $v = 2^u$

number of lines per set = $k = m / v = 2^{r-u}$

Cache principles - 4

Direct mapping

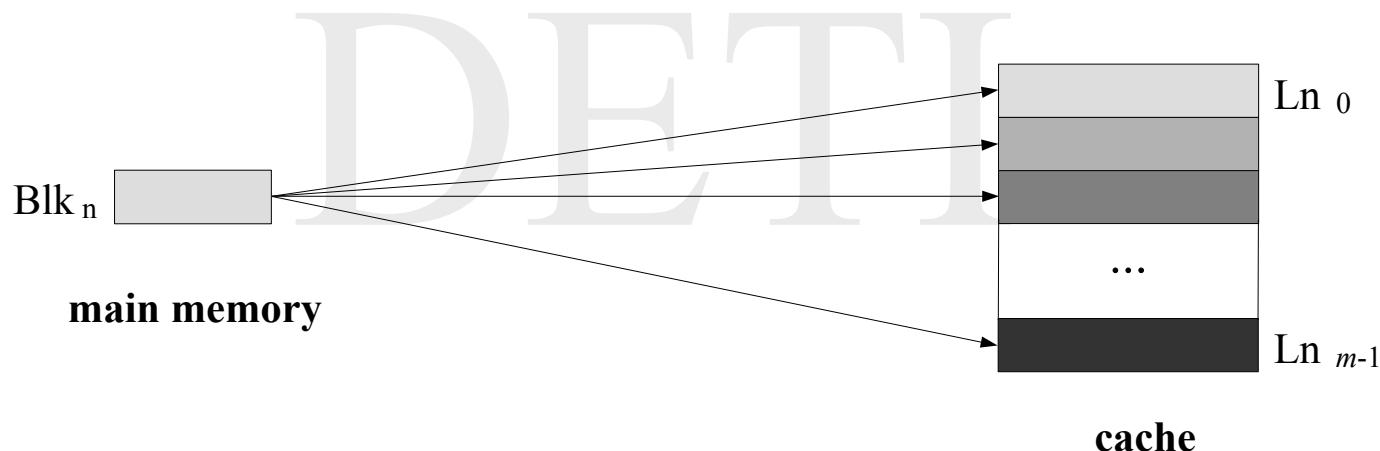
cache line address = block address mod number of lines in the cache



Cache principles - 5

Fully associativity

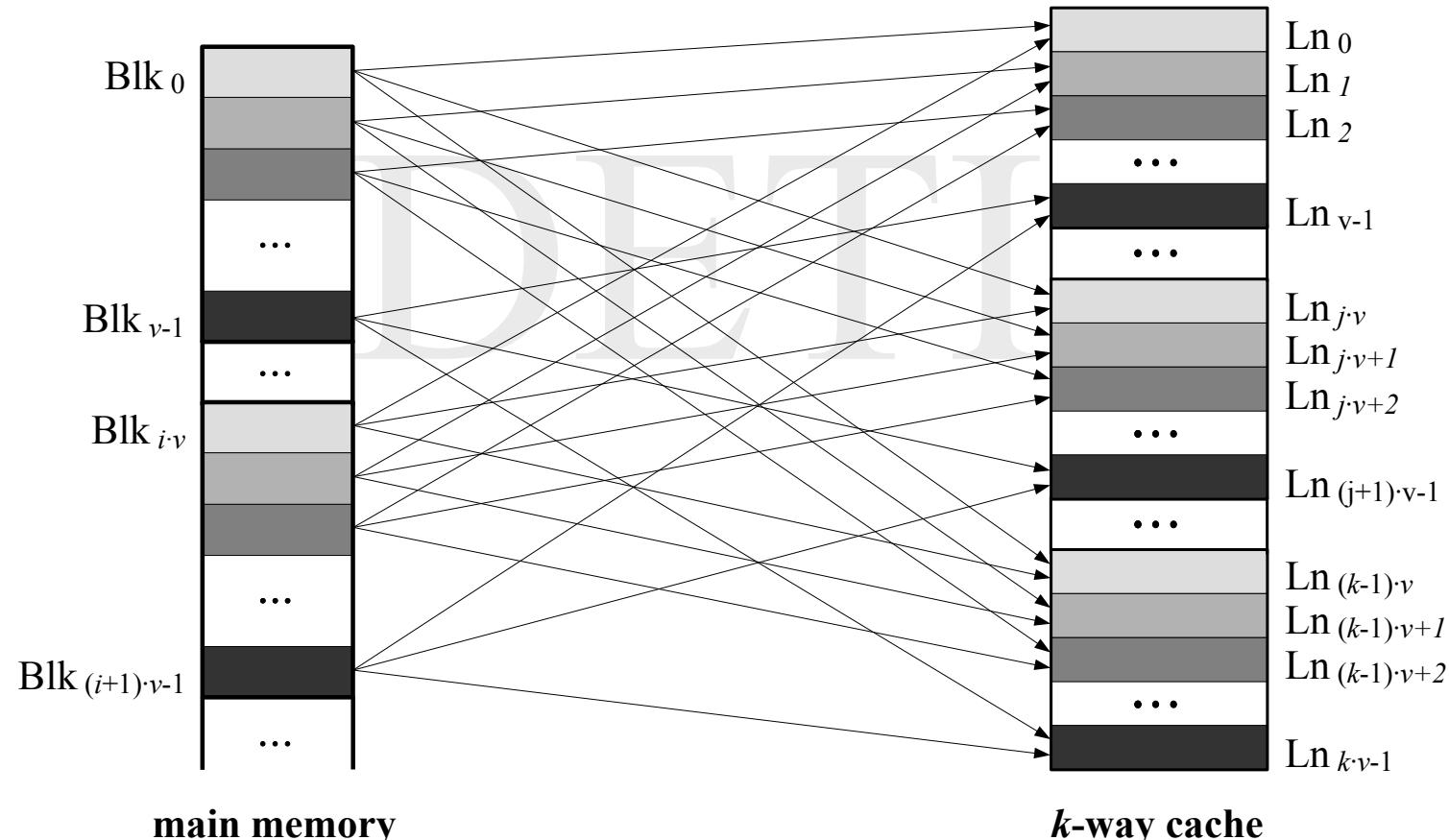
cache line address = any



Cache principles - 6

Set associativity

cache line address = block address mod number of sets in the cache



Cache principles - 7

Besides the contents of a main memory block, a cache line must also contain at least part of its address, usually called the block address *tag field*. The *tag field* is stored in every line of the cache so that it can be checked against the corresponding part of the memory address generated by the processor to determine if there is a match when an access occurs. As a rule, all possible *tags* are checked in parallel to make the search fast. Furthermore, since a data register is never empty, an extra bit (called the *validation bit*) is required to assert if the data presently stored in the line is meaningful or not.

Whenever the *tag field* of a block address is not the whole block address, the remaining bits of the address form a second field, the *index field*, whose goal is to select a specific set within the cache.

block address

tag field ($s-u$ bits)	index field (u bits)
-------------------------	-------------------------

Cache principles - 8

Direct mapping

block address



When the cache is organized through *direct mapping*, there is a single line within the cache where a particular memory block may be stored. This means that the number of lines per set is equal to one, the number of sets is equal to the number of lines and the tag field contained in each line has minimal length.

This organization leads to very simple, fast and efficient implementations. The main disadvantage is the risk of *thrashing*: a phenomenon that arises when the fraction of the addressing space referenced by the processor in an extended period of time contains groups of two or more addresses that map into the same cache lines. When this happens, the hit rate degrades quite a lot and program execution becomes rather slow because no benefit is taken from locality of reference.

Cache principles - 9

Fully associativity

block address

tag field (s bits)

When the cache is organized through *fully associativity*, all the lines within the cache are available for the storage of a particular memory block. This means that the number of lines per set is equal to the number of lines in the cache, the number of sets is equal to one and the tag field contained in each line has maximal length (the index field does not exist).

This organization leads to the minimization of the miss rate since in principle a specific memory block may be stored in any of the lines of the cache. The main disadvantage is the design complexity that it entails and, because of that, for a given implementation technology and a given power budget, it limits the speed of taking a decision for a hit or a miss.

Cache principles - 10

Set associativity

block address

tag field ($s-u$ bits)	index field (u bits)
-------------------------	-------------------------

When the cache is organized through *set associativity* (k -way cache), there are exactly k lines within the cache where a particular memory block may be stored. This means that the number of lines per set is equal to k and the number of sets is equal to v .

This organization tries to attain *the best of the world* as portrayed by the two previous organizations. It leads to not too complex implementations that are still fast and efficient and avoids the risk of *trashing* by providing some redundancy to where a memory block may be stored.

Cache principles - 11

When a miss occurs, the cache controller must select a line whose block will be replaced with the desired data. With *direct mapped* organization, the problem is trivial since only one line is checked for a hit and only this line contents can be modified. With fully associative and set associative organizations, there are in principle many, or at least some, lines to be considered. If the *validation bit* for any of these lines is reset, one of them may be selected, but after some time all of them will contain valid data and so a decision must be taken.

The obvious strategy, the one that minimizes the miss rate, is to choose the line within the group whose data will not be referenced anymore or, if it will be, the reference will happen at the farthest distance from the present – the *principle of optimality*. Unfortunately, this rule is not causal, it would require guessing the future and, therefore, can not be implemented in practice.

Cache principles - 12

The main strategies employed for line selection are

- *random* – a pseudo-random generator is used to spread replacement uniformly among candidate lines; this potentiates a reproducible behavior
- *least recently used* (LRU) – in order to approximate the principle of optimality one relies on the past to predict the future; thus, the candidate line is the one which has not been referenced for the longest period of time
- *first in, first out* (FIFO) – because LRU leads to a complex implementation, an approximation to it that is simpler, but still relies on the past to predict the future, is to consider the line whose contents has remained for the longest period of time in the cache.

Cache principles - 13

**Data cache misses per 1000 instructions for the Alpha architecture (DEC)
using 10 SPEC2000 benchmarks (5 SPECint2000 and 5 SPECfp2000)**

Source: Computer Architecture: A Quantitative Approach

Set associativity (block size = 64 bytes)										
Cache size	2-way			4-way			8-way			
	Random	LRU	FIFO	Random	LRU	FIFO	Random	LRU	FIFO	
16 KB	117,3	114,1	115,5	115,1	111,7	113,3	111,8	109,0	110,4	
64 KB	104,3	103,4	103,9	102,3	102,4	103,1	100,5	99,7	100,3	
256 KB	92,1	92,2	92,5	92,1	92,1	92,5	92,1	92,1	92,5	

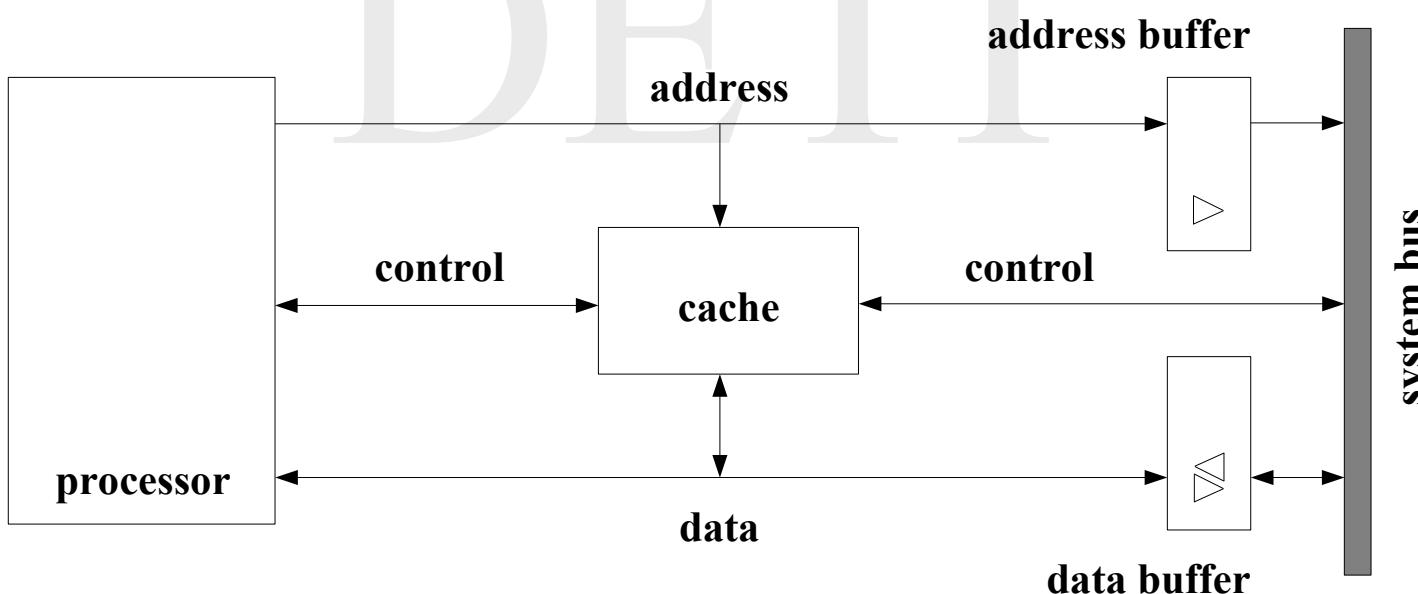
Cache principles - 14

Read operations from memory dominate processor cache accesses. This is so because all *instruction fetches* are read operations and most instructions do not write to memory. In fact, 10 SPEC2000 benchmarks (5 SPECint2000 / 5 SPECfp2000), running on a MIPS processor, suggest an average of 26% / 30% of *load* instructions and of 10% / 9% of *store* instructions over all executed instructions [Computer Architecture: A Quantitative Approach], which leads to a proportion of *read* over *write* operations of 93:7 / 94:6.

Making the *common case fast* means optimizing caches for reads, especially since traditionally processors wait for the completion of reads, but need not wait for the completion of writes. A *read* operation may start as soon as the block address containing the referenced data is made available. At the same time, the tags of candidate lines are compared to the *tag field* of the block address to determine if there is a hit. If so, the requested part of the block is immediately passed on to the processor; otherwise, the block transfer from the lower level is initiated and the read operation is stalled until the transfer is accomplished.

Cache principles - 15

In order to further optimize caches for read operations, contemporary organizations connect the pair processor-cache via address, data and control lines. Address and data lines also connect to address and data buffers that mediate the access to the system bus from which main memory is reached. When a hit occurs, address and data buffers are disabled and all the communication is internal. When a miss occurs, the block address is loaded onto the system bus and data are returned to both the cache and the processor.



Source: Computer Organization and Architecture Designing for Performance

Cache principles - 16

Write operations are a bit different. Modifying the block contents can only start after the tag is checked to determine if there is a hit. Thus, write operations take usually longer than the read operations. Besides, although the processor always specifies the data size and location, only for write operations this is critical because a definite portion of the block contents is changed; for read operations, the access to more data bytes than it is required, is irrelevant.

There are two basic write *policies*

- *write-through* – data are written to both the cache line and to the block in the lower level
- *write-back* – data are written to the cache line; the line block contents is only written to the lower level when the block is replaced.

When write-back policy is implemented, a key feature, known as the block *dirty bit*, is commonly employed to warrant that only modified blocks are transferred back on replacement. When a block is first transferred to the cache, its status bit assumes the value *clean* to signal that its contents has not been changed; when a write occurs, the status bit then assumes the value *dirty* and the block must be written back on replacement.

Cache principles - 17

Write-through advantages

- it is simpler to implement; since all write operations result in a write to the lower level, the cache lines are always clean
- the updated block contents is always present at the lower level, which simplifies the ensurance of data coherency
- it plays an important role in the design of multilevel caches; for the upper levels, the writes need only to propagate to the next lower level rather than all the way to the main memory.

Write-back advantages

- write operations for hits occur at the speed of the cache and multiple writes to the same block require only one write back to the lower level when the block is replaced
- less memory bandwidth is used, making it attractive for multiprocessors
- power is also saved, making it attractive for embedded applications.

Cache principles - 18

The processor stalls for the completion of write operations during a *write-through* procedure. A common optimization to reduce write stalls is to implement a *write buffer*, which allows the processor to continue as soon as data are written to the buffer, overlapping in fact processor execution with memory updating.

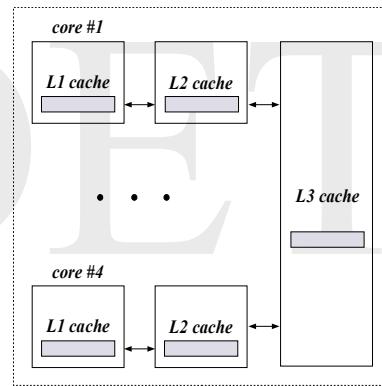
A *write miss* may be dealt with in the following ways

- *write allocate* – a line is allocated whenever a miss occurs, then the write operation takes place; however, if the block size is larger than data to be written, the block must first be retrieved from the lower level
- *non write allocate* – the cache is unaffected by misses, the write operation takes place only at the lower level, which means that blocks stay out of the cache until the processor reads data from them.

Either *write miss policy* can be used with any of the *write policies*. However, *write-back caches* usually implement the *write allocate* policy hoping that subsequent writes to that block are caught by the cache. In the same way, *write-through caches* implement the *non write allocate* policy because all writes must be written at the lower level, so no gain is obtained by the allocation of the block.

Cache principles - 19

For a multicore processor, where multiple simultaneous threads may be competing for access to shared data, protected or not by critical regions, a further problem arises which has to do with *cache coherence*. In such a situation, copies of the same memory block may be stored in lines of level 1 or level 2 caches associated with different processors.



In order to ensure that all processors always see the same data, a *write-through* policy should be implemented for level 1 and level 2 caches and when a *write* takes place, all copies at these levels should be made *stale* so that a transfer from level 3 cache is carried out if a subsequent *read* arises.

Cache principles - 20

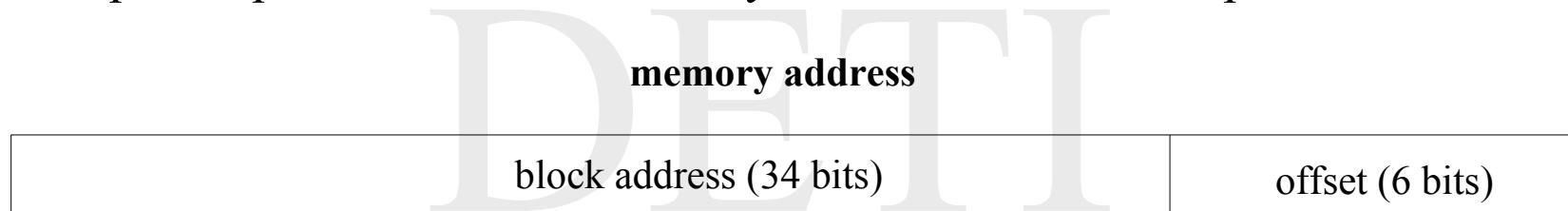
During a program run, the processor not only fetches instructions, but also accesses memory to load and store operands. Although a single, *unified*, or *mixed* cache can supply both, it can generate a communication bottleneck. A typical situation where this happens is when a pipelined processor is executing a *load* or *store* instruction and, at the same time, is fetching the next one. In order to avoid this kind of structural hazard, it is common nowadays to have at the highest level two caches: one dedicated to instructions and another to data. Thus, the communication bandwidth with memory can be doubled through the use of separate ports.

Dedicated caches also allow the individual tuning of each cache, by specifying different storage capacities, block sizes and associativities for each.

The Opteron data cache - 1

The Opteron data cache has a size of 64KB and is organized as a 2-way set associative cache, able to store data blocks of 64 bytes. It features a LRU replacement strategy and implements a policy of *write-back* with *write allocate* on write miss.

The Opteron presents a 40-bit memory address to the cache split as follows.



$$\text{address length} = 40 \text{ bits}$$

$$\text{number of addressable units in main memory} = 2^{40} \text{ bytes} / 2^{37} \text{ words}$$

$$\text{number of blocks in main memory} = 2^{34}$$

$$\text{block size} = 2^6 = 64 \text{ bytes}$$

The Opteron data cache - 2

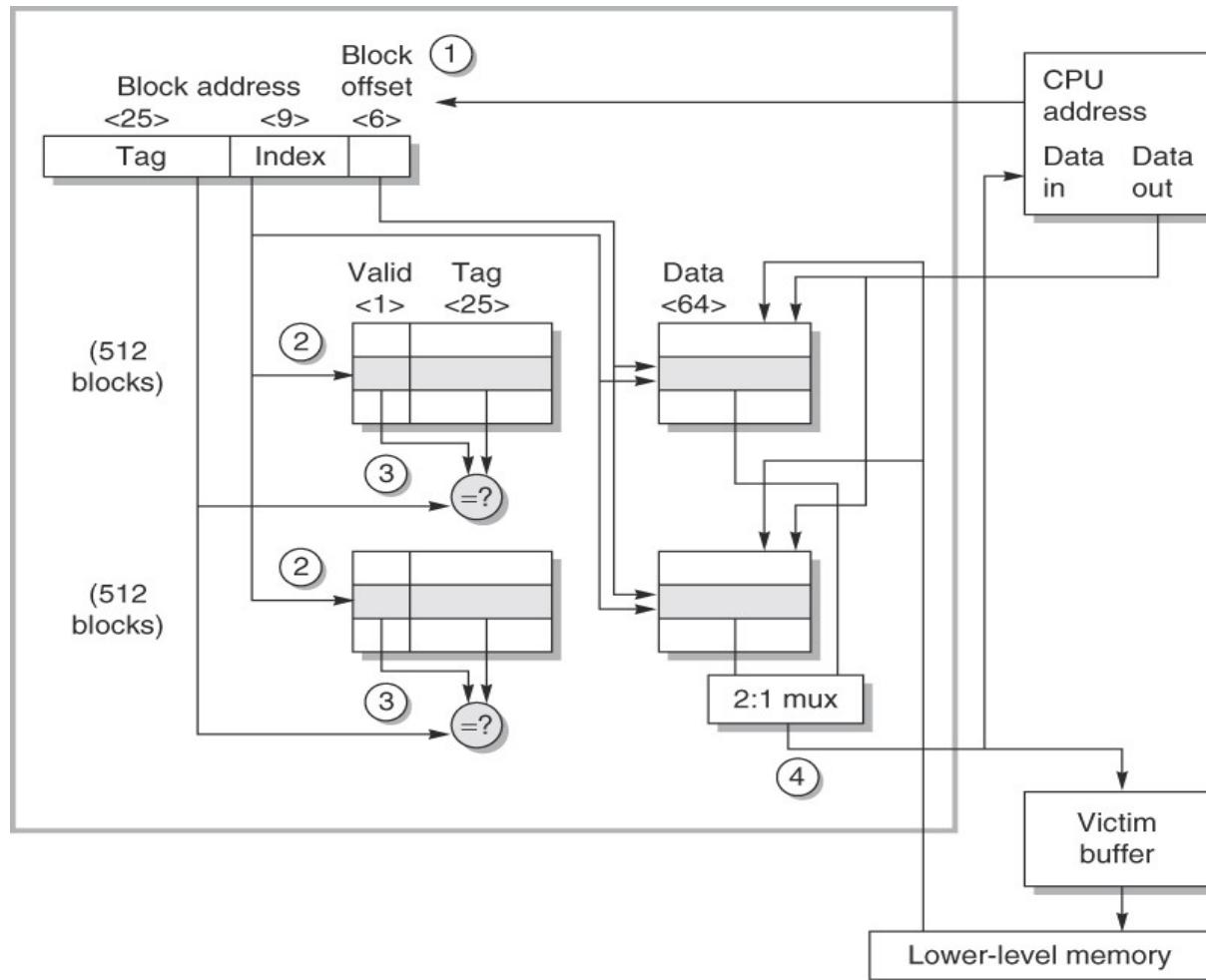
To compute the number of lines in the cache, we make

$$m = \frac{\text{cache size}}{\text{set associativity} \cdot \text{block size}} = \frac{2^{16}}{2 \cdot 2^6} = 2^9 = 512$$



One point worth noting is that, since Opteron is a 64-bit processor, each memory access is at the most 8 bytes wide. Therefore, in addition to the 9 bits of the *index field* to select the proper memory block, the 3 most significant bits of the *block offset* are also used for selection of the proper word within the block.

The Opteron data cache - 3



Source: Computer Architecture: A Quantitative Approach

The Opteron data cache - 4

Several steps may be highlighted in an access to the cache

- *step 1* – the memory address generated by the processor is split into two parts: the *block address*, the most significant 34 bits, which references a particular memory block, and the *block offset*, the least significant 6 bits, which references a specific byte within the block
- *step 2* – the block address, in turn, is also split into two parts: the *tag field*, the most significant 25 bits, which serves to determine if the block is present in the cache, and the *index field*, the least significant 9 bits, which references the set within the cache where the block may be stored
- *step 3* – if the *validation bits* are set, a comparison is carried out between the *tags* of the two lines that belong to the referenced set, and the *tag field* of the block address
- *step 4* – when there is a hit, the cache signals the processor to proceed with the reading or writing of the data; however, since Opteron executes out of order, writing only occurs after the processor signals that the instruction has committed.

The Opteron allows 2 clock cycles for these four steps.

The Opteron data cache - 5

When there is a miss, the cache sends a signal to the processor informing that the data are not yet available. Next, the required block is read from the lower level of the hierarchy, since the cache implements a policy of *write-back* with *write allocate* on write miss. The latency is 7 clock cycles for the first 8 bytes of the block and, then, 2 clock cycles per 8 bytes for the rest of the block.

Due to 2-way set associativity featured by the cache, there are two lines where the block may be stored. If any of them does not contain valid data, its *validation bit* is reset, the line is immediately selected; otherwise, the *least recently used* rule is applied, that is, the selected line is the one whose *LRU bit* is reset. Notice that the implementation of the rule is in this case almost trivial: upon a hit, the line *LRU bit* is set and the *LRU bit* of the counterpart line in the set is reset.

Before doing that, however, because of the *write-back* policy, the *dirty bit* of the line whose block is to be replaced is checked. If it is dirty, its tag and data are first removed to the *victim buffer*. The cache provides space to store up to eight *victim* blocks which are later transferred to the lower level in parallel with other cache activities. If at some moment, however, the *victim buffer* is full, the processor has to stall until space is made available.

The Opteron data cache - 6

Format of a cache line

validation bit	tag (25 bits)	dirty bit	LRU bit	data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)

Cache performance - 1

One method to evaluate *cache performance* is to expand the equation of *processor execution time*. The way to accomplish it is to elicit from the equation the number of clock cycles during which the processor is stalled waiting for a memory access, a parameter usually called *memory stall clock cycles*.

Then one gets

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{memory stall clock cycles}) \cdot \\ \cdot \text{clock cycle time} .$$

Notice that the above equation assumes that the variable *CPU clock cycles* include the number of clock cycles to handle a cache hit and that the processor is completely stalled during a cache miss, the case of an *in-order execution* processor.

Cache performance - 2

The number of *memory stall clock cycles* depends both on the *number of misses* and on the *cost per miss* or *miss penalty*

$$\begin{aligned}\text{memory stall clock cycles} &= \text{number of misses} \cdot \text{miss penalty clock cycles} = \\ &= \text{instruction count} \cdot \frac{\text{misses}}{\text{instruction}} \cdot \text{miss penalty clock cycles} = \\ &= \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \\ &\quad \cdot \text{miss rate} \cdot \text{miss penalty clock cycles}.\end{aligned}$$

Getting the value of *instruction count* is straightforward if one has a listing of the program compilation into assembly language. For speculative processors, however, only committed instructions should be counted. The value of *memory accesses per instruction* can be estimated in the same way: every instruction requires a memory access for instruction fetch and, according to its semantics, may or may not require a data access.

Cache performance - 3

Miss rate can be measured with cache simulators that take an address trace of the instruction and data references, simulate the cache behavior to assert which references hit or miss and report in the end their totals. Many microprocessors today provide hardware to count both the number of misses and of memory references in real time, which turn the task easier.

The variable *miss penalty* is more difficult to estimate. The memory behind the cache may be busy at the time of the miss because of prior memory requests, of a memory refresh, or even of I/O transfers that are taking place. The number of clock cycles also varies at interfaces between different clocks of the processor, the bus or the memory. So, using a single value for *miss penalty* is a simplification.

Finally, since in many cases *miss rate* and *miss penalty* have different values for read and write operations, memory stall cycles could be defined taking this fact into consideration

memory stall clock cycles = instruction count ·

$$\cdot \sum_{i=r,w} \frac{\text{operations}(i)}{\text{instruction}} \cdot \text{miss rate}(i) \cdot \text{miss penalty clock cycles}(i) .$$

Cache performance - 4

Assume a computer system where the value of *clock cycles per instruction* (CPI) is 1.0 when all memory accesses hit the cache. The only data accesses are load and store instructions which account for 50% of all executed instructions on the benchmark being run. The *miss penalty clock cycles* is 25 and the *miss rate* is 2%.

How much slower is it running compared to a computer system having the same processor and executing the same program with no cache misses, or alternatively, how much faster is it running compared to a computer system having the same processor and executing the same program with 100% cache misses?

0% miss rate

$$\begin{aligned}\text{CPU}_{0\text{mr}} \text{ exec time} &= (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot \\ &\quad \cdot \text{clock cycle time} = \\ &= (\text{instruction count} \cdot \text{CPI} + 0) \cdot \text{clock cycle time} = \\ &= 1.00 \cdot \text{instruction count} \cdot \text{clock cycle time}\end{aligned}$$

Cache performance - 5

2% miss rate

$$\begin{aligned}\text{mem stall clock cycles}_{2\text{mr}} &= \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty clock cycles} = \\ &= \text{instruction count} \cdot (1.0 + 0.5) \cdot 0.02 \cdot 25 = \\ &= 0.75 \cdot \text{instruction count}\end{aligned}$$

$$\begin{aligned}\text{CPU}_{2\text{mr}} \text{ exec time} &= (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot \text{clock cycle time} = \\ &= \text{instruction count} \cdot (1.00 + 0.75) \cdot \text{clock cycle time} = \\ &= 1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}\end{aligned}$$

Performance ratio

$$\frac{\text{CPU}_{0\text{mr}} \text{ exec time}}{\text{CPU}_{2\text{mr}} \text{ exec time}} = \frac{1.00 \cdot \text{instruction count} \cdot \text{clock cycle time}}{1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}} = 0.571 .$$

Cache performance - 6

100% miss rate

$$\begin{aligned}\text{mem stall clock cycles}_{100\text{ mr}} &= \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty clock cycles} = \\ &= \text{instruction count} \cdot (1.0 + 0.5) \cdot 1.0 \cdot 25 = \\ &= 37.5 \cdot \text{instruction count} \\ \text{CPU}_{100\text{ mr}} \text{ exec time} &= (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot \\ &\quad \cdot \text{clock cycle time} = \\ &= \text{instruction count} \cdot (1.00 + 37.5) \cdot \text{clock cycle time} = \\ &= 38.5 \cdot \text{instruction count} \cdot \text{clock cycle time}\end{aligned}$$

Performance ratio

$$\frac{\text{CPU}_{100\text{ mr}} \text{ exec time}}{\text{CPU}_{2\text{ mr}} \text{ exec time}} = \frac{38.5 \cdot \text{instruction count} \cdot \text{clock cycle time}}{1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}} = 22.0 .$$

Cache performance - 7

As this example illustrates, cache behavior can have an enormous impact on performance. Cache misses have a double-barrelled impact on a processor with low CPI and a fast clock

- the lower the CPI is, the higher is the relative impact of the processor independent miss penalty measured as a fixed number of clock cycles
- even if the memory hierarchies for two computer systems are identical, the processor with the higher clock rate has a larger number of memory stall cycles.

The importance of the cache for processors with low CPI and high clock rates is thus greater and, therefore, greater is also the danger of neglecting cache behavior in assessing the performance of these computer systems.

Cache performance - 8

Some designers prefer defining *miss rate* as the number of misses per instruction, rather than the number of misses per memory access. These two parameters are related by

$$\frac{\text{total number of misses}}{\text{instruction count}} = \text{miss rate} \cdot \frac{\text{memory accesses}}{\text{instruction count}} .$$

The advantage of using *misses per instruction* to specify the *miss rate* is that it converts this figure of merit into a value which is implementation independent. Speculative processors, for instance, fetch about twice as many instructions as they are actually committed. Thus, reducing artificially the miss rate if measured as misses per memory reference.

The drawback is that *misses per instruction* is architecture dependent, which means it does not make any sense using it to compare two quite different architectures such as MIPS and Intel 80x86.

Cache performance - 9

Instead of concentrating on the *miss rate* to evaluate the performance of the memory hierarchy, one could use as a better figure of merit the *average memory access time*

$$\begin{aligned}\text{average memory access time} &= \text{hit time} + \\ &+ \text{miss rate} \cdot \text{miss penalty clock cycles} \cdot \text{clock cycle time} .\end{aligned}$$

The variable *hit time* is the time to access the cache on a cache hit and the other variables, *miss rate* and *miss penalty*, have the same meaning as before.

Cache performance - 10

A decision must be taken on whether implementing a 16 KB instruction cache and a 16 KB data cache versus a 32 KB unified cache for a particular computer system.

Simulation results on properly chosen benchmarks have indicated the following values for the number misses per 1000 instructions: 3.82 (16 KB instruction cache), 40.9 (16 KB data cache) and 43.3 (32 KB unified cache), where 36% of the totality of the instructions that were executed are data transfer. Assume that a hit takes 1 clock cycle and the miss penalty clock cycles is 200. Assume also that a load or store hit takes 1 extra clock cycle on the unified cache since there is a single port to service two simultaneous requests and that the data cache and the unified cache implement a write-through policy with a write buffer (stalls on writing can be ignored).

$$\text{miss rate} = \frac{\frac{\text{misses per 1000 instructions}}{1000}}{\frac{\text{memory accesses}}{\text{instruction count}}}$$

Cache performance - 11

The *instruction cache* has exactly one memory access per instruction

$$\text{miss rate}_{16 \text{ KB instruction}} = \frac{3.82 \cdot 10^{-3}}{1.00} = 0.004 .$$

The *data cache* has in average 0.36 memory accesses per instruction

$$\text{miss rate}_{16 \text{ KB data}} = \frac{40.90 \cdot 10^{-3}}{0.36} = 0.114 .$$

The *unified cache* has in average 1.36 memory accesses per instruction

$$\text{miss rate}_{32 \text{ KB unified}} = \frac{43.30 \cdot 10^{-3}}{1.36} = 0.032 .$$

Cache performance - 12

The effective *miss rate* of the combined *instruction + data caches* is given by

$$\begin{aligned}\text{miss rate}_{16\text{ KB instruction + data}} &= \frac{\text{mem accesses}_{\text{instruction}}}{\text{mem accesses}_{\text{total}}} \cdot \text{miss rate}_{16\text{ KB instruction}} + \\ &\quad + \frac{\text{mem accesses}_{\text{data}}}{\text{mem accesses}_{\text{total}}} \cdot \text{miss rate}_{16\text{ KB data}} = \\ &= \frac{1.00}{1.36} \cdot 0.004 + \frac{0.36}{1.36} \cdot 0.114 = 0.033 .\end{aligned}$$

Notice that the 32 KB unified cache has a slightly lower miss rate than the combination of two separate 16 KB instruction and data caches!

Cache performance - 13

The *average memory access clock cycles* is given by

average memory access clock cycles =

$$\begin{aligned} &= \frac{\text{mem accesses}_{\text{instruction}}}{\text{mem accesses}_{\text{total}}} \cdot (\text{hit time} + \text{miss rate}_{\text{instruction}} \cdot \text{miss penalty clock cycles}) + \\ &\quad + \frac{\text{mem accesses}_{\text{data}}}{\text{mem accesses}_{\text{total}}} \cdot (\text{hit time} + \text{miss rate}_{\text{data}} \cdot \text{miss penalty clock cycles}) \end{aligned}$$

yielding in each case

average memory access clock cycles_{32 KB unified} =

$$= \frac{1.00}{1.36} \cdot (1.0 + 0.032 \cdot 200) + \frac{0.36}{1.36} \cdot (2 + 0.032 \cdot 200) = 7.66$$

Cache performance - 14

$$\begin{aligned} \text{average memory access clock cycles}_{16\text{ KB instruction + data}} &= \\ &= \frac{1.00}{1.36} \cdot (1.0 + 0.004 \cdot 200) + \frac{0.36}{1.36} \cdot (1 + 0.114 \cdot 200) = 7.62 . \end{aligned}$$

DETI

Although the two separate 16 KB instruction and data caches have a slightly higher effective miss rate than the 32 KB unified cache, the fact is that when one considers the average memory access time, the result is reversed due to the existence in this case of two memory ports per clock cycle, thus avoiding the structural hazard present at the single-port unified cache.

Cache performance - 15

What is the impact of two different cache organizations on the performance of a specific processor?

Assume that the CPI with a perfect cache (no cache misses) is 1.6, that the clock cycle is 0.35 ns and that there are 1.4 memory references per instruction for the benchmark being run. The size of both caches is 128 KB and both have a block size of 64 bytes. The first is organized as direct mapped and the second as 2-way set associative. As the Opteron data cache illustrates, a multiplexor must be added to select between the data in the set depending on the tag match. Since the speed of the processor can be tied directly to the speed of a cache hit, suppose the processor clock cycle time is stretched 1.35 times to accomodate the selection multiplexor of the set associative cache. In both cases, the hit time is 1 clock cycle, the miss penalty is 65 ns and the miss rate is 2.1% and 1.9%, respectively, for the direct map and the 2-way set associative caches.

Cache performance - 16

$$\begin{aligned}\text{average memory access time}_{128\text{KB direct mapped}} &= \\ &= \text{hit clock cycles} \cdot \text{clock cycle time} + \text{miss rate} \cdot \text{miss penalty} = \\ &= 1.0 \cdot 0.35 + 0.021 \cdot 65 = 1.72 \text{ ns}\end{aligned}$$

$$\begin{aligned}\text{average memory access time}_{128\text{KB 2-way set associative}} &= \\ &= \text{hit clock cycles} \cdot \text{effective clock cycle time} + \text{miss rate} \cdot \text{miss penalty} = \\ &= 1.0 \cdot 1.35 \cdot 0.35 + 0.019 \cdot 65 = 1.71 \text{ ns}\end{aligned}$$

Notice that the 128 KB 2-way set associative cache has a slightly lower average memory access time than the 128 KB direct mapped cache!

Cache performance - 17

CPU execution time_{128 KB direct mapped} =

$$\begin{aligned} &= \text{instruction count} \cdot \text{CPI} \cdot \text{clock cycle time} + \\ &\quad + \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty} = \\ &= (1.6 \cdot 0.35 + 1.4 \cdot 0.021 \cdot 65) \cdot \text{instruction count} = 2.47 \cdot \text{instruction count} \end{aligned}$$

CPU execution time_{128 KB 2-way set associative} =

$$\begin{aligned} &= \text{instruction count} \cdot \text{CPI} \cdot \text{effective clock cycle time} + \\ &\quad + \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty} = \\ &= (1.6 \cdot 1.35 \cdot 0.35 + 1.4 \cdot 0.019 \cdot 65) \cdot \text{instruction count} = 2.49 \cdot \text{instruction count} \end{aligned}$$

Although the 128 KB 2-way set associative cache has a lower average memory access time than the 128 KB direct mapped cache, the fact is that when one considers the CPU execution time, the result is reversed due to the stretching of the clock cycle in the case of the associative cache.

Cache performance - 18

For an *out-of-order execution* processor, *miss penalty* can no longer be defined as the full latency of the miss to memory. This question is relevant since *out-of-order* processors are known to tolerate some latency due to cache misses without affecting its performance.

The number of *memory stall cycles* can be redefined to lead to a new definition of *miss penalty* as a non overlapped latency

$$\text{memory stall cycles} = \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot (\text{total miss penalty} - \text{overlapped miss latency}) .$$

It is said that a *processor is stalled in a clock cycle* if it does not retire the maximum possible number of instructions in that cycle. The *stall* is attributed to the first instruction that could not be retired. *Latency*, on the other hand, can be measured from the time the memory instruction is queued in the instruction window, or from the time the address is generated, up to the time the data transfer takes place. Any alternative is valid as long as it is used consistently.

Cache performance - 19

Consider that the processor of the last example has a 1.35 times longer clock cycle time to support out-of-order execution and has a direct mapped cache. Assume also that 30% of the 65 ns miss penalty can be overlapped, thus reducing the average memory stall cycle to 45.5 ns.

$$\begin{aligned}\text{average memory access time}_{\text{direct mapped, OOO}} &= \\ &= \text{hit clock cycles} \cdot \text{clock cycle time} + \text{miss rate} \cdot \text{effective miss penalty} = \\ &= 1.0 \cdot 1.35 \cdot 0.35 + 0.021 \cdot 45.5 = 1.43 \text{ ns}\end{aligned}$$

$$\begin{aligned}\text{CPU execution time}_{\text{direct mapped, OOO}} &= \\ &= \text{instruction count} \cdot \text{CPI} \cdot \text{clock cycle time} = \\ &\quad + \text{instruc count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{effective miss penalty} = \\ &= (1.6 \cdot 1.35 \cdot 0.35 + 1.4 \cdot 0.021 \cdot 45.5) \cdot \text{instruc count} = 2.09 \cdot \text{instruc count}\end{aligned}$$

However, due to its complexity, designers tend to use simulators of the out-of-order processor and of the memory when evaluating the trade-offs in the memory hierarchy to assess that an improvement that helps the average memory latency, also helps the program performance.

Cache optimization - I

The traditional approach to improve the behavior of a cache is to minimize the miss rate. Misses can be modelled into three basic categories

- *compulsory misses* – misses that occur even if the cache had an infinite size; the very first access to any byte or word will always translate into a miss because the memory block where the byte or word resides must be brought first into the cache; they are also known as *cold-start misses* or *first-reference misses*
- *capacity misses* – misses that occur in a fully associative cache; they are directly related to the cache size, if the cache is not large enough, memory blocks will be discarded during program execution and later will be retrieved because they are needed again
- *conflict misses* – misses that occur specifically due to the internal organization of the cache; putting aside the case of a fully associative cache and considering the direct mapped cache as an instance of an 1-way set associative cache, memory blocks may be discarded and later retrieved simply because too many blocks are mapped in some of the sets; they are also known as *collision misses*.

Cache optimization - 2

Miss rate for the Alpha architecture (DEC) using 10 SPEC2000 benchmarks (5 SPECint2000 and 5 SPECfp2000) – LRU replacement – block size = 64 bytes

Source: adapted from Computer Architecture: A Quantitative Approach

Cache size (KB)	Degree of Associativity	Total miss Rate	Miss rate components (value / relative % of total)					
			Compulsory		Capacity		Conflict	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%

Cache optimization - 3

As it should be expected *compulsory misses* are independent of the cache size, *capacity misses* decrease as the cache size increases and *conflict misses* decrease as the degree of associativity increases.

Enlarging the cache size is important. If the upper level memory is too small to contain the fraction of the program's code and data needed by the locality principle, then a significant portion of time is spent transferring memory blocks between two adjacent levels of the hierarchy and *thrashing* will occur. Hence, the computer system will run closer to the speed of the lower-level memory, or even slower due to miss overhead. On the other hand, implementing full associativity to get rid of the conflict misses is expansive in terms of hardware and may lead to a slow clock rate, thus lowering the overall performance.

The *miss model* just presented has its own limits: it gives an insight into average behavior, but does not explain individual misses, or incorporates the replacement policy. Many techniques that reduce miss rates, also increase the *hit time* and/or the *miss penalty*. Therefore, one needs a balanced approach to make the whole computing system faster.

Cache optimization - 4

A straight way to reduce the miss rate is *to increase the block size*. Larger block sizes will diminish *compulsory misses* due to spatial locality. However, if the block size is too large relative to the cache size, the reduction on the number of cache lines tends to increase *capacity* and *conflict misses* and to make the total miss rate worse.

Miss rate vs. block size for DECStation 5000 using SPEC92 benchmarks

Source: Gee, Hill, Pnevimatikatos, Smith, “Cache performance of the SPEC92 benchmark suite”,
IEEE Micro 13:4, August 1993

Block size (bytes)	Cache size			
	4 KB	16 KB	64 KB	256 KB
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.39%	1.15%	0.49%

Cache optimization - 5

At the same time, larger block size tend to increase the *miss penalty*, because of the number of bytes to be transferred also increases

$$\begin{aligned}\text{miss penalty} &= \text{cache latency clock cycles} \cdot \text{clock cycle time} + \\ &+ \text{block size} / \text{bandwidth (clock cycles)} \cdot \text{clock cycle time} .\end{aligned}$$

Average memory access clock cycles vs. block size for DECStation 5000 using SPEC92 benchmarks

Source: Computer Architecture: A Quantitative Approach

$$\text{hit clock cycles} = 1$$

$$\text{cache latency clock cycles} = 80$$

$$\text{bandwidth} = 8 \text{ bytes} / \text{clock cycle}$$

<i>Block size (bytes)</i>	<i>Miss penalty (clock cycles)</i>	<i>Cache size</i>			
		4 KB	16 KB	64 KB	256 KB
16	82	8.027	4.231	2.673	1.894
32	84	7.082	3.411	2.134	1.588
64	88	7.160	3.323	1.933	1.449
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Cache optimization - 6

Since the choice of the block size will be affected by the latency and the bandwidth of the lower level memory, the cache designer has to consider both the miss rate and the miss penalty when deciding on the cache block size.

High latency and high bandwidth encourage a large block size because the cache receives many more bytes per miss for a small increase of the miss penalty. On the other hand, low latency and low bandwidth encourage smaller block sizes because the time saved from a larger block is not significative. One has to remember that having a large number of small blocks may reduce conflict misses.

Capacity misses are obviously reduced by increasing the cache size. However, there is a limit to it because the hit time is potentially longer due to the increase of the logic complexity, making it also both a higher cost and a higher power solution. Notwithstanding, his technique has been specially popular in caches located off the processor chip.

Cache optimization - 7

In the same way, although higher associativity reduces the miss rate, the average memory access time is not always also lessened.

Average memory access clock cycles vs. associativity for DECStation 5000 using SPEC92 benchmarks – LRU replacement – block size = 64 bytes

Source: Computer Architecture: A Quantitative Approach

$$\text{hit clock cycles} = 1$$

$$\text{clock cycle time}_{2\text{-way}} = 1.36 \cdot \text{clock cycle time}_{1\text{-way}}$$

$$\text{clock cycle time}_{4\text{-way}} = 1.44 \cdot \text{clock cycle time}_{1\text{-way}}$$

$$\text{clock cycle time}_{8\text{-way}} = 1.52 \cdot \text{clock cycle time}_{1\text{-way}}$$

Cache size (KB)	Associativity			
	1-way	2-way	4-way	8-way
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82

Main memory

Main memory satisfies primarily the demands of caches and serves as the I/O interface both for the swapping area in a virtual memory organization and for the different device controllers, acting as a destination for input data and a source for output data.

Traditionally, *memory latency*, which affects the cache miss penalty, is the primary concern of the cache, while *memory bandwidth* is the primary concern of the I/O controllers, but it is also important for multilevel caches with their larger block sizes. Higher bandwidth can be achieved by using multiple memory banks, by widening the data bus and by introducing *burst transfer mode*, where multiple words are transferred in the same access.

Main memory is built around DRAM chips. Presently, *memory latency* is expressed through two figures of merit

- *access time* – time interval between the issue of a read / write request and the instant the associated data becomes available / is stored
- *cycle time* – minimum time interval between unrelated memory requests.

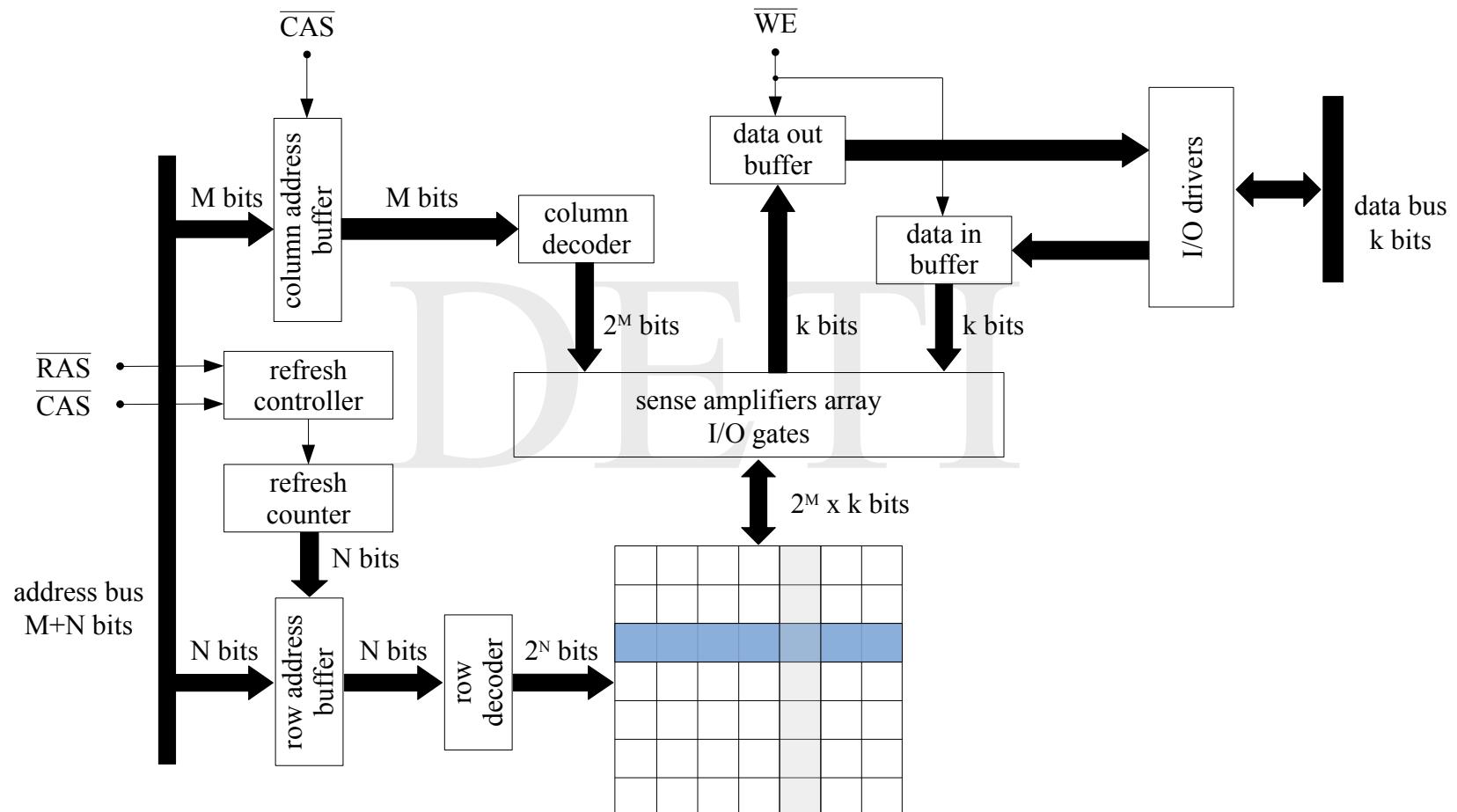
DRAM

As dynamic RAMs grew in capacity, the package cost with all the required address pins became an issue. In order to solve it, the address lines were multiplexed: part of the address is latched internally during the first phase of memory access, with the *row access strobe* control signal, and the remaining address is latched later, during the second phase, with the *column access strobe* control signal.

In order to pack more bits per chip, a single transistor is used to store one bit. Reading this bit destroys the stored information, which means it has to be restored (written back) after each reading. This is one of the reasons why the memory cycle time is longer than the memory access time. With the introduction of multiple bank DRAMs, which enable the rewrite portion of the cycle to be hidden, this problem has been attenuated.

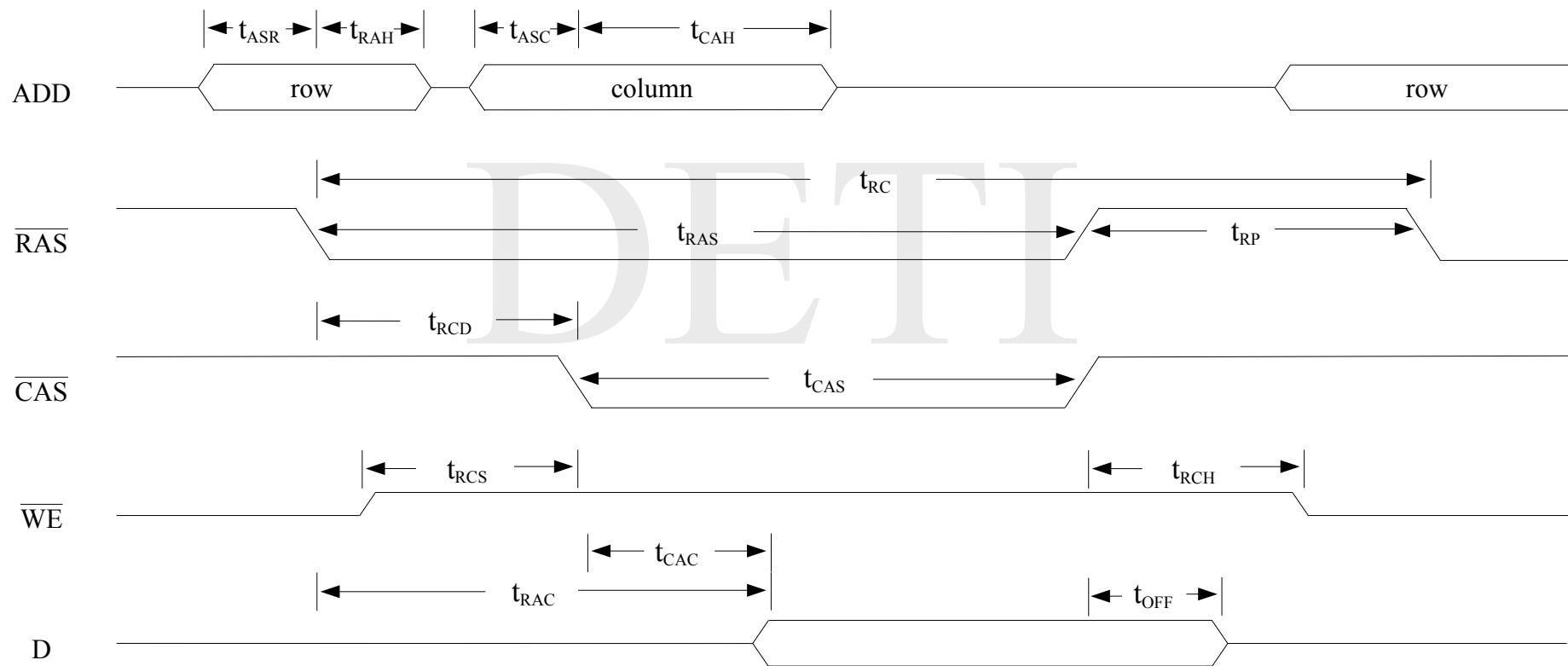
Also, to prevent loss of information when a stored bit is not read for a long period of time, a charge refreshment has to take place. All the bits belonging to the same row can be refreshed simultaneously just by accessing a column of that row. Hence, means have to be provided for every DRAM to access each of its rows within some time frame, typically a few tens of milliseconds. Memory controllers have hardware to carry out this task.

Asynchronous DRAM - 1



Asynchronous DRAM - 2

Read operation



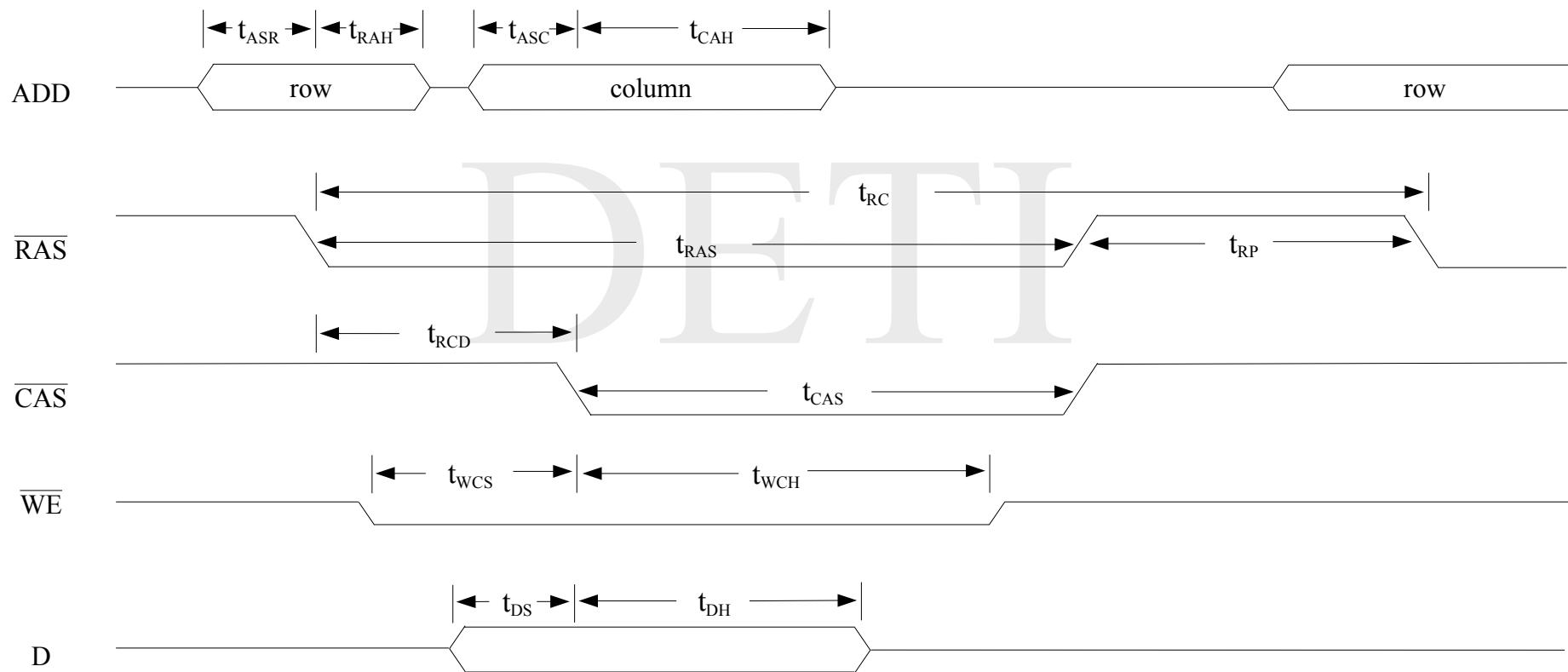
Asynchronous DRAM - 3

Read operation

- the *write enable* signal is asserted high by the *memory controller* to mean a read operation is taking place
- a N-bit address is placed in the address bus and is latched into the *row address buffer* upon the falling edge of the *row access strobe* asserted by the *memory controller*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells
- a M-bit address is placed next in the address bus and is latched into the *column address buffer* upon the falling edge of the *column access strobe* asserted by the *memory controller*
- the data values kept in the selected column of the *sense amplifiers array* is latched into the *data out buffer* to drive the data bus

Asynchronous DRAM - 4

Write operation



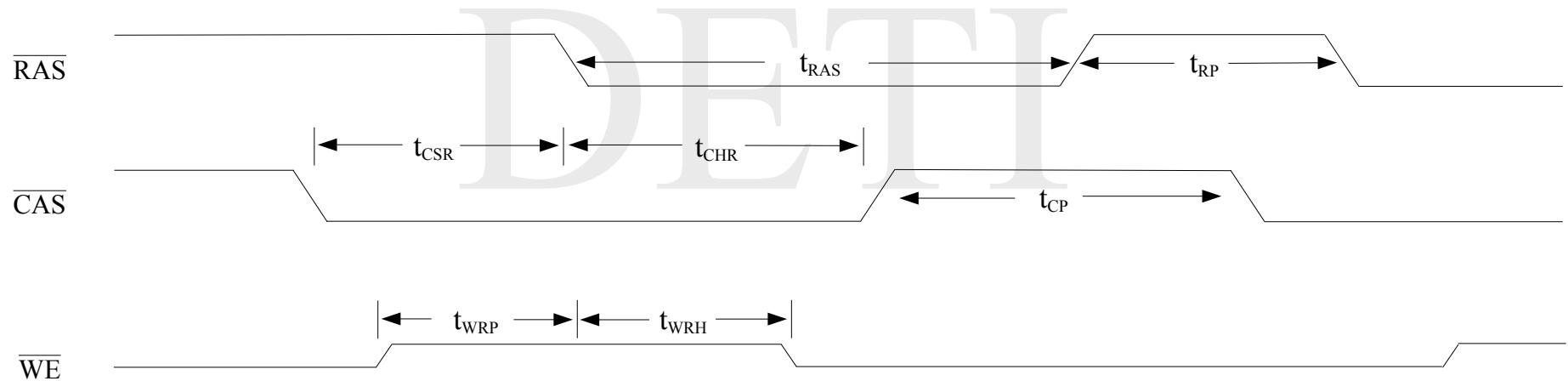
Asynchronous DRAM - 5

Write operation

- the *write enable* signal is asserted low by the *memory controller* to mean a write operation is taking place
- a N-bit address is placed in the address bus and is latched into the *row address buffer* upon the falling edge of the *row access strobe* asserted by the *memory controller*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells
- a M-bit address is placed next in the address bus and is latched into the *column address buffer* upon the falling edge of the *column access strobe* asserted by the *memory controller*
- the data values present in the data bus are latched into the *data in buffer* upon the falling edge of the *column access strobe* and replace the values stored in the selected column of the *sense amplifiers array* before the writing back to the memory cells

Asynchronous DRAM - 6

Refresh operation (CAS before RAS)



Asynchronous DRAM - 7

Refresh operation (CAS before RAS)

- the *column access strobe* signal is asserted low by the *memory controller* before the *row access strobe* and the *write enable* signal is asserted high, the control logic inside the chip interprets this arrangement as a refresh operation
- address lines inputs are ignored and the contents of the *refresh counter* is latched into the *row address buffer* upon the falling edge of the *row access strobe*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells

Asynchronous DRAM - 8

Asynchronous DRAM interface was modified to improve the performance of read and write operations to the same memory row by avoiding the requirement of pre-charging and opening the row repeatedly for access to a different column.

In *page mode* DRAM, after a row is opened by holding *row access strobe* low, the row is kept open and multiple read or write operations are performed to any of the columns of the row. Each column access is started by asserting *column access strobe* low and presenting a column address. For read operations, the *write enable* signal should also be set high. For write operations, the *write enable* signal should be set low and data values to be written should be presented along with the column address.

Page mode DRAM was later improved with a small modification which further reduced latency, giving rise to the so-called *fast page mode* DRAM, or FPM DRAM. In a *page mode* DRAM, *column access strobe* is asserted after the column address is supplied. In a FPM DRAM, the column address can be supplied while *column access strobe* is still deasserted. The column address propagates through the column address data path, but does not output data on the data pins until *column access strobe* is asserted.

Synchronous DRAM - 1

Synchronous DRAM, or SDRAM, changes in a radical way how the external memory interface interacts with the device. Control signals no longer have a direct effect on the internal functions, but an externally controlled clock signal is used to manage a built-in finite state machine which acts upon incoming commands. Thus, $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals no longer act as strobes, but become instead part of the input command.

The internal operations are pipelined to improve performance. Previously initiated operations being completed, while new commands are received and start processing.

The two-dimensional memory cell array is divided into several equally sized, but independent sections, called *banks*, allowing the device to operate on a memory access command, read or write, in each bank concurrently and speed up access in an interleaved manner.

Burst transfer mode is also implemented for accessing multiple data words with the same command.

Synchronous DRAM - 2

Later generations of SDRAMs improved transfer bandwidth by allowing data transfer to take place both in the rising edge and the falling edge of the externally controlled clock signal. This optimization is known as *double data rate*, or DDR.

DDR has given rise to a sequence of standards

- DDR, or DDR1 – has a voltage supply of 2.5 V and operates at clock frequencies of 133, 150 and 200 MHz
- DDR2 – has a voltage supply of 1.8 V and operates at clock frequencies of 266, 333 and 400 MHz
- DDR3 – has a voltage supply of 1.5 V and operate at clock frequencies of 533, 666 and 800 MHz
- DDR4 – has a voltage supply of 1.2 V and operate at clock frequencies of 1066, 1333 and 1600 MHz.

DRAMs are usually sold in small boards, called *dual inline memory modules*, or DIMMs, which contain 4 to 16 SDRAM chips and are normally organized to provide a word length of 64 data bits + *error correcting code* bits.

Synchronous DRAM - 3

64 Mbit SDRAM

(organized in 4 banks of 2048 rows x 1024 columns memory cells of 8 bits each)

CS	RAS	CAS	WE	BA_n	A10	An	Command
H	X	X	X	X	X	X	command inhibit (the device is not selected)
L	H	H	H	X	X	X	no operation
L	H	H	L	X	X	X	burst terminate: stop a burst read, or a burst write, in progress
L	H	L	H	bank	L	column	read a burst of data from the current active row
L	H	L	H	bank	H	column	read a burst of data from the current active row with precharge (close the row) when done
L	H	L	L	bank	L	column	write a burst of data from the current active row
L	H	L	L	bank	H	column	write a burst of data from the current active row with precharge (close the row) when done
L	L	H	H	bank	row		activate (open) the row for a read or write command
L	L	H	L	bank	L	X	precharge, or deactivate (close), the current row of selected bank
L	L	H	L	X	H	X	precharge, or deactivate (close), the current row of all banks
L	L	L	H	X	X	X	refresh one row of each bank (auto refresh) using an internal counter (all rows must be precharged)
L	L	L	L	00	mode		load mode register for chip configuration: the control word is in A10-A0 the most significant settings are to program CAS latency (2 or 3 clock cycles) and burst transfer length (1, 2, 4 or 8 words)

DRAM generations

Time specifications

Source: Computer Architecture: A Quantitative Approach

Production year	Chip size (bit)	DRAM type	Slowest DRAMs (ns)	Fastest DRAMs (ns)	CAS / data transfer time (ns)	Cycle time (ns)
1980	64K	DRAM	180	150	75	250
1983	256K	DRAM	150	120	50	220
1986	1M	DRAM	120	100	25	190
1989	4M	DRAM	100	80	20	165
1992	16M	DRAM	80	60	15	120
1996	64M	SDRAM	70	50	12	110
1998	128M	SDRAM	70	50	10	100
2000	256M	DDR1	65	45	7	90
2002	512M	DDR1	60	40	5	80
2004	1G	DDR2	55	35	5	70
2006	2G	DDR2	50	30	2.5	60
2010	4G	DDR3	36	28	10	37
2012	8G	DDR3	30	24	0.5	31

DDR-SDRAM classification

DIMM characterization

Source: Computer Architecture: A Quantitative Approach

Standard	Clock rate (MHz)	M (transf/s)	DRAM name	MB/s/DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10664	PC10700
DDR3	800	1600	DDR3-1600	12800	PC12800
DDR4	1066-1600	2133-3200	DDR4-3200	17056-25600	PC25600

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
 - Appendix B: *Review of Memory Hierarchy*
 - Chapter 2: *Memory Hierarchy Design*
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
 - Chapter 4: *Cache Memory*
 - Chapter 5: *Internal Memory*
 - Chapter 6: *External Memory*
 - Chapter 8: *Operating System Support*

universidade de aveiro



Arquitecturas de Alto Desempenho

Instruction-Level Parallelism (Complements)

António Rui Borges

Summary - 1

- *Efficiency of pipelining*
- *Data dependences and hazards*
 - *Data dependences*
 - *Name dependences*
 - *Data hazards*
 - *Control dependences*
- *Basic compiler techniques for exposing ILP*
- *Advanced branch prediction*
 - *Correlating branch predictors*
 - *Tournament predictors*
- *Dynamic scheduling*
 - *Scoreboarding*
 - *Tomasulo's algorithm*

Summary - 2

- *Hardware-based speculation*
- *Exploiting ILP using multiple issue*
- *Statically scheduled superscalar processor*
 - *ARM Cortex-A8*
- *Dynamically scheduled superscalar processor*
 - *Intel Core i7*
- *Suggested reading*

Efficiency of pipelining - 1

The simplicity of a instruction set is a fundamental property in building a pipeline for its implementation. Simple instruction sets offer yet another advantage as they are instrumental in making the scheduling of individual instructions more straightforward. These advantages seem to be so significant that almost all recent pipelined implementations of complex instruction sets actually translate first the instructions into simple RISC-like operations and only then proceed to their pipelining and scheduling.

As a means to express the degree of success obtained by running a given program in a pipelined processor, the following equation can be used

$$\text{CPI}_{\text{prog}} = \text{CPI}_{\text{ideal}} + \text{structural stalls} + \text{data stalls} + \text{control stalls} ,$$

where the $\text{CPI}_{\text{ideal}}$, clock cycles per instruction in the ideal situation, is a measure of the maximum performance attainable by the pipelined implementation and the different types of stalls are assumed to be average values per instruction.

Efficiency of pipelining - 2

The amount of parallelism available within the *basic block* of a program – a code segment with no branches in, except for the entry point, and no branches out, except for the exit point – is quite small. For a typical MIPS program, for instance, the average dynamic branch frequency is often between 15% and 25%, which means that as few as three to six instructions execute between a pair of branches. Furthermore, since these instructions are likely to depend upon one another, the amount of instruction overlap that can be exploited within a basic block, is likely to be less than the average block size. Therefore, to obtain substantial performance enhancements, ILP must be exploited across multiple basic blocks.

There are two largely distinct approaches to fulfill this goal

- an approach that relies on software technology to elicit parallelism statically at compile time
- an approach that relies on hardware to help to discover and exploit dynamically the inherent parallelism during execution.

Data dependences and hazards

Determining how one instruction depends upon another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit ILP efficiently one must determine first which instructions can be executed in parallel. If two instructions are *independent* from one another, they can be executed without hindrance in a pipeline of arbitrary depth (no stalls need to be inserted), assuming there are sufficient resources (i.e. no structural hazards exist). If one instruction *depends* upon another, there will be time restrictions enforcing their execution, they must proceed in order and may often be only partially overlapped.

Three types of dependences are relevant

- *data* dependences, also called *true data* dependances
- *name* dependences
- *control* dependences.

Data dependences - 1

An instruction j is said to be *data dependent* upon an instruction i if and only if either of the following conditions hold

- the instruction i produces a value which is used by the instruction j
- the instruction j is data dependent upon an instruction k and the instruction k is data dependent upon the instruction i
- the instructions j and i are connected by a chain of dependences of the second type.

Note that a dependence within an instruction, DADD R1, R1, R1, for instance, is not considered a data dependence.

Data dependences - 2

Consider the following code sequence

LOOP:	L.D	F0, 0 (R1)	↓
	ADD.D	F4, F0, F2	↓
	S.D	F4, 0 (R1)	↓
	DADDUI	R1, R1, -8	↓
	BNE	R1, R2, LOOP	

For simplicity, the effects of delayed branches are ignored. Data dependencies between instructions are represented by vertical arrows.

The existence of dependences implies that there will be a chain of one or more data hazards between successive instructions. Executing the instructions in a pipelined processor with an interlocking unit and a pipeline depth longer than the distance between the instructions measured in clock cycles, causes the processor to detect a hazard and stall, if it cannot be resolved by forwarding. Thus, reducing the overlap. In a pipelined processor without an interlocking unit, on the other hand, it is the compiler responsibility to schedule dependent instructions in a manner that they will not overlap completely, since otherwise the program will not run correctly.

Data dependences - 3

The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved. Dependences are, therefore, a property of *programs*. On the other hand, whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall, are properties of the *pipeline organization*. This difference is crucial to understand how ILP can be exploited.

A data dependence conveys three ideas

- the possibility of a hazard
- the specification of the order the results must be computed
- an upper bound on how much parallelism can possibly be exploited.

A dependence can be overcome in two different ways

- maintaining a dependence, but avoiding a hazard
- eliminating a dependence by transforming the code.

Data dependences - 4

Scheduling the code is the primary method to avoid a hazard without altering a dependence and such scheduling can be done both by the compiler and by the hardware.

A data value is communicated between instructions either through a register or the register bank, or a memory location. When the flow of information occurs through a register, detecting the dependence is straightforward since the register names are fixed for each instruction; although it may become more complicated if branches intervene, since correctness concerns force the compiler or the hardware to be conservative. When the flow of information takes place through memory locations, the detection is more difficult since two addresses may refer to the same memory location, but look different. In addition, the effective address of a load or store instruction may change from one execution of the instruction to the next, further complicating the detection of a dependence.

Name dependences - 1

A *name dependence* is present when two instructions use the same register or memory location, called *name*, but without any flow of information taking place between them. There are two types of *name dependences*

- an *antidependence* between an instruction i and an instruction j occurs when the instruction j writes to a register or memory location that the instruction i reads – the original instruction ordering must be preserved to ensure that the correct value is read
- an *output dependence* between an instruction i and an instruction j occurs when both instructions i and j write a value to the same register or memory location – the original instruction ordering must be preserved to ensure that the value finally written is the correct one.

Name dependences - 2

Since a *name dependence* is not a true data dependence, as there is no value transmitted between the involved instructions, the instructions themselves may execute simultaneously, or be reordered, provided the *name* (register or memory location) used in the instructions is modified in one of them to remove the conflict.

This renaming can be more readily done for register operands, where it is called *register renaming*. Register renaming can be performed either statically by the compiler, or dynamically by the hardware.

Data hazards - 1

A *data hazard* exists whenever there is a data or name dependence between instructions and they are close enough to generate, by their overlap in the pipeline during execution, a change in the order of access to the operand involved in the dependence.

Because of the dependence, the *program order of execution* must be preserved, that is, the order of instruction execution if they were taken one at the time and executed sequentially as determined by the original source code. The goal of both software and hardware techniques is to exploit parallelism by preserving program order *only when it affects the outcome of the program*, and not in all circumstances.

Detecting and avoiding hazards ensures that the necessary program order is preserved.

Data hazards - 2

Data hazards can be classified in three different categories depending on the combination of operand accesses present in the instructions. A name convention is used that portrays the instruction order that must be preserved by the pipeline.

Consider two instructions i and j , with i preceding j in the program, then the possible hazards are

- RAW (*read after write*) – j tries to read the operand before i writes a value to it, so j gets a wrong value; it is the most common form of data hazard and corresponds to a true data dependence
- WAW (*write after write*) – j tries to write a value to an operand before i writes its value to it, so the final value is wrong; it arises in pipelines that allow writing in more than one pipe stage, or allow out of order instruction completion, and corresponds to an output dependence

Data hazards - 3

- WAR (*write after read*) – j tries to write a value to an operand before i reads it, so i gets a wrong value; it cannot occur in most static issue pipelines, even deeper pipelines or floating point pipelines, because usually all reads are early and all writes are late; it arises when there are some instructions that write results early in the pipeline and other instructions that read operands late, or when out of order execution is allowed.

Notice that the RAR (*read after read*) is not a data hazard. The combination of operations is idempotent.

Control dependences - 1

A *control dependence* determines the ordering of an instruction i with respect to a branch, so that the instruction i is executed in the correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches and, in general, these control dependences must be preserved to maintain the program order.

Consider the code segment

```
if (cond1) S1;  
if (cond2) S2;
```

S1 is control dependent on condition cond1 and S2 is control dependent on condition cond2, but not on condition cond1.

Two constraints are in general imposed by control dependencies

- an instruction that is control dependent on a branch, cannot be moved *before* the branch so that its execution is *no longer controlled* by the branch
- an instruction that is not control dependent on a branch, cannot be moved *after* the branch so that its execution is *controlled* by the branch.

Control dependences - 2

When processors preserve strict program order, they ensure that control dependences are also preserved. One should point out, however, that it is permissible, although inefficient, to execute instructions that should not have been executed, thereby violating the control dependences, if this can be done without affecting the program correctness. Which means that control dependence is not *the critical* property that must be preserved. The two critical properties to program correctness – and normally preserved by maintaining both data and control dependences – are *exception behavior* and *data flow*.

Control dependences - 3

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program.

Consider the code segment (again the effects of delayed branches are ignored)

DADDU	R2, R3, R4
BEQZ	R2, L1
LD	R1, 0(R2)
L1:	...

It is clear that if the data dependence involving the register R2 is not maintained, the result of the program may be changed. Less obvious is the fact that if one ignores the control dependence and move the *load* instruction to a place just before the branch, this instruction may cause a memory protection violation.

To be able to reorder the instructions, and still preserve the data dependence, it would be necessary to ignore the exception when the branch is taken. Later on, a hardware technique, called *speculation*, will be studied and, as it will be seen, will allow the overcoming of this problem.

Control dependences - 4

The *data flow* is the actual flow of data values among instructions that produce them and instructions that consume them. Branches make the data flow dynamic, since they allow the data for a given instruction to come from different source locations, thereby an instruction may be data dependent on several predecessor instructions. Program order is what in fact determines which predecessor will deliver the value at each moment and program order is ensured by maintaining the control dependences.

Consider the code segment (again the effects of delayed branches are ignored)

DADDU	R1, R2, R3
BEQZ	R4, L
DSUBU	R1, R5, R6
L:	...
OR	R7, R1, R8

As it can be seen, the value of the register R1 used by the *or* instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness, control dependence is also required. As for the exception problem, *speculation* will also help to lessen the impact of control dependence, while keeping the data flow.

Control dependences - 5

Sometimes it can be determined that violating the control dependence will neither affect the exception behavior, nor the data flow.

Consider the code segment (again the effects of delayed branches are ignored)

DADDU	R1, R2, R3
BEQZ	R12, SKIP
DSUBU	R4, R5, R6
DADDU	R5, R4, R9
.	.
SKIP:	OR R7, R8, R9

Suppose it is known that the registers R4 and R5, destinations of the *subtract* and *add* instructions, respectively, are not used after SKIP – the property of whether a value will be used by an upcoming instruction is called *liveness*. Then, changing the values of R4 and R5 just before the branch would not affect the data flow, since R4 and R5 would be dead in the code region after SKIP.

This type of code scheduling is also a form of speculation, often called *software speculation*. The compiler is betting on the branch outcome, the bet being the branch is usually not taken.

Basic compiler techniques for exposing ILP - 1

To keep a pipeline full, the parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles at least equal to the pipeline latency of that source instruction.

The compiler ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the pipeline functional units.

As an example, consider how the compiler can increase the amount of available ILP by transforming loops as the one bellow

```
for (i = 999; i >= 0; i--)  
    x[i] += s;
```

As it is immediately seen, the loop is highly parallel: each iteration is totally independent from any other one.

Basic compiler techniques for exposing ILP - 2

Straightforward MIPS code of the loop (delayed branches are ignored)

```
LOOP:    L.D      F0, 0(R1)
          ADD.D    F4, F0, F2
          S.D      F4, 0(R1)
          DADDUI   R1, R1, -8
          BNE     R1, R2, LOOP
```

It is assumed that the register R1 contains initially the address of the element $x[999]$ of the array and that the contents of the register R2+8 is the address of the element $x[0]$.

Basic compiler techniques for exposing ILP - 3

Latencies of FP operations

Source: Computer Architecture: A Quantitative Approach

Instruction producing the result	Instruction using the result	Latency in clock cycles
FP ALU op	another FP ALU op	3
FP ALU op	store double	2
load double	FP ALU op	1
load double	store double	0

Basic compiler techniques for exposing ILP - 4

**MIPS code of the loop, with elimination of data dependencies
(delayed branches are ignored)**

			<i>clock cycle</i>
LOOP:	L.D	F0, 0 (R1)	1
	<i>stall</i>		2
	ADD.D	F4, F0, F2	3
	<i>stall</i>		4
	<i>stall</i>		5
	S.D	F4, 0 (R1)	6
	DADDUI	R1, R1, -8	7
	<i>stall</i>		8
	BNE	R1, R2, LOOP	9

The processing of each element takes 9 clock cycles (it assumes that there is no forwarding at the ID stage).

Basic compiler techniques for exposing ILP - 5

**MIPS code of the loop, scheduled for the pipeline
(delayed branches are ignored)**

			<i>clock cycle</i>
LOOP:	L.D	F0, 0 (R1)	1
	DADDUI	R1, R1, -8	2
	ADD.D	F4, F0, F2	3
	<i>stall</i>		4
	<i>stall</i>		5
	S.D	F4, 8 (R1)	6
	BNE	R1, R2, LOOP	7

The processing of each element takes 7 clock cycles.

Basic compiler techniques for exposing ILP - 6

**Straightforward MIPS code of the loop, with loop unrolled 4 times
(delayed branches are ignored)**

```
LOOP:    L.D      F0, 0(R1)
          ADD.D    F4, F0, F2
          S.D      F4, 0(R1)
          L.D      F6, -8(R1)
          ADD.D    F8, F6, F2
          S.D      F8, -8(R1)
          L.D      F10, -16(R1)
          ADD.D   F12, F10, F2
          S.D      F12, -16(R1)
          L.D      F14, -24(R1)
          ADD.D   F16, F14, F2
          S.D      F16, -24(R1)
          DADDUI  R1, R1, -32
          BNE     R1, R2, LOOP
```

In real programs, the loop upper bound is not usually known. Assuming it to be n and unrolling the loop k times, two consecutive loops are generated: the first iterates $n \bmod k$ times and has a body equal to the original loop; the second iterates n / k times and has the unrolled body.

Basic compiler techniques for exposing ILP - 7

**MIPS code of the loop, with loop unrolled 4 times and scheduled for the pipeline
(delayed branches are ignored)**

LOOP:	L.D	F0, 0 (R1)
	L.D	F6, -8 (R1)
	L.D	F10, -16 (R1)
	L.D	F14, -24 (R1)
	ADD.D	F4, F0, F2
	ADD.D	F8, F6, F2
	ADD.D	F12, F10, F2
	ADD.D	F16, F14, F2
	S.D	F4, 0 (R1)
	S.D	F8, -8 (R1)
	DADDUI	R1, R1, -32
	S.D	F12, 16 (R1)
	S.D	F16, 8 (R1)
	BNE	R1, R2, LOOP

The processing of each element takes 3.5 clock cycles.

Basic compiler techniques for exposing ILP - 8

Decisions and transformations required to unroll a loop can be summarized in the following points

- determination if unrolling the loop is really useful by finding the degree of independence among its iterations
- use of different registers to avoid unnecessary constraints that would arise by the use of same registers for different computations
- elimination of the extra test and branch instructions and adjustment of the loop termination and iteration code
- determination if the loads and stores in the unrolled loop can be interchanged – this transformation requires analyzing the memory addresses
- scheduling the code by preserving any dependencies needed to yield the same result as the original code.

The key requirement underlying all these transformations is the understanding of how one instruction depends upon another and how the different instructions can be changed and/or reordered, given the dependencies.

Basic compiler techniques for exposing ILP - 9

Three different effects limit the gains from loop unrolling

- a decrease in the amount of overhead saved with each unroll – as predicted by the Amdhal's Law
- code size limitations – for large loops, the code size growth may lead to an increase in the instruction cache miss rate; furthermore, one has to be also concerned with the potential shortfall in registers, called *register pressure*, that is created by applying aggressive unrolling and scheduling strategies
- compiler limitations – the use of sophisticated high-level transformations, whose potential improvements are difficult to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Advanced branch prediction

Branches will hurt pipeline performance because enforcing control dependencies generate hazards which require stalling the pipeline progress in many situations. Loop unrolling is a method to reduce the number of control hazards during execution. The performance loss of branches, however, is more adequately dealt with if their behavior can be predicted.

Simple branch predictors that rely either on compile time information, or on the observed dynamic behavior of a branch in isolation, have already been studied. As the number of instructions in flight, that is, whose execution is taking place or are being considered for, has increased significantly in the past two decades, the need for accurate branch prediction has become more pressing.

Correlating branch predictors - 1

The *2-bit prediction* scheme takes into consideration the recent behavior of a single branch to predict its future behavior. The prediction accuracy is improved if one considers the recent behavior of *other* branches in the program as well.

To see why this idea is relevant, look at the following code segment where the effects of delayed branches are ignored.

	DADDIU	R3, R1, -2	
	BNEZ	R3, L1	; branch b1: R1 ≠ 2?
	DADD	R1, R0, R0	; R1 = 0
L1:	DADDIU	R3, R2, -2	
	BNEZ	R3, L2	; branch b2: R2 ≠ 2?
	DADD	R2, R0, R0	; R2 = 0
L2:	DSUBU	R3, R1, R2	
	BEQZ	R3, L3	; branch b3: R1 = R2?

Notice that the behavior of branch *b3* is correlated to the behavior of branches *b1* and *b2*: if both branches are *untaken*, then branch *b3* is *taken*. A predictor that uses only the behavior of a single branch to predict what happens next, can never capture such a behavior.

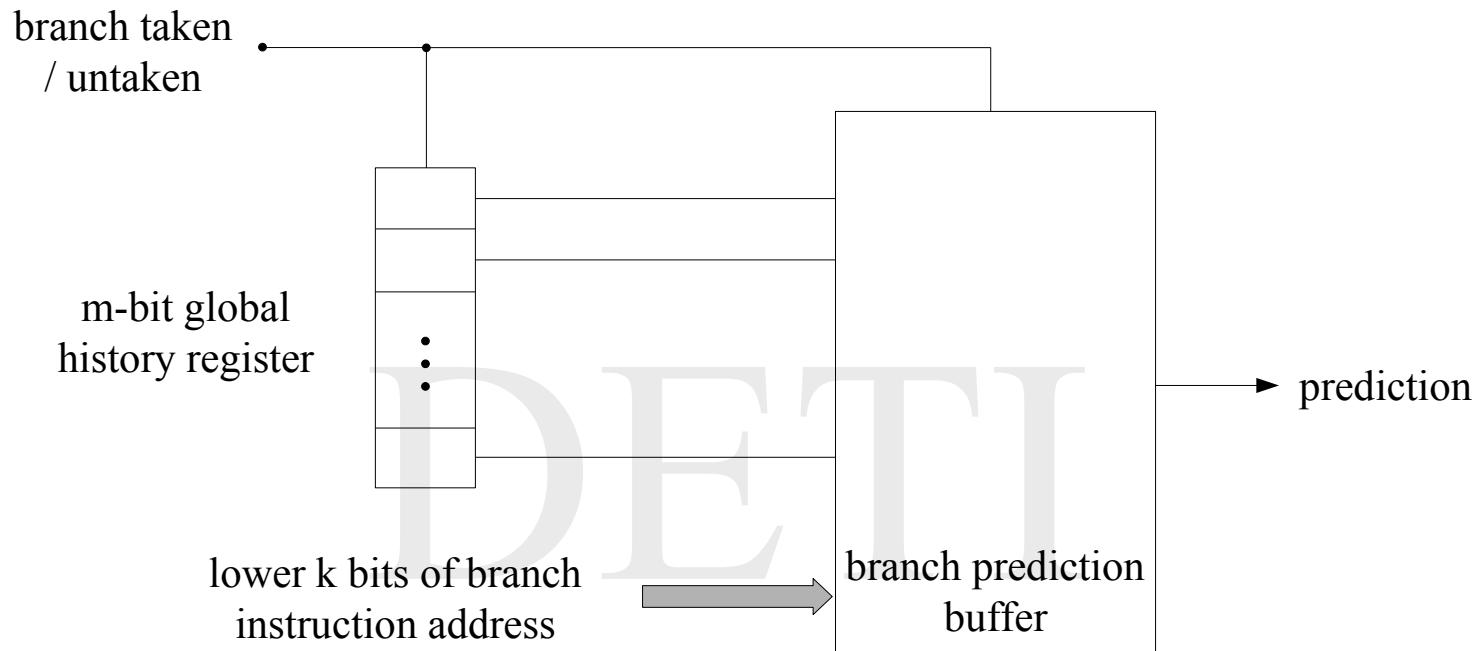
Correlating branch predictors - 2

Branch predictors which add the execution behavior of other branches, besides their own, to make a prediction, are called *correlating predictors*.

A (1,2) *correlating branch predictor*, for instance, considers the execution behavior of the previous branch to choose among a pair of 2-bit branch predictors in predicting a particular branch. In general, a (m,n) *branch predictor* adds the execution behavior of previous m branches to choose among $2^m n$ -bit branch predictors in predicting a particular branch.

The popularity of this kind of branch predictors is that it can yield a higher prediction rate than the 2-bit prediction scheme, while requiring a trivial amount of additional hardware: the global history of the most recent m branches can be recorded in a m -stage shift register, called the *global history register*, where the bit contents at each stage specifies whether the branch of the corresponding order was *taken*, 1, or *untaken*, 0; the *branch prediction buffer* is now indexed by the concatenation of the m -bit global history with the low-order k address bits of the branch instruction.

Correlating branch predictors - 3



A 2-bit predictor with no global history is simply a (0,2) correlating branch predictor and, in comparing branch predictors, the branch predictor buffer size should be kept invariant, that is,

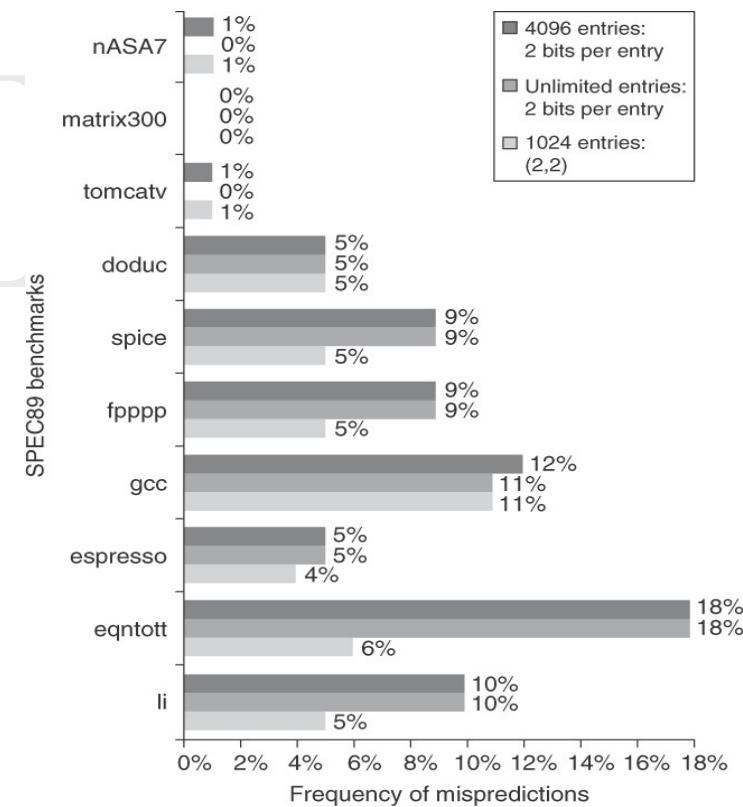
$$2^m \cdot n \cdot 2^k = \text{constant} .$$

Correlating branch predictors - 4

A (2,2) correlating branch predictor with 1K entries is compared with (0,2) simple predictors with 4K entries and an unlimited number of entries.

Comparing 2-bit branch predictors of different types in SPEC89 benchmarks

Source: Computer Architecture: A Quantitative Approach



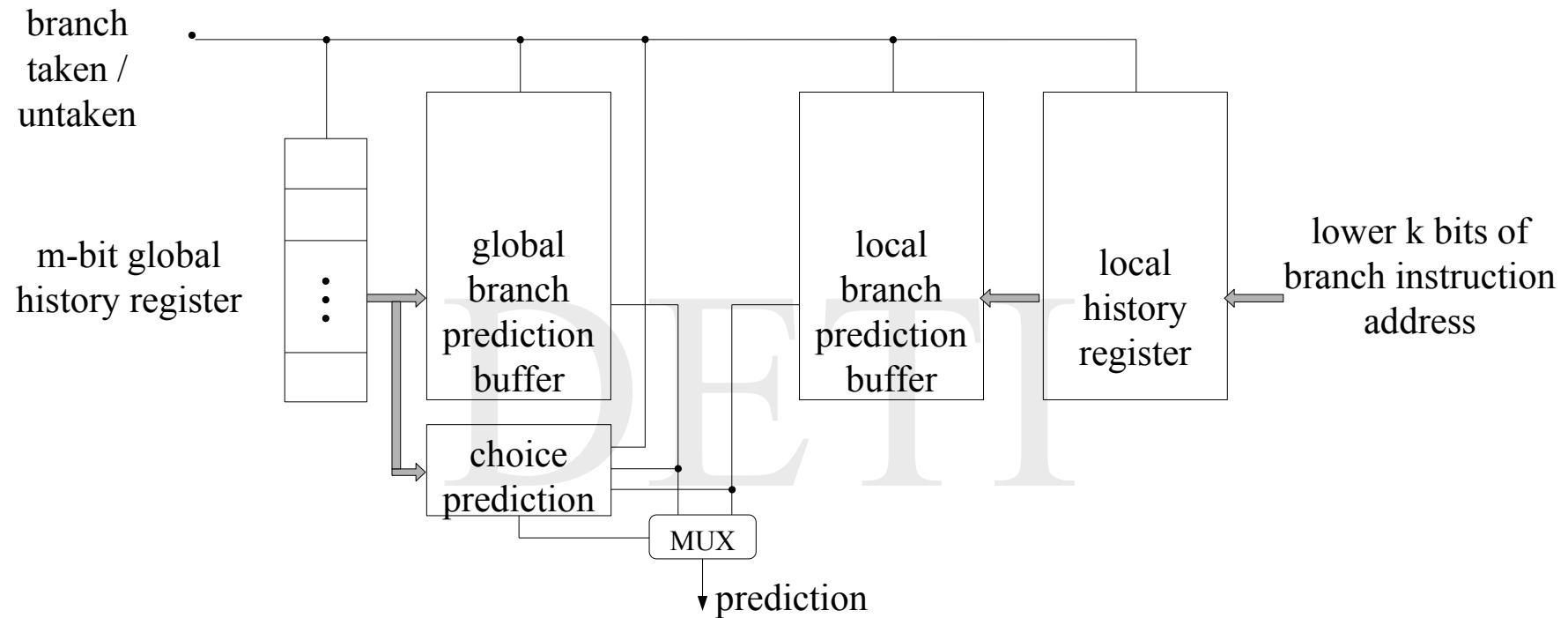
Tournament predictors - 1

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that, by adding global information, the performance could be improved. *Tournament predictors* take this insight to the next level by using multiple predictors, usually one based on global information and one on local information, and combining them with a selector. Tournament predictors can achieve better accuracy at medium sizes of the branch predictor buffer and also make effective use of very large numbers of prediction bits.

Existing tournament predictors use a 2-bit hysteresis counter per branch to choose among two different predictors based on which predictor (local, global or a mix of the two) was the most effective in recent predictions. A 2-bit *hysteresis counter* requires two mispredictions in succession to change its state.

The advantage of tournament predictors is their ability to select the right predictor for a particular branch, which is specially crucial for integer programs. Typically, they select the global predictor almost 40% of the time for integer benchmarks and less than 15% of the time for floating point benchmarks.

Tournament predictors - 2

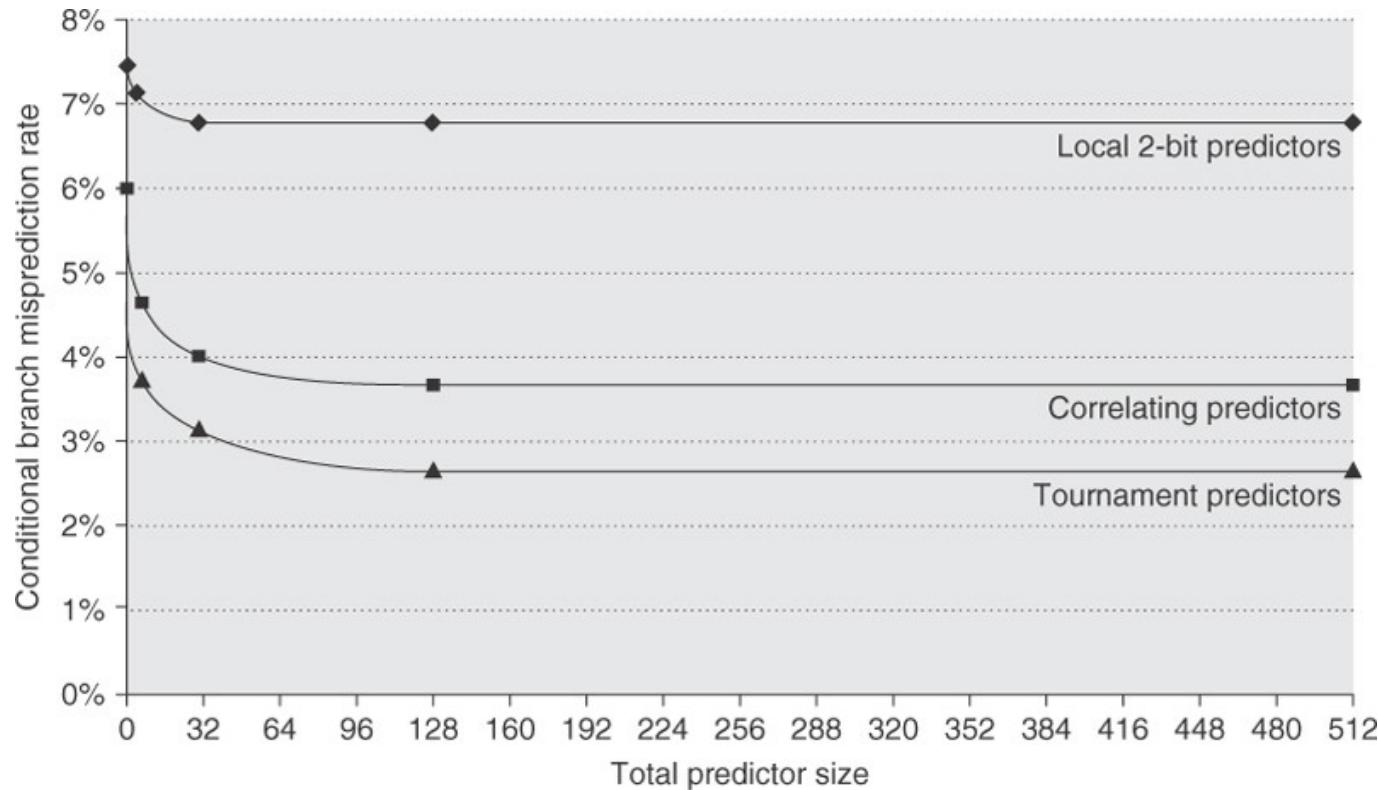


Both the *global branch prediction buffer* and the *choice prediction buffer* consist of $2^m n$ -bit predictors and the *local branch predictor buffer* of $2^u v$ -bit predictors. The *local history register*, on the other hand, consists of $2^k u$ -bit *local history registers*.

Tournament predictors - 3

Comparing the misprediction rate for three different predictors in SPEC89 benchmarks

Source: Computer Architecture: A Quantitative Approach



Although these data is from an older version of SPEC, recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limits at slightly larger predictor sizes.

Dynamic scheduling - 1

A major limitation of simple pipelining techniques is they use *in-order* instruction issue and execution. If a data dependence between an instruction in the pipeline and an incoming instruction occurs, which cannot be solved by the *forwarding unit*, the *interlocking unit* stalls the pipeline, starting at the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared.

Dynamic scheduling is another way of addressing the problem: the hardware re-arranges instruction execution, while maintaining data flow and exception behavior, so that stalls are minimized. This entails, however, a significant increase in complexity.

There are several advantages resulting from the use of this technique

- it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, eliminating the need of multiple binaries
- it enables handling some cases where dependences are unknown at compile time, involving, for instance, memory references and data dependent branches or the use of dynamic linked libraries
- it allows the processor to cope with unpredictable delays, such as cache misses, by executing other code while waiting for the miss to resolve.

Dynamic scheduling - 2

Consider the code segment.

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F12, F8, F14

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet, SUB.D is not data dependent on the two preceding instructions. The performance limitation created by this hazard could be eliminated by not requiring instructions to execute in program order.

To fulfill this aim, the issue process could be decomposed in two parts: checking for structural hazards, and waiting for the absence of a data hazard. Thus, *in-order* instruction issue is still used, but an instruction may start execution as soon as its data operands are available. A pipeline with these features performs *out-of-order* execution, which also implies *out-of-order* completion.

Dynamic scheduling - 3

Out-of-order execution introduces the possibility of WAR and WAW hazards, which did not exist in the classical 5-stage pipeline, the former, and only with multi-cycle floating-point operations that give rise to *out-of-order* completion, the latter.

Consider the code segment.

DIV.D	F0, F2, F4
ADD.D	F6, F0, F8
SUB.D	F8, F10, F14
MUL.D	F6, F10, F8

There is an *antidependence* between instructions ADD.D and SUB.D and an *output dependence* between instructions ADD.D and MUL.D. Both these hazards could be removed if *register renaming* is used.

Dynamic scheduling - 4

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order, *actually* do arise. Dynamically scheduled processors preserve exception behavior by delaying the notification of an associated exception until the processor knows that the involved instruction is the next one completed.

Although exception behavior is to be preserved, dynamically scheduled processors could generate *imprecise* exceptions. *Imprecise* exceptions can occur because of the two facts below

- the pipeline may have *already completed* some instructions that succeed in program order the instruction causing the exception and whose result cannot be reversed
- the pipeline may have *not yet completed* some instructions that precede in program order the instruction causing the exception.

The way to solve this problem and get precise exceptions, will be discussed later on in the context of *speculative* processors.

Dynamic scheduling - 5

To allow *out-of-order* execution, the ID stage of the classical 5-stage pipeline is split into two stages

- *issue* – instruction decoding and checking for structural hazards
- *read operands* – waiting until all data hazards are cleared before reading the operands.

An *instruction fetch* stage precedes the *issue* stage and the instruction that has been fetched is placed either into an *instruction register*, or into a *queue of pending instructions*; instructions are then issued from the register or the queue if conditions allow. The *execution* stage follows the *read operands* stage. Depending on the operation, the execution stage may take a variable number of cycles.

In this sense, the moment an instruction *begins* execution must be distinguished from the moment the instruction *completes* execution; the instruction is *in execution* between these two times. Having multiple instructions *in execution* at the same time requires multiple functional units, pipelined functional units or both. Since these two capabilities are essentially equivalent on what concerns pipeline control, it will be assumed the processor has multiple functional units.

Dynamic scheduling - 6

In a dynamically scheduled pipeline, all instructions pass through the *issue* stage *in-order*. They can, nevertheless, be stalled and overtaken by others at the *read operands* stage and enter in execution *out-of-order*.

There are two basic techniques that allow instructions to execute *out-of-order* when there are sufficient resources and no data dependencies among them: *scoreboarding* was the first technique to be introduced, it came up in the design of the CDC 6600 supercomputer in the middle of 1960s; *Tomasulo's algorithm* was the second, developed by Robert Tomasulo in 1967 and applied to the floating point unit of IBM 360/91.

The primary difference between them is that the *Tomasulo's algorithm* handles antidependences and output dependences by effectively renaming dynamically the registers. Additionally, it can also be extended to handle *speculation*, a technique aimed to reducing the effect of control dependences by predicting the outcome of a branch through the execution of instructions at the predicted target address and taking corrective actions when the prediction is wrong.

Scoreboarding - 1

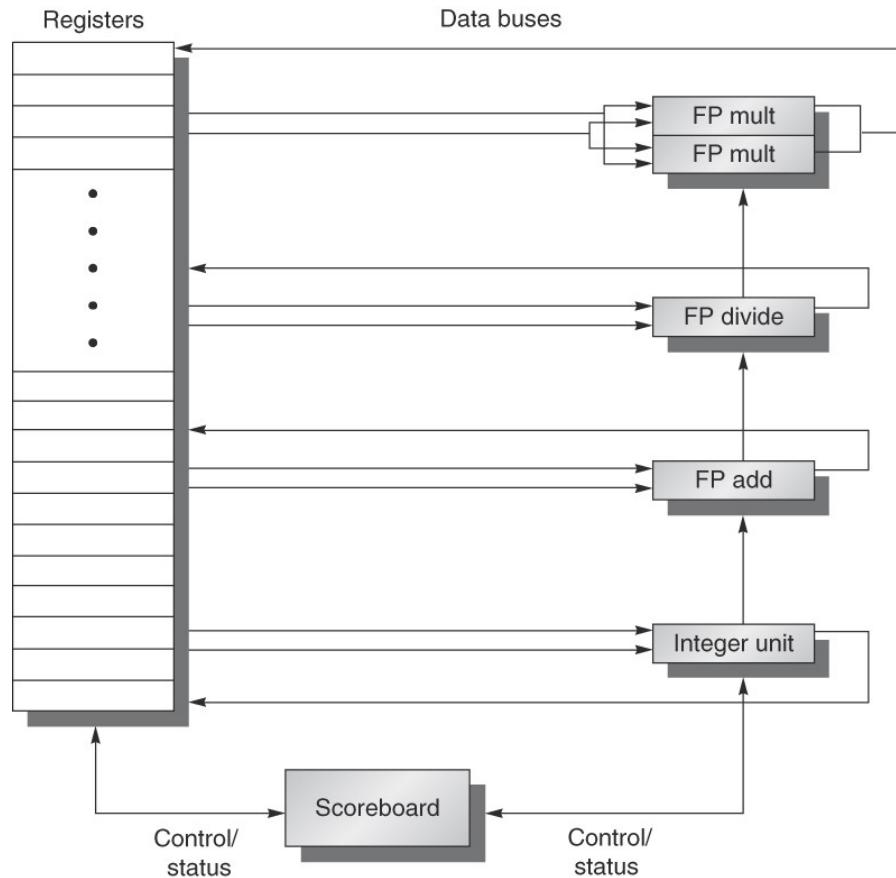
The goal of a *scoreboard* is to try to maintain an execution rate of one instruction per clock cycle, provided there are no structural hazards. Thus, instructions are executed as early as possible. When an instruction which has been issued is stalled, other instructions in the processing table, that do not depend on any active or stalled instruction, are looked up and if one is found, it will be executed. The *scoreboard* takes full control for instruction issue and execution, including all hazard detection.

On a processor with MIPS architecture, scoreboards makes sense primarily on the floating point unit, since the latency of the other functional units is very small. It will be assumed there are two multipliers, one adder, one divider and a single integer unit for all memory references, branches and integer operations.

Scoreboarding - 2

Basic organization of a MIPS processor with a scoreboard

Source: Computer Architecture: A Quantitative Approach



Scoreboarding - 3

Every instruction goes through four processing steps while under scoreboard control. This is a simplification of the real situation actually, since memory access, necessary for load and store operations, is being disregarded. The four steps, which replace ID, EX and WB in the standard pipeline, are as follows

1. *Issue* – if a FP functional unit of the required type is free and no other active instruction has the same register as destination, the scoreboard issues the instruction to the unit and updates its internal data structure – this step replaces the first half of the MIPS pipeline ID stage; by ensuring that no other active functional unit writes its result into the same destination register, it is guaranteed that no WAW hazards are present; if a structural or a WAW hazard exists, instruction issue stalls and no further instructions will issue until these hazards are cleared; when the issue stage stalls, it causes the buffer between instruction fetch and issue to fill and instruction fetch to stall right away, if the buffer is a single register, or when the buffer is full, if it is a queue.

Scoreboarding - 4

2. *Read operands* – the scoreboard monitors the availability of the source registers: a source register is available if no previous instruction, still in execution, is going to write it – this step replaces the second half of the MIPS pipeline ID stage; when the source operands are available, the scoreboard lets the functional unit to proceed to read the operands from the registers and begin execution; RAW hazards are resolved dynamically in this stage and instructions may be sent into execution out of order.
3. *Execution* – the functional unit starts execution upon receiving its operands; when the result is ready, it notifies the scoreboard that it has completed the operation – this step replaces the MIPS pipeline EX stage and takes a variable number of clock cycles in the MIPS FP pipeline.
4. *Write result* – once the scoreboard is aware the functional unit has terminated its operation, it checks for WAR hazards and, if required, prevents the instruction from completing – this step replaces the MIPS pipeline WB stage.

Scoreboarding - 5

At first glance, it might appear the scoreboard has difficulty in distinguishing between RAW and WAR hazards. Because the operands for an instruction are read only when the contents of both source registers has the updated value, the scoreboard does not take advantage of forwarding. This is not a large penalty since the instructions write their result to the destination register as soon as they complete their execution (provided there are no WAR hazards). The consequence is a reduced pipeline latency and the benefits of forwarding are attained in this indirect manner. There is, however, an extra clock cycle added to the latency because *write the result* and *read the operand* operations cannot overlap.

Based on its own internal data structure, the scoreboard controls instruction progression from one step to the next through communication with the functional units. There is still a small complication. Since the number of buses connecting the register bank and the functional units are limited, leading to potentially structural hazards, the scoreboard must guarantee that the number of functional units allowed to proceed into steps 2 and 4 does not exceed the number of buses available.

Scoreboarding - 6

The scoreboard *internal data structure* consists of three elements

- *instruction status* – it is a table with as many entries as the number of instructions under processing; for each instruction, it specifies which of the four states the instruction is in
- *functional unit status* – it is a table with as many entries as the number of functional units; each entry has nine fields
 - *busy* – it indicates whether the unit is busy or free
 - *op* – it indicates the operation to be performed in the unit
 - F_i – number of the destination register
 - F_j, F_k – numbers of the source registers
 - Q_j, Q_k – identification of the functional units whose result is to be stored in the source registers F_j and F_k
 - R_j, R_k – flags signaling when the source registers are ready to be read and have not yet been read
- *register result status* – it is a single entry table with as many fields as there are registers in the register bank; it indicates which functional unit will write the register if an active instruction has the register as its destination.

Scoreboarding - 7

Consider the code segment

L.D	F6, 34 (R2)
L.D	F2, 45 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

There are true data dependences between the first L.D instruction and the SUB.D and DIV.D instructions, between the second L.D instruction and the MUL.D, SUB.D and ADD.D instructions, between the MUL.D and the DIV.D instructions and between the SUB.D and the ADD.D instructions, potentially leading to RAW hazards. There is also an antidependence between the SUB.D and DIV.D instructions and the ADD.D instruction and an output dependence between the first L.D instruction and the ADD.D instruction, potentially leading to WAR and WAW hazards, respectively.

Assume the following latencies: load – 1 clock cycle, addition – 2 clock cycles, multiplication – 6 clock cycles and division – 12 clock cycles.

Scoreboarding - 8

Components of the scoreboard internal data structure when the second load instruction is about to write the result to the destination register

<i>Instruction status</i>									
<i>Instruction</i>	<i>Issue</i>	<i>Read operands</i>		<i>Execution terminated</i>		<i>Write result</i>			
L.D F6, 34 (R2)	yes		yes		yes			yes	
L.D F2, 45 (R3)	yes		yes		yes				
MUL.D F0, F2, F4	yes								
SUB.D F8, F6, F2	yes								
DIV.D F10, F0, F6	yes								
ADD.D F6, F8, F2									

<i>Functional unit status</i>									
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
integer	yes	load	F2	R3				no	
mult1	yes	mult	F0	F2	F4	integer		no	yes
mult2	no								
add	yes	sub	F8	F6	F2		integer	yes	no
divide	yes	div	F10	F0	F6	mult1		no	yes

<i>Register result status</i>									
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>	
mult1	integer			add	divide				

Scoreboarding - 9

Components of the scoreboard internal data structure when the multiplication instruction is about to write the result to the destination register

<i>Instruction status</i>					
<i>Instruction</i>	<i>Issue</i>	<i>Read operands</i>	<i>Execution terminated</i>	<i>Write result</i>	
L.D F6, 34 (R2)	yes	yes	yes	yes	
L.D F2, 45 (R3)	yes	yes	yes	yes	
MUL.D F0, F2, F4	yes	yes	yes		
SUB.D F8, F6, F2	yes	yes	yes		yes
DIV.D F10, F0, F6	yes				
ADD.D F6, F8, F2	yes	yes	yes		

<i>Functional unit status</i>									
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
integer	no								
mult1	yes	mult	F0	F2	F4			no	no
mult2	no								
add	yes	add	F6	F8	F2			no	no
divide	yes	div	F10	F0	F6	mult1		no	yes

<i>Register result status</i>									
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>	
mult1			add		divide				

Scoreboarding - 10

Components of the scoreboard internal data structure when the division instruction is about to write the result to the destination register

<i>Instruction</i>	<i>Issue</i>	<i>Read operands</i>	<i>Execution terminated</i>	<i>Write result</i>
L.D F6, 34 (R2)	yes	yes	yes	yes
L.D F2, 45 (R3)	yes	yes	yes	yes
MUL.D F0, F2, F4	yes	yes	yes	yes
SUB.D F8, F6, F2	yes	yes	yes	yes
DIV.D F10, F0, F6	yes	yes	yes	
ADD.D F6, F8, F2	yes	yes	yes	yes

<i>Functional unit status</i>									
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
integer	no								
mult1	no								
mult2	no								
add	no								
divide	yes	div	F10	F0	F6	mult1		no	no

<i>Register result status</i>								
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
					divide			

Scoreboarding - 11

Required checks and bookkeeping actions for each step in instruction execution when a scoreboard is used

<i>Instruction status</i>	<i>Wait until</i>	<i>Bookkeeping</i>
Issue	(busy[FU] == no) && (regResult[DR] == empty)	busy[FU] = yes; op[FU] = op; Fi[FU] = DR; Fj[FU] = SR1; FkFU] = SR2; Qj[FU] = regResult[SR1]; Qk[FU] = regResult[SR2]; Rj[FU] = (Qj[FU] == empty) ? yes : no; Rk[FU] = (Qk[FU] == empty) ? yes : no; regResult[DR] = FU;
Read operands	(Rj[FU] == yes) && (Rk[FU] == yes)	Qj[FU] = empty; Qk[FU] = empty; Rj[FU] = no; Rk[FU] = no;
Execution	FU operation terminated	
Write result	$\forall f ((Fj[f] \neq Fi[FU]) \parallel (Rj[f] == no)) \&\& ((Fk[f] \neq Fi[FU]) \parallel (Rk[f] == no))$	$\forall f \text{ if } (Qj[f] == FU) Rj[f] == yes;$ $\forall f \text{ if } (Qk[f] == FU) Rk[f] == yes;$ regResult[Fi[FU]] = empty; busy[FU] = no;

where **FU** is the functional unit associated with the instruction, **SR1** and **SR2** its source registers and **DR** its destination register. The test at the write result stage is carried out to prevent WAR hazards.

Scoreboarding - 12

A scoreboard uses the available ILP to minimize the number of stalls arising from the program true data dependences. In eliminating stalls, a scoreboard is limited by several factors

- *the amount of parallelism inherent to the program* – this factor determines whether independent instructions can be found; if every instruction depends on its predecessor, no dynamic scheduling scheme can reduce the number of stalls
- *the number of scoreboard entries* – this factor determines how far ahead can the pipeline look for independent instructions
- *the number and types of functional units* – this factor determines how relevant the structural hazards are, which may increase when dynamic scheduling is used
- *the presence of antidependences and output dependences* – which leads to WAR and WAW hazards and can generate further stalls.

Tomasulo's algorithm - 1

In the scheme developed by Robert Tomasulo, RAW hazards are avoided by executing an instruction only when its operands are available, which is exactly what the simpler scoreboard provides. WAR and WAW hazards, which arise from name dependencies, are eliminated by register renaming. *Register renaming*, in fact, eliminates these hazards by altering the name of all destination registers, including those with a pending read or write from an earlier instruction, so that the out-of-order write does not affect any instructions that depend on the older value of the operand.

Tomasulo's algorithm - 2

To better understand how register renaming works, consider the following code sequence

DIV.D	F0, F2, F4
ADD.D	F6, F0, F8
S.D	F6, 0 (R1)
SUB.D	F8, F10, F14
MUL.D	F6, F10, F8

There are antidependences between the ADD.D instruction and the SUB.D and between the S.D instruction and the MUL.D instruction, and an output dependence between the ADD.D and the MUL.D instruction, potentially leading to WAR and WAW hazards, respectively. There are also true data dependences between the DIV.D instruction and the ADD.D instruction, between the ADD.D instruction and the S.D instruction and between the SUB.D instruction and the MUL.D instruction.

Tomasulo's algorithm - 3

The three name dependences can all be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, S and T. The code sequence can be rewritten without any name dependences as

DIV.D	F0, F2, F4
ADD.D	S, F0, F8
S.D	S, 0 (R1)
SUB.D	T, F10, F14
MUL.D	F6, F10, T

In addition, any subsequent uses of register F8 must be replaced by register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of register F8 that appear later in the code, requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the code segment and a later use of F8.

Tomasulo's algorithm - 4

In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to be executed. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register of the register bank. Pending instructions designate the reservation buffer that will provide their input. Finally when successive writes to the same register overlap in execution, only the last one in program order will actually update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation stations which provide the updated values.

Since there can be more reservation stations than real registers, this approach can even eliminate hazards arising from name dependences that could not be taken care of by the compiler.

Tomasulo's algorithm - 5

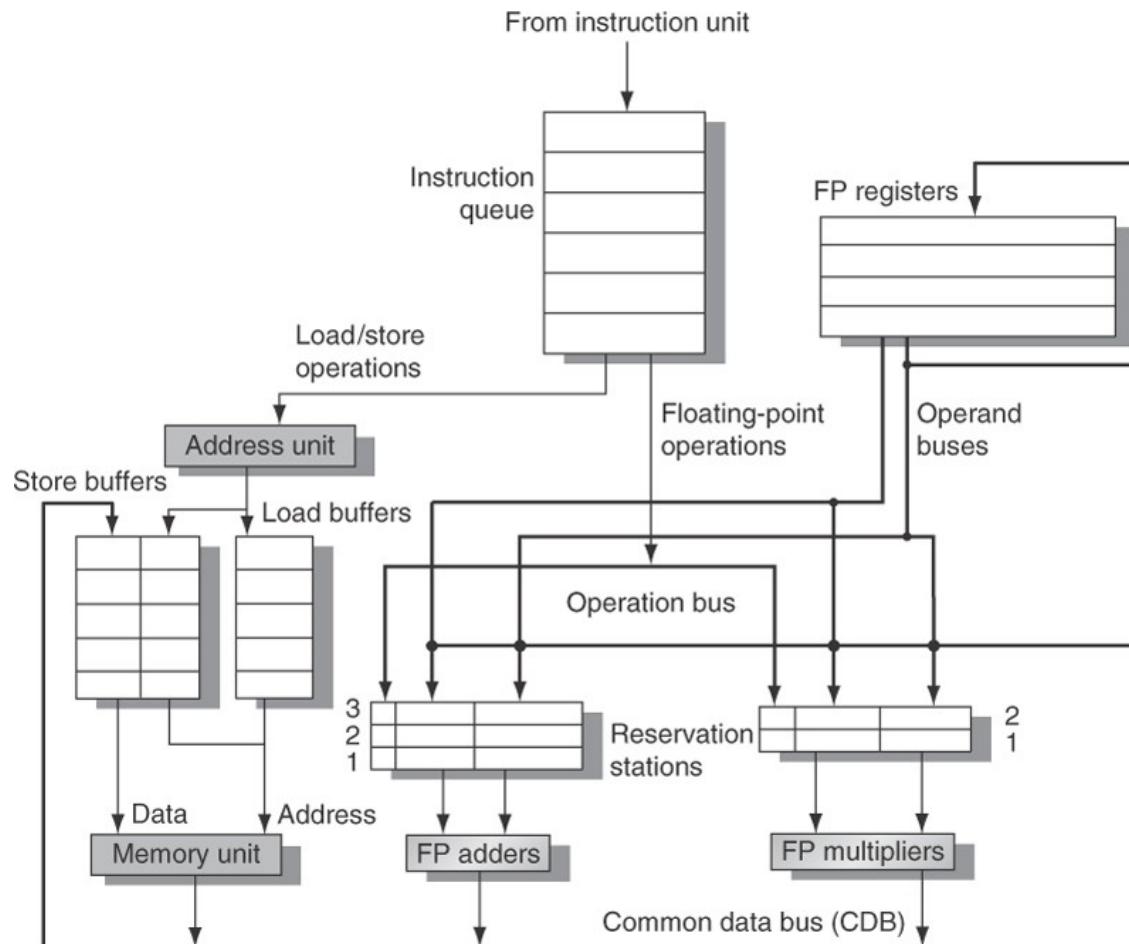
The use of reservations stations, rather than a centralized register bank, leads to two important properties

- *hazard detection and hazard control are distributed* – the information held in the reservation stations at each functional unit determines when an instruction can begin execution at that unit
- *results are passed directly to the functional units from the reservation stations where they are buffered, rather than going through the registers of the register bank* – this bypassing is done using a common result bus that allows all units waiting for an operand to be loaded simultaneously (the bus is called *common data bus* in the IBM 360/91); in pipelines with multiple execution units and issuing multiple instructions per clock cycle, more than one result bus is needed.

Tomasulo's algorithm - 6

Basic organization of a MIPS floating point unit using the Tomasulo's algorithm

Source: Computer Architecture: A Quantitative Approach



Tomasulo's algorithm - 7

Each reservation station holds both an instruction that has been issued and is waiting for execution at the associated functional unit, and either the operand values for that instruction, if they are available, or the names of the reservation stations that will provide the operand values once they are computed.

The load and store buffers hold addresses and data transferred from or to memory and behave almost exactly like the reservation stations, so they will be distinguished only when necessary. In particular, load and store buffers have three functions

- they hold the full source or destination address after its computation by the *address unit*
- they track outstanding loads that are waiting for transfer completion
- they control the results of complete loads that are waiting for the availability of the CDB, or hold the value to be stored until the *memory unit* is available.

All the results from the functional and the memory units are put in the common data bus, which connects everything but the load buffers. All reservation stations and load and store buffers have tag fields employed by the pipeline control.

Tomasulo's algorithm - 8

Every instruction goes through three processing steps, although each one may take an arbitrary number of clock cycles. The three steps, which replace ID, EX and WB in the standard pipeline, are as follows

1. *Issue* – the next instruction is obtained from the head of the instruction queue, which is maintained in FIFO order to ensure the correct data flow. If a matching reservation station or buffer is free, the instruction is issued to the station or buffer together with the operand values, if they are currently saved in registers of the register bank. Otherwise, tracking of the functional units which will produce the operand values is performed. It is here that the registers are renamed, eliminating in consequence WAR and WAW hazards. This stage is sometimes called *dispatch*, within the context of dynamically scheduled processors.

Tomasulo's algorithm - 9

2. *Execute* – If one or both operands are unavailable, the common data bus is monitored waiting for it or them to be computed. As soon as an operand becomes available, it is stored both at the register bank and at any station or store buffer requiring it. When all the operands are available, the operation can be executed at the corresponding functional or memory unit. By delaying instruction execution until all the operands are available, RAW hazards are avoided.

Several instructions can become ready in the same clock cycle. Although independent functional units can start execution at the same clock cycle, if more than one instruction is ready to execute at the same functional unit, a selection has to be made among them. Floating point reservation stations may be chosen arbitrarily.

Load and store instructions, however, are maintained in program order through the use of a load/store queue that contains the identification of the buffer being used. Their execution require a two-step process: in step 1, the effective address is computed; in step 2, the access to memory for data transfer is carried out.

Tomasulo's algorithm - 10

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede it in program order have completed. This restriction guarantees that an instruction that causes an exception during execution, will have been really executed. In a processor using branch prediction, as all dynamically scheduled processors do, this means that it must be known the branch prediction is correct, before allowing an instruction after the branch in program order to start execution. If the processor records the occurrence of an exception, but does not actually raise it, an instruction can start execution, but not stall until it reaches the write result step.

3. *Write result* – When the operation result is available, the CDB is accessed to save it in a register of the register bank and in the reservation stations, including store buffers, that are waiting for it.

Tomasulo's algorithm - 11

The data structures that detect and eliminate hazards are attached to the reservation stations, to the register bank and to the load and store buffers, with slightly different information attached to each type. These tags are essentially names for the extended set of virtual registers used in the renaming process.

In the illustration shown, the tag field is a 4-bit quantity that denotes one of the five reservation stations (three FP adders for addition and subtraction and two FP multipliers for multiplication and division) or one of the five load buffers. This produces the equivalent of ten registers that can be designated as *result registers*.

The tag field describes which reservation station, or load buffer, contains the instruction that will produce a result needed as a source operand. Once an instruction is issued and is waiting for a source operand, it denotes the operand through the reservation station number, or load buffer number, where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in a register of the register bank. The fact that there are more reservation stations, or load buffers, than actual register numbers, makes the elimination of WAR and WAW hazards trivial.

Tomasulo's algorithm - 12

In Tomasulo's scheme, results are broadcast in a bus which is monitored by the reservation stations and the store buffers for data retrieval. This kind of arrangement implements the forwarding and bypassing mechanisms used in a statically scheduled pipeline. In doing so, however, a dynamically scheduled scheme adds one latency clock cycle between source and result, since the matching of a result and its use cannot be done until the *write result* stage. Thus, in a dynamically scheduled pipeline, the effective latency between a producing and a consuming instruction is at least one clock cycle longer than the latency of the functional unit producing the result.

It is important to remember that the tags in the Tomasulo's algorithm refer to the unit or the buffer that produces a result: the register names are discarded once an instruction is issued. This is, in fact, a key difference between Tomasulo's scheme and scoreboardding. In scoreboardding, operands stay in the registers of the register bank and are only read after the producing instructions completes and the consuming instruction is ready to execute.

Tomasulo's algorithm - 13

Each *reservation station* or *buffer* has seven fields

- *busy* – it indicates whether the reservation station or buffer is occupied or free
- *op* – it indicates the operation to be performed in the unit
- Q_j, Q_k – identification of the reservation station, or buffer, that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in registers V_j and V_k , or is unnecessary
- V_j, V_k – the value of the source operands; notice that only one of the fields, Q or V , is valid for each operand; for load buffers, the V_k field is used to hold the offset field
- A – used to hold the effective memory address for a load or store instruction.

Each *register* of the register bank has one field

- Q_i – identification of the reservation station, or buffer, that will produce the result to be stored in the register; a value of zero indicates that no active instruction is computing a value to be stored in the register.

Tomasulo's algorithm - 14

Consider the code sequence

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
SUB.D	F8, F2, F6
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

What is the execution state when only the first load instruction has completed and written its result?

Assume the following latencies: load / store – 1 clock cycle, addition – 2 clock cycles, multiplication – 6 clock cycles and division – 12 clock cycles.

Tomasulo's algorithm - 15

Reservation stations / buffers and register tags when only the first load instruction has completed and written its result

<i>Instruction status</i> (not part of the hardware)			
<i>Instruction</i>	<i>Issue</i>	<i>Execute</i>	<i>Write result</i>
L.D F6, 32 (R2)	yes	yes	yes
L.D F2, 44 (R3)	yes	yes	
MUL.D F0, F2, F4	yes		
SUB.D F8, F2, F6	yes		
DIV.D F10, F0, F6	yes		
ADD.D F6, F8, F2	yes		

<i>Reservation stations / buffers</i>							
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>	<i>A</i>
load1	no						
load2	yes	load					44+reg[R3]
add1	yes	sub		mem[32+reg[R2]]	load2		
add2	yes	add			add1	load2	
add3	no						
mult1	yes	mult		reg[F4]	load2		
mult2	yes	div		mem[32+reg[R2]]	mult1		

<i>Register status</i>									
<i>Field</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Qi	mult1	load2		add2	add1	mult2			

Tomasulo's algorithm - 16

Reservation stations / buffers and register tags when the multiplication instruction is ready to write its result

<i>Instruction status</i> (not part of the hardware)			
<i>Instruction</i>	<i>Issue</i>	<i>Execute</i>	<i>Write result</i>
L.D F6, 32 (R2)	yes	yes	yes
L.D F2, 44 (R3)	yes	yes	yes
MUL.D F0, F2, F4	yes	yes	
SUB.D F8, F2, F6	yes	yes	yes
DIV.D F10, F0, F6	yes		
ADD.D F6, F8, F2	yes	yes	yes

<i>Reservation stations / buffers</i>							
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>	<i>A</i>
load1	no						
load2	no						
add1	no						
add2	no						
add3	no						
mult1	yes	mult	mem[44+reg[R3]]	reg[F4]			
mult2	yes	div		mem[32+reg[R2]]	mult1		

<i>Register status</i>									
<i>Field</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Qi	mult1						mult2		

Tomasulo's algorithm - 17

Required checks and bookkeeping actions for each step in instruction execution when Tomasulo's algorithm is used

<i>Instruction status</i>	<i>Wait until</i>	<i>Bookkeeping</i>
Issue FP operation	(RS[r].busy == no)	<pre> if(regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if(regStat[rt].Qi != 0) RS[r].Qk = regStat[rt].Qi; else { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } regStat[rd].Qi = r; RS[r].busy = yes;</pre>
Issue load	(RS[r].busy == no)	<pre> if(regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } RS[r].A = imm; regStat[rt].Qi = r; insert r into the load-store queue; RS[r].busy = yes;</pre>
Issue store	(RS[r].busy == no)	<pre> if(regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } RS[r].A = imm; if(regStat[rt].Qi != 0) RS[r].Qk = regStat[rt].Qi; else { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } insert r into the load-store queue; RS[r].busy = yes;</pre>

Tomasulo's algorithm - 18

Required checks and bookkeeping actions for each step in instruction execution when Tomasulo's algorithm is used (continuation)

Execution FP operation	$(RS[r].Qj == 0) \&\& (RS[r].Qk == 0)$	compute result – operands are in V_j and V_k
Execution load / store step 1	$(RS[r].Qj == 0) \&\&$ r is the head of load-store queue	$RS[r].A = RS[r].V_j + RS[r].A;$
Execution load step 2	load step 1 is complete	read from $mem[RS[r].A]$
Write result FP operation or load	r has completed execution $\&\&$ CDB is available	$\forall x \left(\begin{array}{l} \text{if } (regStat[x].Qi == r) \\ \quad \{ reg[x] = result; regStat[x].Qi = 0; \} \end{array} \right)$ $\forall x \left(\begin{array}{l} \text{if } (RS[x].Qj == r) \\ \quad \{ RS[x].V_j = result; RS[x].Qj = 0; \} \end{array} \right)$ $\forall x \left(\begin{array}{l} \text{if } (RS[x].Qk == r) \\ \quad \{ RS[x].V_k = result; RS[x].Qk = 0; \} \end{array} \right)$ $RS[r].busy = no;$
Write result store	r has completed execution $\&\&$ $(RS[r].Qk == 0)$	$mem[RS[r].A] = RS[r].V_k;$ $RS[r].busy = no;$

where $RS[r]$ is the reservation station / buffer associated with the instruction and $regStat$ is the register status.

Tomasulo's algorithm - 19

A loop is considered next to illustrate the full power of eliminating WAW and WAR hazards through register renaming (the effects of delayed branches are ignored)

Loop:	L.D	F0, 0(R1)
	MUL.D	F4, F0, F2
	S.D	F4, 0(R1)
	DADDIU	R1, R1, -8
	BNE	R1, R2, Loop

It is assumed that the register R1 contains initially the address of the last array element and that the contents of the register R2+8 is the address of the first array element.

If the branch is predicted *taken*, multiple executions of the loop can proceed in parallel through the use of multiple reservation stations and load and store buffers. This feature is gained without any code change since the loop is unrolled dynamically by the hardware. The reservation stations and load and store buffers play the role of additional registers.

Assume the same instruction latencies as in the last example.

Tomasulo's algorithm - 20

Reservation stations / buffers and register tags when two successive loop iterations have been issued but no instruction has yet completed

<i>Instruction status (not part of the hardware)</i>				
<i>Instruction</i>	<i>Loop Iteration</i>	<i>Issue</i>	<i>Execute</i>	<i>Write result</i>
L.D F0, 0 (R1)	i	yes	yes	
MUL.D F4, F0, F2	i	yes		
S.D F4, 0 (R1)	i	yes		
L.D F0, 0 (R1)	i+1	yes	yes	
MUL.D F4, F0, F2	i+1	yes		
S.D F4, 0 (R1)	i+1	yes		

<i>Reservation stations / buffers</i>							
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>	<i>A</i>
load1	yes	load					reg[R1]+0
load2	yes	load					reg[R1]-8
mult1	yes	mult		reg[F2]	load1		
mult2	yes	mult		reg[F2]	load2		
store1	yes	store	reg[R1]+0			mult1	
store2	yes	store	reg[R1]-8			mult2	

<i>Register status</i>									
<i>Field</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Qi	load2		mult2						

Tomasulo's algorithm - 21

When extended to multiple instruction issue, Tomasulo's approach can sustain more than one instruction per clock cycle. Loads and stores can be safely performed out-of-order provided they access different memory addresses. When they refer to the same address, then either

- the load is *before* the store in program order and interchanging their execution results in a WAR hazard, or
- the load is *after* the store in program order and interchanging their execution results in a RAW hazard.

Similarly, interchanging the execution of two stores to the same memory address results in a WAW hazard.

Hence, to determine if a load instruction can be executed, the processor must check whether there is any uncompleted store instruction preceding the load in program order and accessing the same address. Similarly, a store instruction must wait until there are no unexpected loads and stores that precede it and access the same address.

Tomasulo's algorithm - 22

To prevent the data hazards to occur, the processor must have computed the effective address associated with any earlier memory operation still in progress. This can be done in a simple, but not necessarily optimal, way by performing all the effective address calculations in program order. (In fact, one only needs to keep the relative order between stores and other memory references, i. e., the load operations may be freely reordered).

If a load operation comes first, the existence of an address conflict can be readily checked by examining the A field of all active store buffers. A conflict being detected, the load instruction is not issued until the conflicting store completes.

In case of store instructions, the procedure is similar except that the processor must check for conflicts both the load and store buffers, since conflicting stores can not be reordered with respect to either loads and stores.

Tomasulo's algorithm – 23

Tomasulo's scheme was not used for many years after its application to the IBM 360/91 floating point unit, but it became very popular for multiple-issue processors, starting in the 1990s, for several reasons

- although the algorithm was designed before the cache era, the presence of caches with inherently unpredictable delays, turned to be one of the major motivations for dynamic scheduling because *out-of-order* execution allows the processor to keep executing instructions while waiting for the completion of a cache miss, thus hiding all or part of the penalty
- as the processors become more aggressive on their issue capability and designers more concerned with the performance of difficult-to-schedule code, such as most non-numeric code is, the application of techniques like *register renaming*, *dynamic scheduling* and *speculation* become more relevant
- through its application, one can achieve high-performance without requiring the compiler to target the generated code to a specific pipeline structure.

Hardware-based speculation - 1

Maintaining control dependences becomes an increasing burden when one tries to exploit instruction-level parallelism further. Branch prediction reduces the direct stalls attributable to branches, but just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism for a processor executing multiple instructions per clock cycle. A wide-issue processor may need to execute a branch every clock cycle to keep performance at a peak value. Hence, exploiting more parallelism requires that the limitation of control dependence be overcome.

Overcoming control dependence is achieved by *speculating* on the outcome of branches and executing the code as if the guess were correct. This idea represents a subtle, but crucial, extension over branch prediction with dynamic scheduling: with *speculation*, instructions are fetched, issued and executed as if branch predictions were always right; dynamic scheduling only fetches and issues such instructions. Of course, mechanisms are needed to handle the situation where the speculation turns to be incorrect.

Hardware-based speculation - 2

Hardware-based speculation combines three key ideas

- *dynamic branch prediction*, to select which instructions to execute
- *speculation*, to allow the execution of instructions before the control dependencies are resolved (with the understanding that the effects produced by an incorrect speculated sequence can be undone)
- *dynamic scheduling*, to deal with the issuing of different combinations of basic blocks.

In contrast, dynamic scheduling [without speculation] overlaps basic blocks only partially, because it requires a branch to be resolved before actually executing any instructions in the successor block.

Hence, hardware-based speculation follows the predicted flow of data values to determine when to execute the instructions. This method of execution is essentially a *data flow execution*: operations are carried out as soon as their source operands become available.

Hardware-based speculation - 3

In order to extend Tomasulo's algorithm to support speculation, the bypassing of results among instructions, which is needed to execute instructions speculatively, has to be separated from the actual completion of instructions. If such a mechanism is accomplished, an instruction can be allowed to execute and bypass its result to others, without allowing the instruction to perform any updates that can not be reversed until it is established that the instruction is no longer speculative.

Using a bypassed value is like performing a speculative register read, since it may not be known at that moment whether the instruction providing the value for the [source] register is providing a *real* result. Only when the instruction is no longer speculative, the specific register of the register bank, or the referred memory location, may be updated – this additional step in instruction execution is called *instruction commit*.

Therefore, instructions can execute *out-of-order*, but are forced to commit *in-order* so that any irreversible action, such as state updating or exception taking, is prevented. The separation of instruction completion from instruction commit is essential because instructions may finish execution considerably before they are ready to commit.

Hardware-based speculation - 4

The introduction of the commit phase to instruction execution requires an additional set of hardware buffers which hold the results of completed instructions that have not committed yet. This buffer bank, called the *reorder buffer* (ROB), is also used to pass results among instructions.

The *reorder buffer* provides additional registers in just the same way as the reservation stations and the load and the store buffers extend the register set in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will get the value from the register bank, while with speculation, the register bank is not updated until the instruction commits (that is, when it is definitively established that the instruction should execute), which means that ROB supplies operands to other instructions in the interval between instruction completion and instruction commit. ROB is similar to the store buffers in Tomasulo's algorithm, so the functionality of store buffers is integrated into ROB for simplicity.

Hardware-based speculation - 5

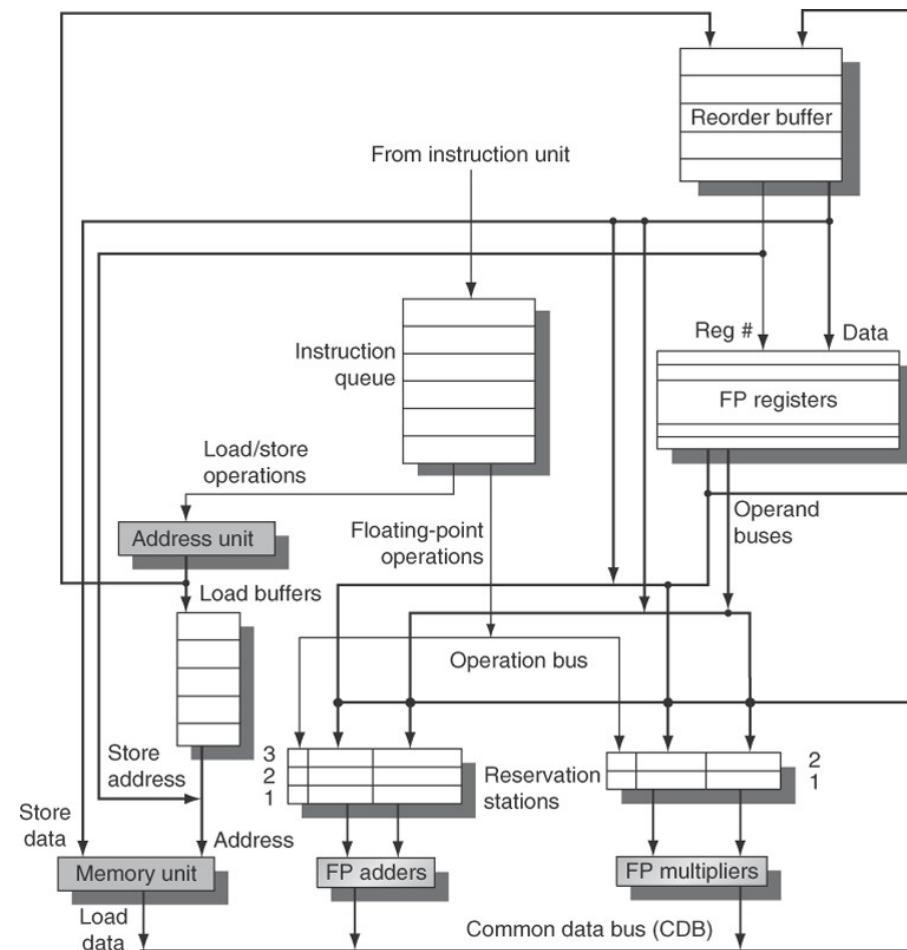
Each ROB entry contains five fields

- *busy* – it indicates whether the entry is occupied or free
- *instruction type* – it signals whether the instruction is a branch, with no destination result, a store, with a memory destination, or a load or ALU operation, with a register bank destination
- *destination location* - it supplies the memory address, for stores, or the register number, for loads and ALU operations, where the instruction result should be written to
- *value* – it holds the value of the instruction result between instruction completion and instruction commit
- *ready* - it indicates whether the instruction has completed execution.

Hardware-based speculation - 6

Basic organization of a MIPS floating point unit using the Tomasulo's algorithm extended to handle speculation

Source: Computer Architecture: A Quantitative Approach



Hardware-based speculation - 7

Stores still execute in two steps, but the second step is included in instruction commit.

Although the renaming function of the reservation stations is replaced by the ROB, a place is still needed to buffer operations and operands between the time they issue and the time they end execution. This function is carried out by the reservation stations and the load buffers. On the other hand, since every instruction has an entry in ROB until it commits, the operation result is tagged using the ROB entry rather than the reservation station or the load buffer number. This tagging requires that the ROB entry assigned to the instruction is tracked at the reservation station or load buffer. Later on, an alternative implementation, which uses extra registers for renaming and a queue to replace ROB, is described and shown how it can decide when the instructions can commit.

Hardware-based speculation - 8

The four steps involved in instruction execution are as follows

1. *Issue* – the next instruction is obtained from the head of the instruction queue, which is maintained in FIFO order to ensure the correct data flow. If both a matching reservation station or buffer *and* a ROB slot are free, the instruction is issued to the station or buffer together with the operand values, if they are currently saved in registers of the register bank or in ROB entries. Otherwise, tracking of the ROB entries which will eventually contain the operand values is performed. The number of the ROB entry allocated to the instruction is also sent to the reservation station or buffer so that the number can be used to tag the result when it is placed on the CDB.
2. *Execute* – the CDB is monitored while waiting for the operands to be computed so that RAW hazards are prevented. When the operands are available, the operation is executed. Instruction execution may take multiple clock cycles. Loads still require two steps. Stores, on the other hand, only compute the effective address.

Hardware-based speculation - 9

3. *Write result* – when the instruction completes, the result is placed on the CDB, together with the ROB entry tag, to be written into the ROB and into all the reservation stations and buffers waiting for the value. The reservation station or buffer is marked free and the *ready* field of the ROB entry is set. Special actions are required for store instructions: if the value to be stored is available, it is written into the *value* field of the ROB entry; if the value is unavailable, the CDB is monitored for the broadcast of the value, it is saved in the buffer and then the *value* field of the ROB entry is updated.
4. *Commit* – the associated set of actions depend upon whether the committing instruction is a branch with a wrong prediction, a store or any other case (*normal commit*). The *normal commit* occurs when the instruction attains the head of ROB and the *ready* field is set, the processor then updates the register of the register bank, if any, and finishes. Committing a store is similar, except that a memory location is updated instead of a register. Finally, for a branch with a wrong prediction, the speculation is found to be incorrect, ROB is flushed and fetching is restarted at the proper address defined by the branch behavior.

Hardware-based speculation - 10

Consider the code sequence (identical to the one used in the example for the Tomasulo's algorithm and assume the same instruction latencies)

L.D	F6, 32 (R2)
L.D	F2, 44 (R3)
MUL.D	F0, F2, F4
S.D	F8, F2, F6
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

What is the execution state when the multiplication instruction is ready to commit, that is, the MULT instruction is at the head of ROB?

Hardware-based speculation - 11

Reorder buffer, reservation stations / buffers and register tags when the multiplication instruction is ready to commit

<i>Reorder buffer</i>						
<i>entry</i>	<i>state</i>	<i>busy</i>	<i>instruction type</i>	<i>destination location</i>	<i>value</i>	<i>ready</i>
1	commit	no	L.D F6, 32 (R2)	F6	mem[reg[R2]+32]	yes
2	commit	no	L.D F2, 44 (R3)	F2	mem[reg[R3]+44]	yes
3	write result	yes	MUL.D F0, F2, F4	F0	#2 x reg[F4]	yes
4	write result	yes	SUB.D F8, F2, F6	F8	#2 - #1	yes
5	execute	yes	DIV.D F10, F0, F6	F10		no
6	write result	yes	ADD.D F6, F8, F2	F6	#4 + #2	yes

<i>Reservation stations / buffers</i>								
<i>Name</i>	<i>busy</i>	<i>op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>	<i>dest</i>	<i>A</i>
load1	no							
load2	no							
add1	no							
add2	no							
add3	no							
mult1	no	mult	mem[reg[R3]+44]	reg[F4]			#3	
mult2	yes	div		mem[reg[R2]+32]	#3		#5	

<i>Register status</i>									
<i>Field</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
busy	yes	no	no	yes	yes	yes	no	...	no
ROB entry	3			6	4	5		...	

Hardware-based speculation - 12

Consider the following code sequence, identical to the one used in the example for the Tomasulo's algorithm (the effects of delayed branches are ignored)

Loop:	L.D	F0, 0 (R1)
	MUL.D	F4, F0, F2
	S.D	F4, 0 (R1)
	DADDIU	R1, R1, -8
	BNE	R1, R2, Loop

It is assumed that the register R1 contains initially the address of the last array element and that the contents of the register R2+8 is the address of the first array element.

If the branch is predicted *taken*, multiple executions of the loop can proceed in parallel. Assume that all the instructions in the loop have been issued twice, the load and the multiplication instructions of the first iteration have committed and all the others have completed execution.

Also assume the same instruction latencies as in the last example.

Hardware-based speculation - 13

Reorder buffer and register tags when two successive loop iterations have been issued, the load and multiplication instructions of the first iteration have committed and all the others have completed execution

Reorder buffer						
<i>entry</i>	<i>state</i>	<i>busy</i>	<i>instruction type</i>	<i>destination location</i>	<i>value</i>	<i>ready</i>
1	commit	no	L.D F0, 0(R1)	F0	mem[reg[R1]+0]	yes
2	commit	no	MUL.D F4, F0, F2	F4	#1 x reg[F2]	yes
3	write result	yes	S.D F4, 0(R1)	reg[R1]+0	#2	yes
4	write result	yes	DADDIU R1, R1, -8	R1	reg[R1] - 8	yes
5	write result	yes	BNE R1, R1, Loop			yes
6	write result	yes	L.D F0, 0(R1)	F0	mem[#4]	yes
7	write result	yes	MUL.D F4, F0, F2	F4	#6 x reg[F2]	yes
8	write result	yes	S.D F4, 0(R1)	#4+0	#7	yes
9	write result	yes	DADDIU R1, R1, -8	R1	#4 - 8	yes
10	write result	yes	BNE R1, R1, Loop			yes

Register status									
<i>Field</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
busy	yes	no	yes	no	no	no	no	...	no
ROB entry	6		7					...	

Hardware-based speculation - 14

Since neither register values, nor memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. The instructions prior to the branch will commit in succession when each reaches the head of ROB. When the turn of the branch comes, ROB is flushed and the processor starts to fetch instructions from the correct path.

In speculative processors, performance is very sensitive to branch prediction. Thus, all aspects of handling branches such as prediction accuracy, latency of misprediction detection and misprediction recovery time, become very important.

Exceptions are handled by not recognizing the exception until the instruction that produced it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the associated ROB entry. When a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with ROB. If the instruction reaches the head of ROB, it is no longer speculative and the exception is now taken.

Hardware-based speculation - 15

Required checks and bookkeeping actions for each step in instruction execution when hardware-based speculation is used

<i>Instruction status</i>	<i>Wait until</i>	<i>Bookkeeping</i>
Issue FP operation	(RS[r].busy == no) && (ROB[b].busy == no)	<pre> if(regStat[rs].busy == yes) { x = regStat[rs].reorder; if(ROB[x].ready == yes) { RS[r].Vj = ROB[x].value; RS[r].Qj = 0; } else RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if(regStat[rt].busy == yes) { x = regStat[rt].reorder; if(ROB[x].ready == yes) { RS[r].Vj = ROB[x].value; RS[r].Qk = 0; } else RS[r].Qk = x; } else { RS[r].Vj = reg[rt]; RS[r].Qk = 0; } regStat[rd].reorder = b; regStat[rd].busy = yes; RS[r].dest = b; RS[r].busy = yes; ROB[b].inst = opcode; ROB[b].dest = rd; ROB[b].ready = no; ROB[b].busy = yes; </pre>
Issue load	(RS[r].busy == no) && (ROB[b].busy == no)	<pre> if(regStat[rs].busy == yes) { x = regStat[rs].reorder; if(ROB[x].ready == yes) { RS[r].Vj = ROB[x].value; RS[r].Qj = 0; } else RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } regStat[rt].reorder = b; regStat[rt].busy = yes; RS[r].A= imm; RS[r].dest = b; RS[r].busy = yes; ROB[b].inst = opcode; ROB[b].dest = rt; ROB[b].ready = no; ROB[b].busy = yes; </pre>

Hardware-based speculation - 16

Required checks and bookkeeping actions for each step in instruction execution when hardware-based speculation is used (continuation)

Issue store	(RS[r].busy == no) && (ROB[b].busy == no)	<pre> if(regStat[rs].busy == yes) { x = regStat[rs].reorder; if(ROB[x].ready == yes) { RS[r].Vj = ROB[x].value; RS[r].Qj = 0; } else RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if(regStat[rt].busy == yes) { x = regStat[rt].reorder; if(ROB[x].ready == yes) { RS[r].Vk = ROB[x].value; RS[r].Qk = 0; } else RS[r].Qk = x; } else { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } RS[r].A= imm; RS[r].busy = yes; ROB[b].inst = opcode; ROB[b].ready = no; ROB[b].busy = yes; </pre>
Execution FP operation	(RS[r].Qj == 0) && (RS[r].Qk == 0)	compute result – operands are in Vj and Vk
Execution load step 1	(RS[r].Qj == 0) && there are no previous stores in ROB queue	RS[r].A= RS[r].Vj + RS[r].A;
Execution load step 2	load step 1 is complete && all previous stores in ROB queue have different effective addresses	read from mem[RS[r].A]
Execution store	(RS[r].Qj == 0) && b is the head of ROB queue	ROB[b].dest = RS[r].Vj + RS[r].A;

Hardware-based speculation - 17

Required checks and bookkeeping actions for each step in instruction execution when hardware-based speculation is used (continuation)

Write result FP operation or load	r has completed execution && CDB is available	<pre>b = RS[r].dest; forall x (if (RS[x].Qj == b) { RS[x].Vj = result; RS[x].Qj = 0; }) forall x (if (RS[x].Qk == b) { RS[x].Vk = result; RS[x].Qk = 0; }) ROB[b].value = result; ROB[b].ready = yes; RS[r].busy = no;</pre>
Write result store	r has completed execution && (RS[r].Qk == 0)	<pre>ROB[b].value = RS[r].Vk; RS[r].busy = no;</pre>
Commit	b is the head of ROB queue && (ROB[b].ready == yes)	<pre>if (ROB[b].inst == branch) { if (mispredicted branch) { flush ROB; reset reservation stations / load buffers; reset register status; fetch at branch destination; } } else if (ROB[b].inst == store) mem[ROB[b].dest] = [ROB[b].value]; else { reg[ROB[b].dest] = ROB[b].value; if (regStat[ROB[b].dest].reorder == b) regStat[ROB[b].busy = no; } ROB[b].busy = no;</pre>

where ROB [b] is the ROB entry and RS [r] the reservation station / buffer associated with the instruction and regStat is the register status.

Exploiting ILP using multiple issue - 1

The techniques just described can be used to eliminate stalls resulting from data and control dependences and approach an ideal CPI of one when running a given program

$$\text{CPI}_{\text{prog}} = \text{CPI}_{\text{ideal}} + \text{structural stalls} .$$

To improve performance further, one needs to decrease the ideal CPI to a value less than one, but this can not be done if only one instruction is issued per clock cycle.

Multiple issue processors are of three basic types

- *statically scheduled superscalar processors* – a variable number of instructions are issued together in multiple pipelines, each statically scheduled
- *VLIW (very long instruction word) processors* – a fixed number of instructions formatted either as one large instruction, or a fixed instruction packet, are issued together
- *dynamically scheduled superscalar processors* – a variable number of instructions are issued together in multiple pipelines, each dynamically scheduled.

Exploiting ILP using multiple issue - 2

Primary approaches in use for multiple issue processors

Source: Computer Architecture: A Quantitative Approach

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
superscalar (static)	dynamic	hardware	static	in-order execution	mostly in embedded environments: MIPS and ARM (including ARM Cortex-A8)
superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution, but no speculation	none at the present time
superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Intel Core i3, i5, i7, AMD Phenom and IBM POWER 7
VLIW / LIW	static	primarily software	static	all hazards determined and indicated by the compiler (often implicitly)	mostly in signal processing: TI C6x
EPIC	primarily static	primarily software	mostly static	all hazards determined and indicated by the compiler (explicitly)	Itanium

Statically scheduled superscalar processor

A *statically scheduled superscalar processor* typically issues in-order a variable number of instructions per clock cycle up to an upper limit that corresponds to the number of parallel pipelines which are implemented.

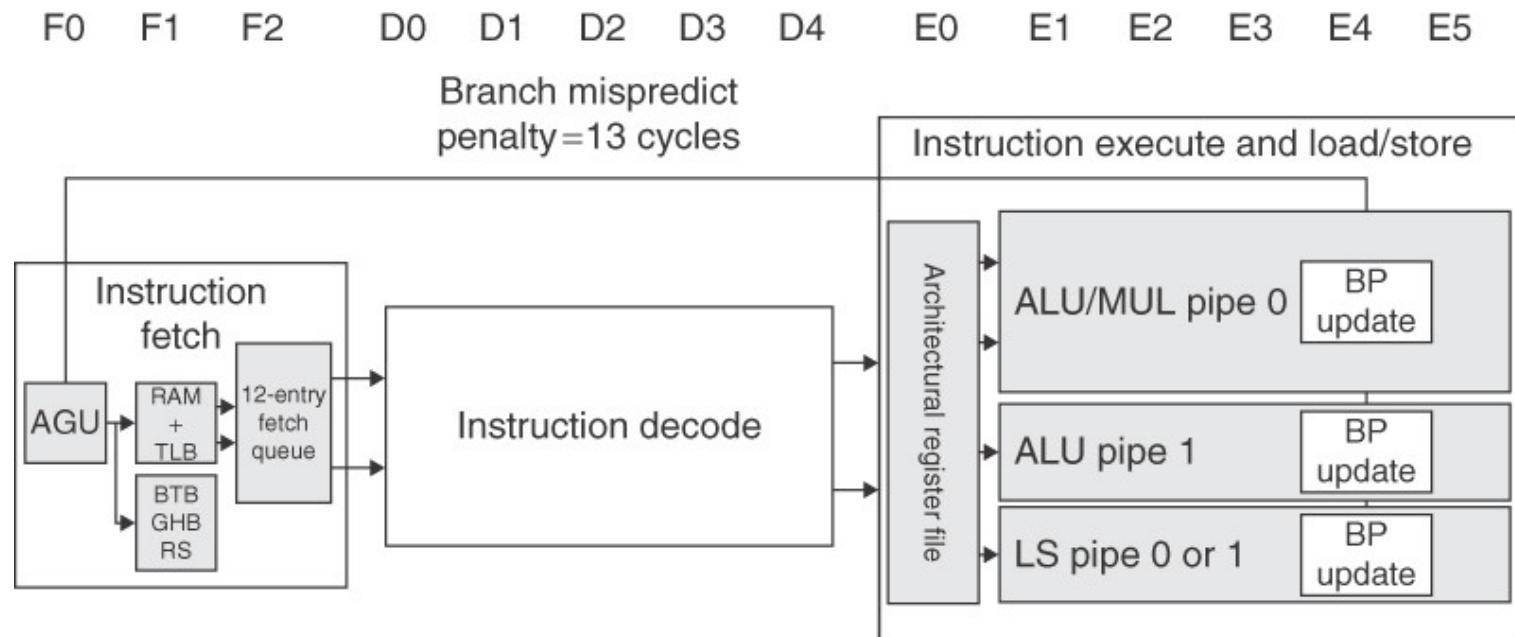
. The reason why the number of instructions issued per clock cycle is variable has to do mainly with two factors

- the multiple pipelines are not exactly alike, which may lead to structural hazards if any combination of instructions is considered
- although forwarding is used in an extensive way across the different pipelines, it is not possible, even with compiler's help, to prevent stalls due to data hazards among successive instructions.

Thus, there are in fact diminishing advantages for a statically scheduled superscalar architecture as the instruction issue width increases. This is why the issue width is normally just two.

ARM Cortex-A8 - 1

The A8 is a dual-issue, statically scheduled superscalar processor with dynamic issue detection, which enables it to issue one or two instructions per clock cycle.



A8 basic 13-stage pipeline structure

Source: Computer Architecture: A Quantitative Approach

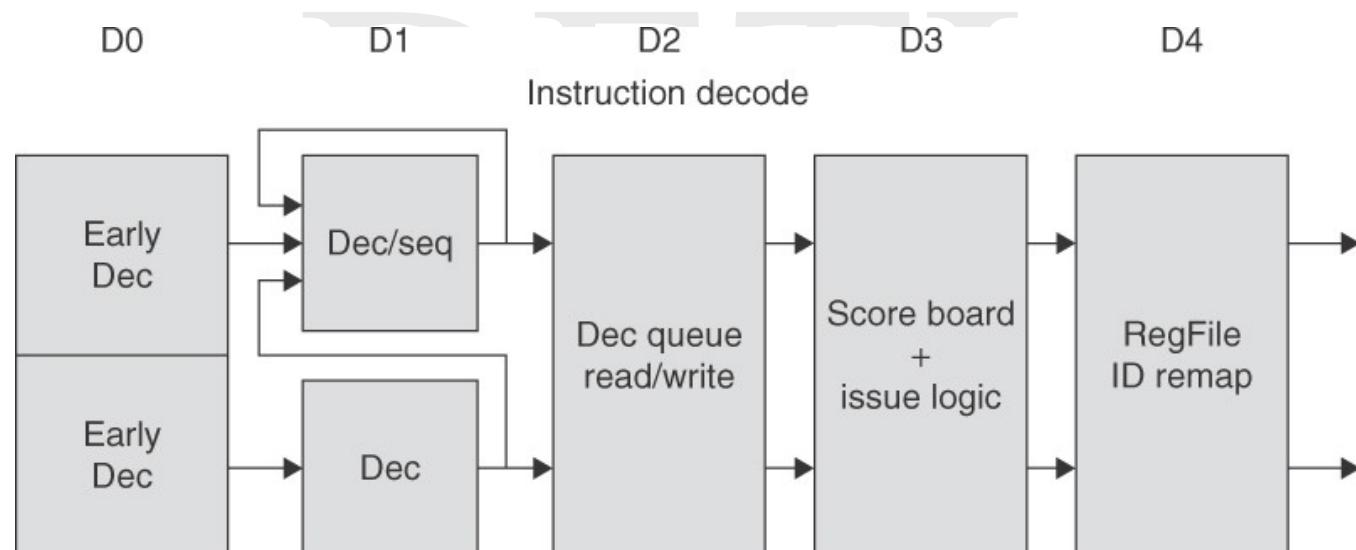
ARM Cortex-A8 - 2

The A8 uses a dynamic branch predictor with a 512-entry 2-way set associative branch target buffer and a 4K-entry global history buffer, which is indexed by the branch history and the current PC. In the event that the branch target buffer misses, a prediction is obtained from the global history buffer, which is then used to compute the branch address.

In addition, an 8-entry return stack is kept to track return addresses. An incorrect prediction results in a 13-cycle penalty as the pipeline is flushed.

ARM Cortex-A8 - 3

Up to two instructions per clock cycle can be issued using an in-order issue mechanism. A simple scoreboard structure is used to track when an instruction can issue. A pair of dependent instructions can be processed through the issue logic, but they will be serialized at the scoreboard, unless the forwarding paths can resolve the dependence.

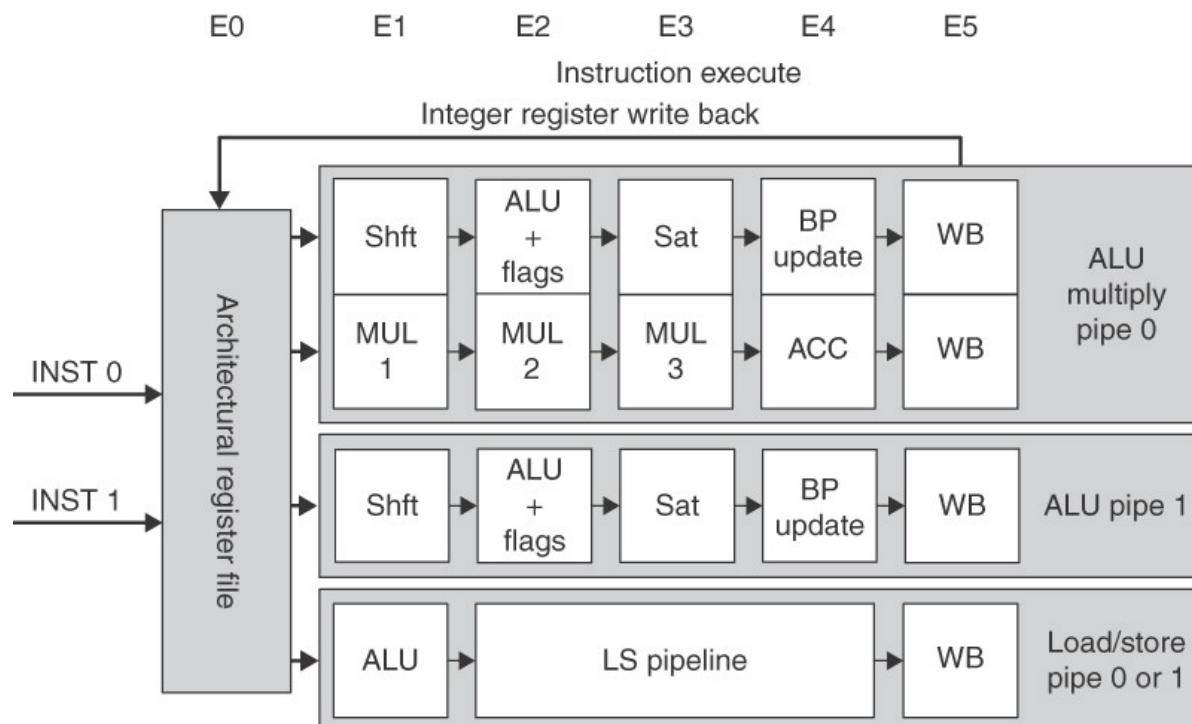


A8 5-stage instruction decode

Source: Computer Architecture: A Quantitative Approach

ARM Cortex-A8 - 4

Either instruction 1 or 2 can go to the load / store pipeline. Fully bypassing is supported between the pipelines in order to minimize stalling at the scoreboard.



A8 execution pipeline

Source: Computer Architecture: A Quantitative Approach

ARM Cortex-A8 - 5

The A8 has an ideal CPI of 0.5 due to its dual-issue structure. Pipeline stalls can arise from three sources

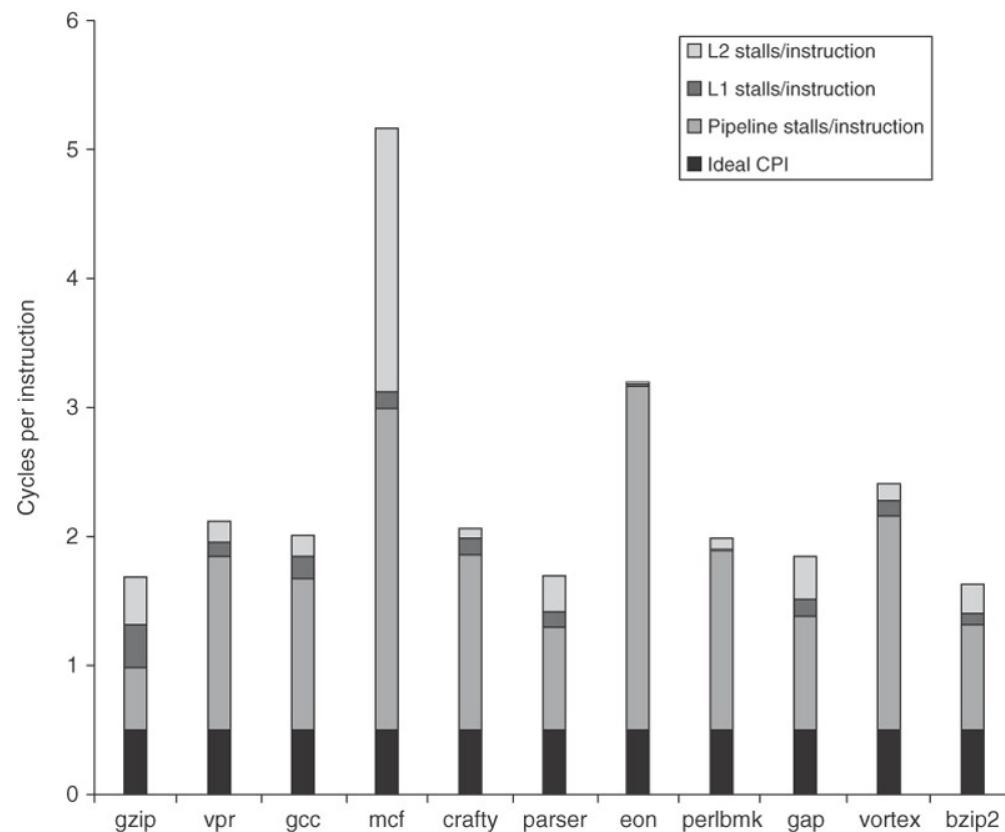
- *structural hazards* – they occur when two adjacent instructions selected for issue simultaneously require the same functional pipeline; since A8 is statically scheduled, it is the compiler duty to try to avoid such conflicts; when they can not be avoided, the A8 can issue at most one instruction in that clock cycle
- *data hazards* – they are detected early in the pipeline, at the scoreboard, and may stall either both instructions (if the first instruction can not issue, the second is always stalled) or just the second of a pair; again it is the compiler duty to try to prevent such stalls whenever possible
- *control hazards* - they only arise when branches are mispredicted.

In addition to stalls due to hazards, L1 and L2 cache misses at the load / store pipeline will also produce stalls.

ARM Cortex-A8 - 6

Estimated composition of CPI of ARM A8 when running the Minnespec benchmark suite

Source: Computer Architecture: A Quantitative Approach



Dynamically scheduled superscalar processor - 1

For simplicity, an issue rate of two instructions per clock cycle will be assumed. The key concepts are not different from those found in modern processors that issue three or more instructions per clock cycle.

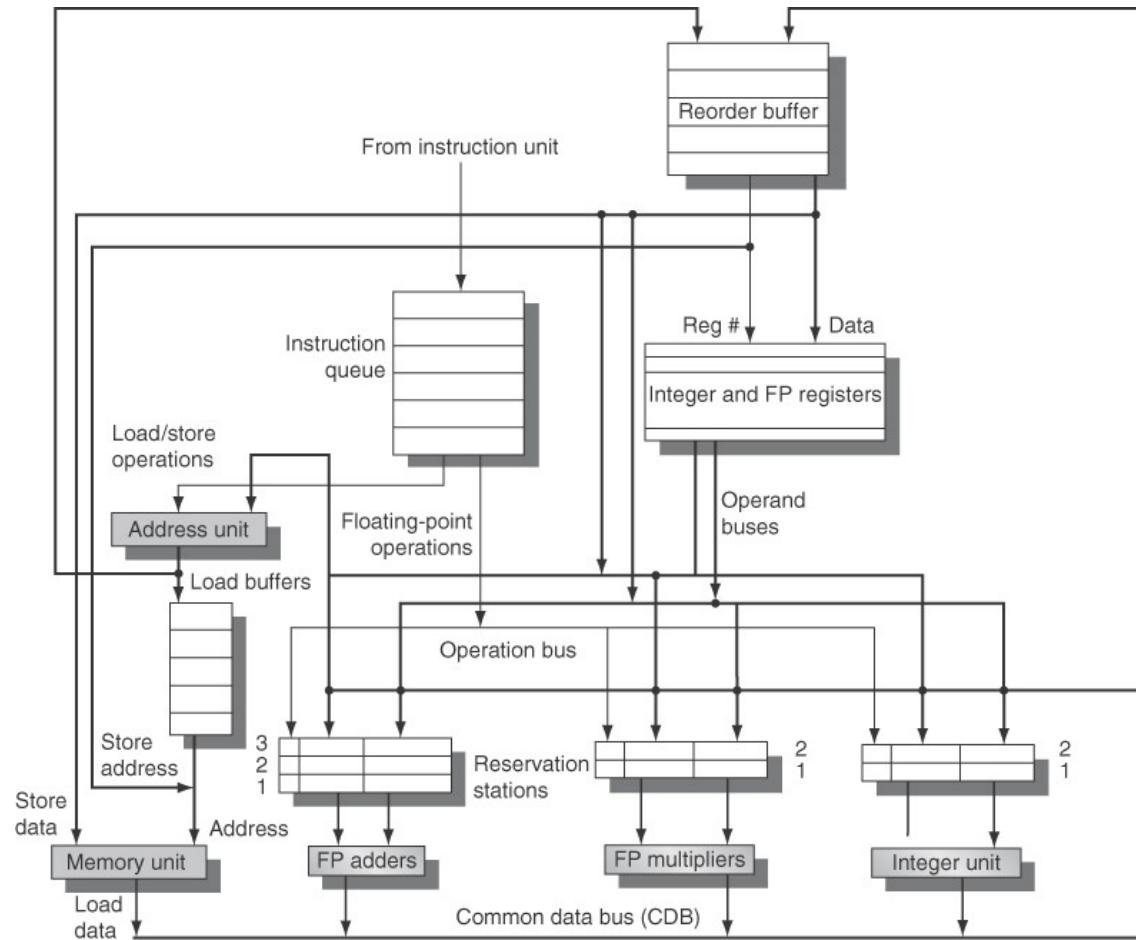
Tomasulo's algorithm will be extended to support a multiple issue speculative superscalar pipeline with separate integer, load / store and floating point functional units, each of which can initiate an operation every clock cycle. To gain full advantage of dynamic scheduling, the pipeline is allowed to issue any combination of two instructions.

Because the interaction of integer and floating point instructions is determinant, Tomasulo's scheme is also extended to deal with both integer and floating point functional units and registers.

Dynamically scheduled superscalar processor - 2

Basic organization of a multiple issue processor with speculation

Source: Computer Architecture: A Quantitative Approach



Dynamically scheduled superscalar processor - 3

Issuing multiple instructions in the same clock cycle in a dynamically scheduled processor, with or without speculation, is a very complex task as the instructions may depend upon one another. Due to this fact, the control tables must be updated in parallel; otherwise, the values will be incorrect or the dependence may be lost.

Two different approaches have been used. The first is to run the step in a fraction of the whole clock cycle for each instruction. For instance, when the issue width is two, this means that it is run in one half of the clock cycle. Unfortunately, it can not be extended in a straightforward manner to handle four instructions! The second is to build the necessary logic to run simultaneously two or more instructions, including any possible dependences among them.

Modern superscalar processors that issue four or more instructions per clock cycle may include both: they pipeline and wide the issue logic.

This issue step is one of the most fundamental bottlenecks in developing dynamically scheduled superscalar processors.

Dynamically scheduled superscalar processor - 4

Required checks and bookkeeping actions for the issue of a bundle of two instructions where **inst1** is a FP load and **inst2** is a FP operation

<i>Instruction status</i>	<i>Wait until</i>	<i>Bookkeeping</i>
Issue load (first instruction of the bundle)	(RS[r1].busy == no) && (ROB[b1].busy == no)	<pre> if(regStat[rs1].busy == yes) { x = regStat[rs1].reorder; if(ROB[x].ready == yes) { RS[r1].Vj = ROB[x].value; RS[r1].Qj = 0; } else RS[r1].Qj = x; } else { RS[r1].Vj = reg[rs1]; RS[r1].Qj = 0; } regStat[rt1].reorder = b1; regStat[rt1].busy = yes; RS[r1].A = imm; RS[r1].dest = b1; RS[r1].busy = yes; ROB[b1].inst = load; ROB[b1].dest = rt1; ROB[b1].ready = no; ROB[b1].busy = yes; </pre>
Issue FP operation (second instruction of the bundle) the first operand comes from load	(RS[r2].busy == no) && (ROB[b2].busy == no)	<pre> RS[r2].Qj = b1; if(regStat[rt2].busy == yes) { x = regStat[rt2].reorder; if(ROB[x].ready == yes) { RS[r2].Vj = ROB[x].value; RS[r2].Qk = 0; } else RS[r2].Qk = x; } else { RS[r2].Vj = reg[rs2]; RS[r2].Qk = 0; } regStat[rd2].reorder = b2; regStat[rd2].busy = yes; RS[r2].dest = b2; RS[r2].busy = yes; ROB[b2].inst = FP op; ROB[b2].dest = rd2; ROB[b2].ready = no; ROB[b2].busy = yes; </pre>

where ROB [b1] and ROB [b2] are ROB entries and RS [r1] and RS [r2] the reservation stations / buffers associated with the two instructions and regStat is the register status.

Dynamically scheduled superscalar processor - 5

In a real situation, every possible combination of dependent instructions allowed to issue in the same clock cycle must be considered. Since the number of possibilities increases as the square of the number of instructions issued in a clock cycle, this turns out to be a pressing concern for a large issue width (beyond four, for instance).

Dynamically scheduled superscalar processor - 6

The basic strategy to update the issue logic in a dynamically scheduled superscalar processor with up to n issues per clock cycle is as follows

1. Assign a reorder buffer and a reservation station / buffer for every instruction that might be issued in the next bundle. This assignment can be done before the instruction types are known by preallocating the reorder buffer entries sequentially to the instructions in the bundle and by ensuring that there are enough reservation stations / buffers available, independent of the bundle content. Should not sufficient reservation stations / buffers be available, the bundle has to be broken and only a subset of these instructions, in the original program order, is issued. The remaining instructions will join the next bundle.
2. Find all dependences among all the instructions in the bundle.

Dynamically scheduled superscalar processor - 7

3. If a dependence of an instruction in the bundle is found to a preceding one in the bundle, the assigned reorder buffer entry number should be used to update the reservation table for the dependent instruction; otherwise, the existing reorder buffer and reservation table information should be used to update the reservation table for the issuing instruction.

At the back end of the pipeline, it is also required to complete and commit multiple instructions per clock cycle. The steps here are, however, somewhat simpler than the issue problem, since the instructions which can actually commit in the same clock cycle must have already dealt with and resolved the dependences.

Dynamically scheduled superscalar processor - 8

Consider the code sequence bellow and analyze its execution on a 2-issue processor without and with speculation (the effects of delayed branches are ignored)

Loop:	L.D	R2, 0 (R1)
	DADDIU	R2, R2, 1
	S.D	R2, 0 (R1)
	DADDIU	R1, R1, 8
	BNE	R1, R3, Loop

It is assumed that the register R1 contains initially the address of the first array element and that the contents of the register R3-8 is the address of the last array element.

It is also assumed that there are separate functional units for effective address calculation, for ALU operations and for branch condition evaluation. The first three iterations should be analysed.

Dynamically scheduled superscalar processor - 9

Code execution on a 2-issue processor without speculation

<i>iteration number</i>	<i>instruction</i>	<i>issues at clock cycle number</i>	<i>executes at clock cycle number</i>	<i>memory access at clock cycle number</i>	<i>write CDB at clock cycle number</i>	<i>comment</i>
1	L.D R2, 0 (R1)	1	2	3	4	first issue
1	DADDIU R2, R2, 1	1	5		6	wait for L.D
1	S.D R2, 0 (R1)	2	3	7		wait for DADDIU
1	DADDIU R1, R1, 8	2	3		4	execute
1	BNE R1, R3, Loop	3	7			wait for DADDIU
2	L.D R2, 0 (R1)	4	8	9	10	wait for BNE
2	DADDIU R2, R2, 1	4	11		12	wait for L.D
2	S.D R2, 0 (R1)	5	9	13		wait for DADDIU
2	DADDIU R1, R1, 8	5	8		9	execute
2	BNE R1, R3, Loop	6	13			wait for DADDIU
3	L.D R2, 0 (R1)	7	14	15	16	wait for BNE
3	DADDIU R2, R2, 1	7	17		18	wait for L.D
3	S.D R2, 0 (R1)	8	15	19		wait for DADDIU
3	DADDIU R1, R1, 8	8	14		15	execute
3	BNE R1, R3, Loop	9	19			wait for DADDIU

Dynamically scheduled superscalar processor - 10

Code execution on a 2-issue processor with speculation

<i>iteration number</i>	<i>instruction</i>	<i>issues at clock cycle number</i>	<i>executes at clock cycle number</i>	<i>memory access at clock cycle number</i>	<i>write CDB at clock cycle number</i>	<i>commit at clock cycle number</i>	<i>comment</i>
1	L.D R2,0(R1)	1	2	3	4	5	first issue
1	DADDIU R2,R2,1	1	5		6	7	wait for L.D
1	S.D R2,0(R1)	2	3			7	wait for DADDIU
1	DADDIU R1,R1,8	2	3		4	8	commit in order
1	BNE R1,R3,Loop	3	7			8	wait for DADDIU
2	L.D R2,0(R1)	4	5	6	7	9	no execute delay
2	DADDIU R2,R2,1	4	8		9	10	wait for L.D
2	S.D R2,0(R1)	5	6			10	wait for DADDIU
2	DADDIU R1,R1,8	5	6		7	11	commit in order
2	BNE R1,R3,Loop	6	10			11	wait for DADDIU
3	L.D R2,0(R1)	7	8	9	10	12	earliest possible
3	DADDIU R2,R2,1	7	11		12	13	wait for L.D
3	S.D R2,0(R1)	8	9			13	wait for DADDIU
3	DADDIU R1,R1,8	8	9		10	14	executes earlier
3	BNE R1,R3,Loop	9	13			14	wait for DADDIU

Dynamically scheduled superscalar processor - 11

The example shows how speculation can be advantageous when there are data dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. It is important to note that incorrect speculation harms performance and dramatically lowers energy efficiency!

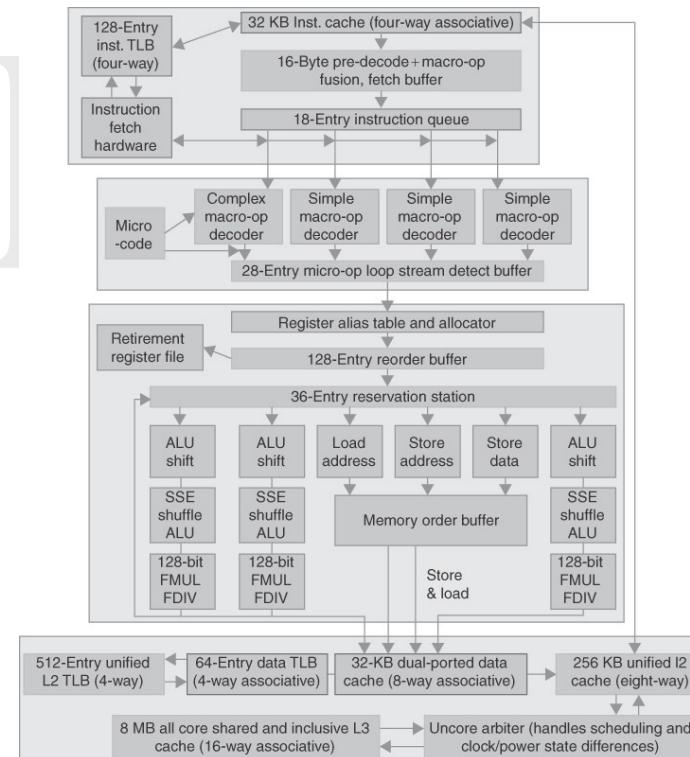
Why is it so?

Intel Core i7 - 1

The Intel Core i7 uses an aggressive out-of-order speculative microarchitecture with reasonably deep pipelines having as goal the attaining of high instruction throughput by combining multiple issue and high clock rates.

Intel Core i7 pipeline structure with memory system interface

Source: Computer Architecture: A Quantitative Approach



Intel Core i7 - 2

Some features of Intel Core i7 pipeline are next presented

1. The processor uses a multilevel branch target buffer, located at the instruction fetch stage, to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 clock cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. These 16 bytes are placed in the pre-decode instruction buffer, where a procedure called macro-op fusion is executed. *Macro-op fusion* takes instruction combinations such a compare followed by a branch and generates a single operation. The pre-decode stage also breaks the 16 bytes into individual x86 instructions. Individual x86 instructions, including some fused instructions, are placed in the 18-entry instruction queue.

Intel Core i7 - 3

3. Individual x86 instructions are translated into micro-ops which are simple MIPS-like instructions executed directly by the pipeline. This approach was introduced in the Pentium Pro and has been used ever since. Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that produces the corresponding micro-ops sequence: it can generate up to four micro-ops per clock cycle and goes on until the whole sequence is produced. The micro-ops are placed in the 28-entry micro-op buffer according to the order of the x86 instructions.
4. The micro-op buffer performs *loop stream detection* and *microfusion*. If there is a small sequence of instructions, less than 28 or 256 bytes in length that comprises a loop, the loop stream detector will find the loop and directly issue micro-ops from the buffer, eliminating the need for the instruction fetch and the instruction decode stages to be activated. On the other hand, microfusion combines instructions pairs such as load / ALU operation and ALU operation / store and issues them to a single reservation station, where they can still issue independently.

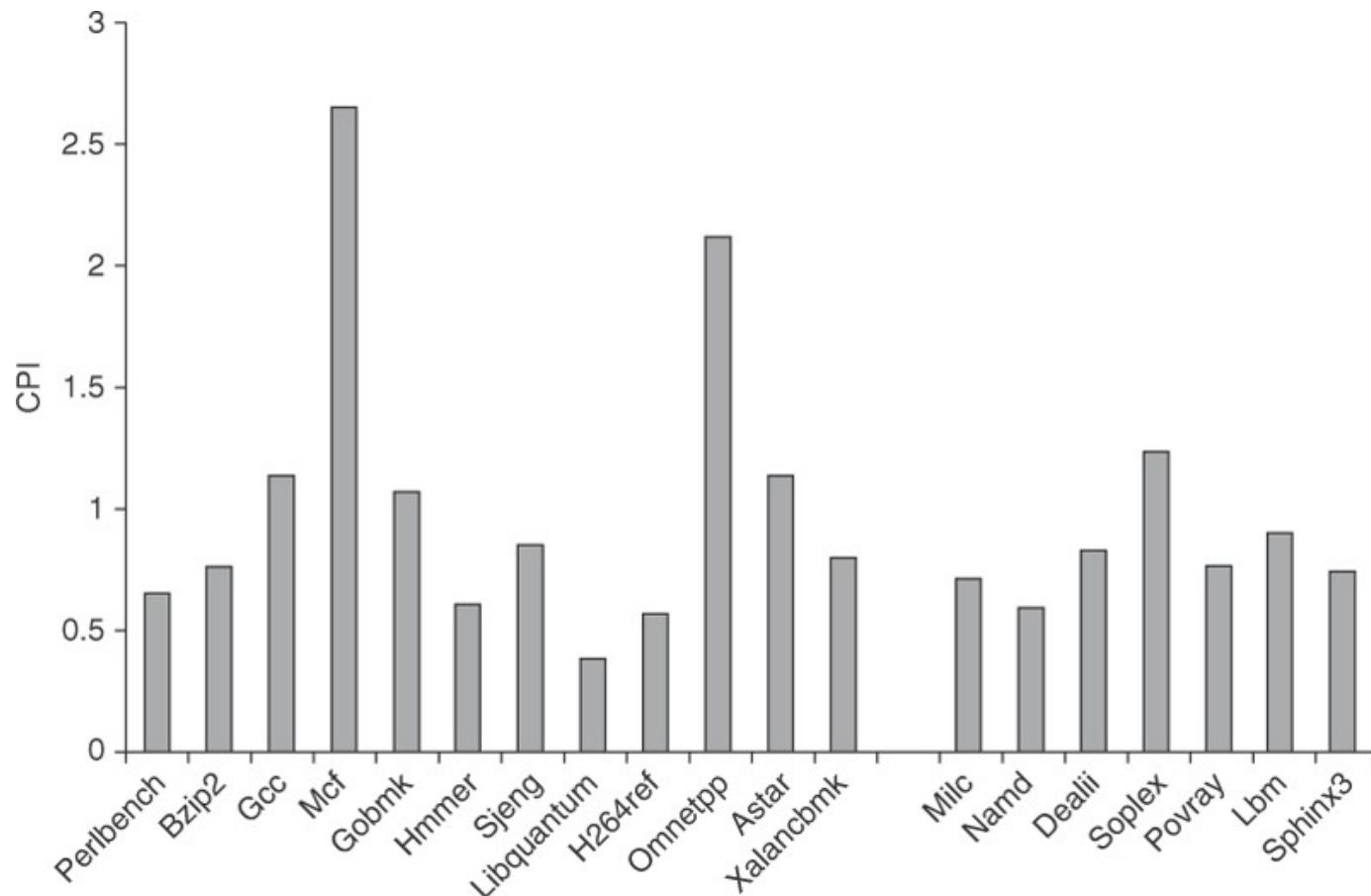
Intel Core i7 - 4

3. The basic instruction issue comprises looking up the register location in the register tables, renaming the registers, allocating a reorder entry and fetching any results from the registers or the reorder buffer, before sending the micro-ops to the reservation stations.
6. The i7 uses a 36-entry centralized reservation station shared by six function units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. Micro-ops are executed by the individual functional units and the results are sent back to any waiting reservation station as well as to the register retirement unit, where the register state is updated once it is asserted that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instruction at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed and the instructions are removed from the reorder buffer.

Intel Core i7 - 5

Performance of Intel Core i7 CPI for the SPECCPU2006 benchmark suite

Source: Computer Architecture: A Quantitative Approach



Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
 - Chapter 3: *Instruction-Level Parallelism and its Exploitation* (Sections 1 to 12)
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
 - Chapter 16: *Instruction-Level Parallelism and Superscalar Processors*

universidade de aveiro



Arquitecturas de Alto Desempenho

Data-Level Parallelism

António Rui Borges

Summary

- *Types of data-level parallelism*
- *Vector architectures*
- *Vector operations*
- *Improving performance*
 - *Multiple lanes: behind one element per clock cycle*
 - *Vector length registers: handling loops not equal to the register size*
 - *Vector mask registers: handling conditional statements in vector loops*
 - *Memory banks: supporting bandwidth for vector load / store units*
- *Widening the application area*
 - *Stride: handling multidimensional arrays*
 - *Gather-scatter: handling sparse matrices*
 - *Programming vector architectures*
- *SIMD instruction set extensions for multimedia*
 - *Programming multimedia SIMD architectures*
- *Suggested reading*

Types of data-level parallelism - 1

Data-level parallelism (DLP) arises when multiple data items are processed at the same time. Two different types of computer architectures can fulfill this aim: SIMD and MIMD. Since a single instruction can launch many data operations, SIMD is potentially more energy efficient than MIMD, where one instruction is fetched and executed per data operation. A further advantage of SIMD over MIMD is that the programmer may still keep thinking sequentially and, yet, get a parallel speed up by carrying out independent simultaneous data operations.

Types of data-level parallelism - 2

Three variations of SIMD will be discussed

- *vector architectures* – they represent essentially a pipelined execution of many data operations; they were traditionally targeted to high-end scientific applications where data are well-structured and the number of computations is very large; supercomputers of the past were built in this way
- *multimedia instruction set extensions* – they represent essentially a parallel execution of data operations and are found today in most instruction set architectures that support multimedia applications
- *graphic processing units (GPUs)* – they share many characteristics with vector architectures, but there is a key difference: typically, they act as coprocessors in computer systems which include a conventional processor and its associated memory in addition to the GPU and the graphic memory, giving rise to what is now called *heterogeneous computing*.

Vector architectures - 1

Vector architectures grab sets of data elements scattered about memory, transfer them into large, sequential register banks, operate on those register banks through matrix-oriented computations and disperse the results back into memory. Thus, a single instruction operates on data vectors, giving rise to dozens of register-register operations on independent data elements.

The large register banks act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth. Due to the fact that loads and stores are deep pipelined, the program minimizes memory latency to once per vector load and store, instead of once per element. Indeed, vectors programs strive to keep memory access continuous so that it can take place in parallel to the computations being carried out by the processor.

Vector architectures - 2

To enhance the discussion about vector processing, a processor loosely based on Cray-1 is described. The processor's instruction set architecture will be called VMIPS because its scalar portion is MIPS and its vector portion is the logical vector extension of MIPS.

The primary components of VMIPS instruction set architecture are

- *vector registers* – each vector register is in itself a fixed-length bank holding a single vector; VMIPS has eight vector registers, each holding 64 64-bit wide elements; the vector register bank provides enough input / output ports to feed all the vector functional units with a high degree of overlap; there are 16 read ports and 8 write ports which are connected to the functional units by a crossbar switch
- *vector functional units* – each unit is fully-pipelined and may start a new operation every clock cycle; one needs a control unit to detect hazards, both structural hazards for functional unit allocation and data hazards on register access

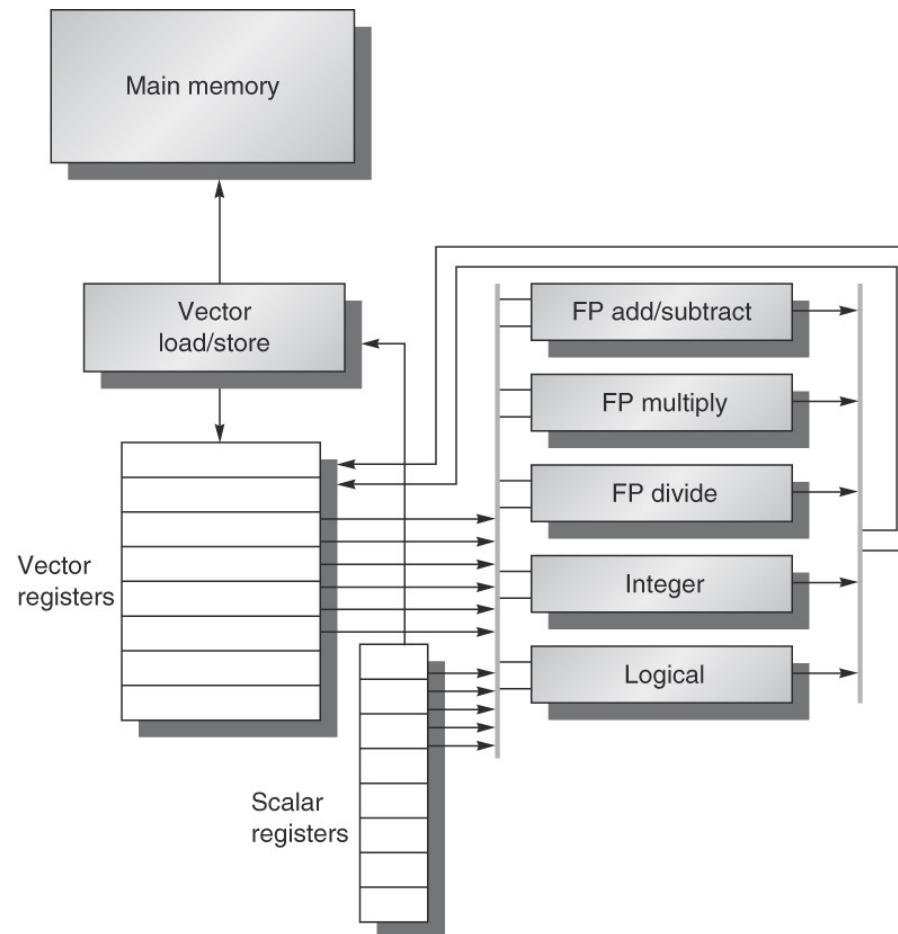
Vector architectures - 3

- *vector load / store unit* – the vector load / store unit loads from or stores to memory a data vector; VMIPS unit is fully-pipelined so that words are moved between the vector registers and memory with a bandwidth of one word per clock cycle, after the initial latency; this unit will also handle scalar loads and stores
- *scalar registers* – scalar registers can also provide input to the vector functional units; they are intended to hold the addresses to pass to the vector load / store unit as well; VMIPS has the usual 32 general purpose registers and the 32 floating point registers of MIPS, all 64-bit wide.

Vector architectures - 4

Basic organization of VMIPS architecture

Source: Computer Architecture: A Quantitative Approach



Vector architectures - 5

VMIPS vector instructions (only double precision floating point operations are shown)

<i>instruction</i>	<i>operands</i>	<i>comment</i>
ADDVV.D	V1, V2, V3	$V1[k] = V2[k] + V3[k]$ (vector-vector addition)
ADDVS.D	V1, V2, F3	$V1[k] = V2[k] + F3$ (vector-scalar addition)
SUBVV.D	V1, V2, V3	$V1[k] = V2[k] - V3[k]$ (vector-vector subtraction)
SUBSV.D	V1, F2, V3	$V1[k] = F2 - V3[k]$ (scalar-vector subtraction)
SUBVS.D	V1, V2, F3	$V1[k] = V2[k] - F3$ (vector-scalar subtraction)
MULVV.D	V1, V2, V3	$V1[k] = V2[k] \times V3[k]$ (vector-vector multiplication)
MULVS.D	V1, V2, F3	$V1[k] = V2[k] \times F3$ (vector-scalar multiplication)
DIVVV.D	V1, V2, V3	$V1[k] = V2[k] / V3[k]$ (vector-vector division)
DIVSV.D	V1, F2, V3	$V1[k] = F2 / V3[k]$ (scalar-vector division)
DIVVS.D	V1, V2, F3	$V1[k] = V2[k] / F3$ (vector-scalar division)
LV	V1, R1	$V1[k] = \text{mem}[R1 + k]$ (vector elements are adjacent)
SV	R1, V1	$\text{mem}[R1 + k] = V1[k]$ (vector elements are adjacent)
LVWS	V1, (R1, R2)	$V1[k] = \text{mem}[R1 + k \times R2]$ (vector elements are separated by the fixed value in R2)
SVWS	(R1, R2), V1	$\text{mem}[R1 + k \times R2] = V1[k]$ (vector elements are separated by the fixed value in R2)
LVI	V1, (R1+V2)	$V1[k] = \text{mem}[R1 + V2[k]]$ (vector elements are random separated by values in V2)
SVI	(R1+V2), V1	$\text{mem}[R1 + V2[k]] = V1[k]$ (vector elements are random separated by values in V2)
CVI	V1, R1	create the index vector $V1[k] = k \times R1$
S__VV.D	V1, V2	compare the elements $V1[k], V2[k]$ ($=$ EQ, NE, GT, LT, GE, LE) $VM[k] = 1$ (condition true) - 0 (otherwise)
S__VF.D	V1, F2	compare the elements $V1[k], F2$ ($=$ EQ, NE, GT, LT, GE, LE) $VM[k] = 1$ (condition true) - 0 (otherwise)
POP	R1, VMR	$R1 = \text{number of } 1\text{s in vector mask register VM}$
CVM		set <i>vector mask register</i> contents to all 1s
MTC1	VLR, R1	move contents of R1 to <i>vector length register VL</i>
MFC1	R1, VLR	move contents of <i>vector length register VL</i> to R1
MVTM	VMR, F1	move contents of F1 to <i>vector mask register VM</i>
MVFM	F1, VMR	move contents of <i>vector mask register VM</i> to F1

Vector architectures - 6

The power wall leads to value architectures that can deliver high performance without the energy consumption and the design complexity of out-of-order superscalar processors. Vector instructions are a natural match to this trend, since they can be used to increase the performance of simple in-order scalar processors without greatly increasing energy demands and design complexity. In practice, developers can express many of the programs that ran well in complex out-of-order designs more efficiently as data-level parallelism in the form of vector instructions.

With a vector expression, the system can perform the operations on the vector data elements in many different ways, including operating simultaneously on these elements. This flexibility lets vector designs use slow, but wide, execution units to achieve high performance at low power. Furthermore, the independence of elements within a vector instruction set allows scaling the functional units without carrying out additional dependency checks as superscalar require.

Vectors naturally accomodate varying data types: one view of a vector register size is 64 64-bit wide elements, but 128 32-bit wide elements, 256 16-bit wide elements, or even 512 8-bit wide elements, are equally valid views. Such a multiplicity makes a vector architecture useful for multimedia as well as scientific applications.

Vector operations - 1

Consider the following vector problem

$$Y = a \times X + Y$$

where X and Y are vectors, initially resident in memory, and a is a scalar.

This problem is called SAXPY or DAXPY, according to the operands are single or double precision, and forms the inner loop of the *Linpack benchmark*. *Linpack* is a library of linear algebra subroutines and the *Linpack benchmark* consists of a program for performing Gaussian elimination.

It is assumed for now that the vectors length is 64 (the length of VMIPS vector registers) and one wants to compare the code for the DAXPY loop written both in MIPS and VMIPS.

The initial addresses of the memory locations where vectors X and Y are stored, are assumed to be in registers R_x and R_y , respectively.

Vector operations - 2

MIPS code for DAXPY

```
L.D      F0,a  
DADDIU   R4,Rx,512  
Loop:  
        L.D      F2,0(Rx)  
        MUL.D    F2,F2,F0  
        L.D      F4,0(Ry)  
        ADD.D    F4,F4,F2  
        S.D      F4,0(Ry)  
        DADDIU   Rx,Rx,8  
        DADDIU   Ry,Ry,8  
        DSUBU    R20,R4,Rx  
        BNEZ    R20,Loop
```

VMIPS code for DAXPY

```
L.D      F0,a  
LV       V1,Rx  
MULVS.D V2,V1,F0  
LV       V3,Ry  
ADDVV.D V4,V2,V3  
SV       Ry,V4
```

Vector operations - 3

The most striking difference between the two programs is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus 578 for MIPS. This reduction is due to the fact that the vector operations work directly on the 64 elements, as a whole, and that the overhead instructions, which represent nearly half the loop on MIPS, are not present here.

When the compiler generates vector instructions for a sequence and the resulting code spends much of its time running in vector mode, the code is said to be *vectorized* or *vectorizable*. Loops can be vectorized only if they do not have dependences between loop iterations, or *loop-carried dependences*, as they are also called.

Another important difference is the frequency of pipeline interlocks. In the MIPS code, they occur for every loop iteration, while for the vector processor, each vector instruction will only stall for the first element in each vector, subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *instruction*, rather than once per vector *element*. Vector architects call forwarding of element-dependent operations *chaining*, because the dependent operations are chained together.

Vector operations - 4

The execution time of a sequence of vector operations depends primarily on three factors

- the length of the operand vectors
- the number of structural hazards, and their kind, that the operations underlie
- the data dependences among successive operations.

Given the vector length and the initiation rate, which is the rate a specific vector unit consumes new operands and produces new results, the time for a single vector instruction can be computed. For simplicity, it will be assumed that VMIPS implementation has all functional units having a single pipeline, or *lane*, with an initiation rate of one element per clock cycle for individual operations. Thus, the execution time for a single vector operation is approximately the vector length.

Vector operations - 5

The notion of convoy will be also introduced to simplify the discussion of vector execution and vector performance. A *convoy* is to be understood as the set of vector instructions that can potentially be executed together. The instructions in a convoy *must not* contain any structural hazards. If such hazards were present, they would need to be serialized and initiated in different convoys. Again, to make analysis simple, it will be assumed that a convoy of instructions must complete execution before any other instructions, scalar or vector, may begin execution.

It might appear that besides vector instruction sequences with structural hazards, sequences with RAW hazards should also be in separate convoys, but one has to remember that chaining allows them to be executed together. Through its application, a vector operation may start as soon as the individual elements of their vector source operands become available: the results from the first functional unit in the chain are forwarded to the other functional units. In practice, this is implemented by allowing the processor to read and write a particular vector register at the same time, provided that different elements are accessed. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, provided that a structural hazard is not generated.

Vector operations - 6

Consider the following VMIPS code sequence

```
LV      V1, Rx  
MULVS.D V2, V1, F0  
LV      V3, Ry  
ADDVV.D V4, V2, V3  
SV      Ry, V4
```

A minimum of three convoys are needed to execute it.

The first convoy starts with the first `LV` instruction. The `MULVS.D` is dependent on it, but chaining allows it to be included in the same convoy. The second convoy starts with the second `LV` instruction, a structural hazard would be induced if placed in the first convoy (why?). The `ADDVV.D` is dependent on it, but it can again be placed in the same convoy via chaining. Finally, the `SV` instruction induces another structural hazard, if placed in the second convoy, so it must go in a third convoy.

Vector operations - 7

To turn convoys into execution time, a timing metric is needed to estimate the execution time of a convoy. This metric is called a *chime*. Therefore, a vector code sequence consisting of m convoys executes in m chimes: for vectors of size n , this would mean an execution time of approximately $m \cdot n$ clock cycles for VMIPS.

The chimes approximation ignore some processor specific overheads, many of which are independent on the vector length. Hence, measuring time in chimes is a better approximation for longer vectors than for short ones.

One ignored source of overhead in measuring chimes is the possible limitation on initiating multiple vector instructions in the same clock cycle. If only one vector instruction can be initiated per clock cycle, as it happens in most vector processors, the chime count will underestimate the actual execution time of a convoy.

The most important source of overhead ignored by the chime model is *vector start up time*. The start up time is chiefly determined by the pipelining latency of the vector functional units. The same pipeline depths, as the ones present in Cray-1, will be used for VMIPS although latencies in modern processors tend to increase, specially for vector load operations. The pipeline depths are 6 clock cycles for FP add, 7 for FP multiply, 20 for FP divide and 12 for vector load.

Improving performance

The following questions must have adequate answers if one aims to improve the performance of a vector processor

- how can a vector processor execute a single vector more than one element per clock cycle?
- how does a vector processor handle programs where the vector lengths are not the same as the length of the vector registers?
- what happens when there are conditional statements inside the code to be vectorized?
- what does a vector processor need from the memory system?

Multiple lanes: beyond one element per clock cycle - 1

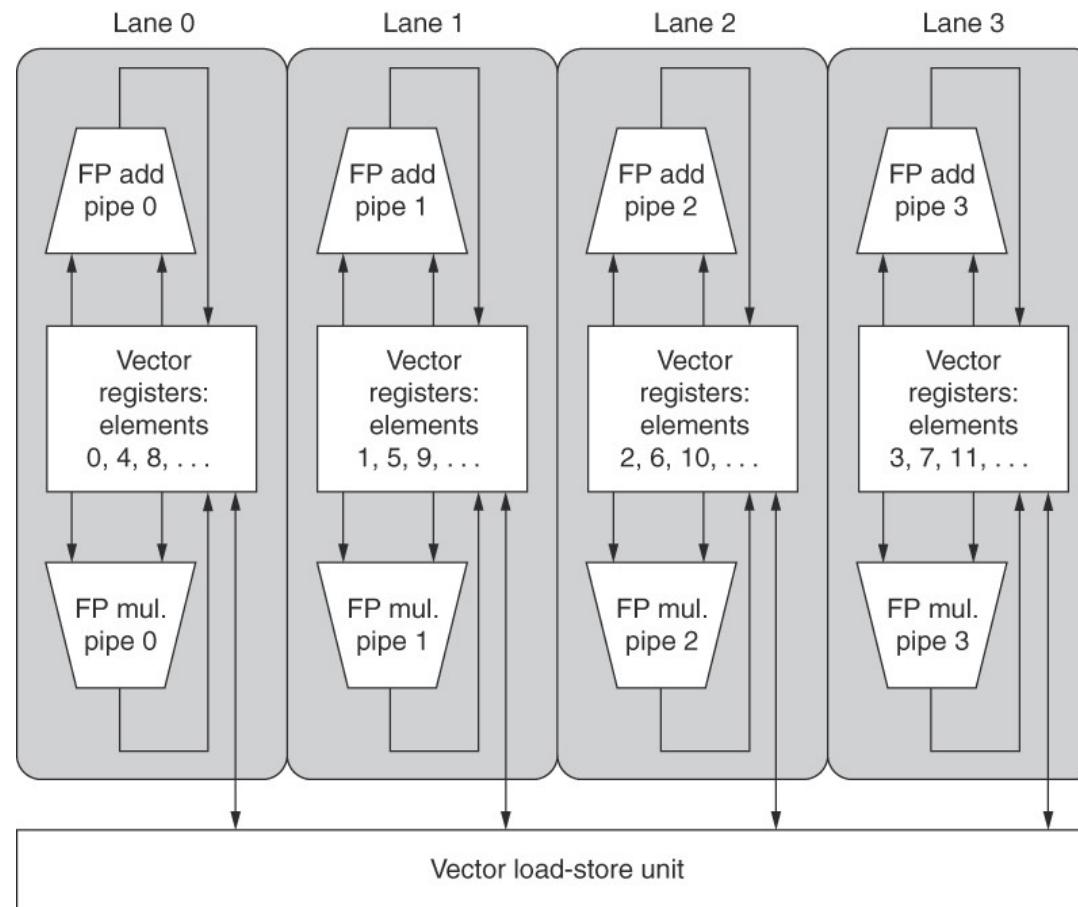
A critical advantage of a vector instruction set is that it allows the program to pass a large amount of parallel work to the hardware using only a single short instruction. This instruction includes scores of independent operations, yet encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allows an implementation to execute these elemental operations through either a deeply pipelined functional unit, as VMIPS, or an array of parallel functional units, or a combination of both.

The VMIPS instruction set has the property that all vector arithmetic instructions only allow element k of one vector register to take part in operations with element k of another vector register. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as having multiple parallel lanes.

Multiple lanes: beyond one element per clock cycle - 2

Structure of a vector unit organized in four lanes

Source: Computer Architecture: A Quantitative Approach



Multiple lanes: beyond one element per clock cycle - 3

Going from one to four lanes reduces the number of clock cycles for a chime from 64 to 16. However, for a multilane implementation to be effective, both the applications and the architecture must support long vectors; otherwise the execution will be so fast that there is a risk of running out of instruction bandwidth and, thus, requiring ILP techniques to feed enough vector instructions.

Each lane contains a portion of the vector register bank and one execution pipeline from each vector functional unit. The vector functional units, thus, execute vector instructions at the rate of one *group of elements* per clock cycle. Avoiding interlane communication reduces both the wiring cost and the number of register bank ports needed to build a highly parallel execution unit.

Adding multiple lanes is a popular technique to improve vector performance, as it requires little increase in control complexity and does not imply changes to existing machine code. It also allows designers to trade off die area, clock rate and energy without sacrificing peak performance (if, for instance, the clock rate is halved, doubling the number of lanes will retain the same potential performance).

Vector length registers: handling loops not equal to the register size - 1

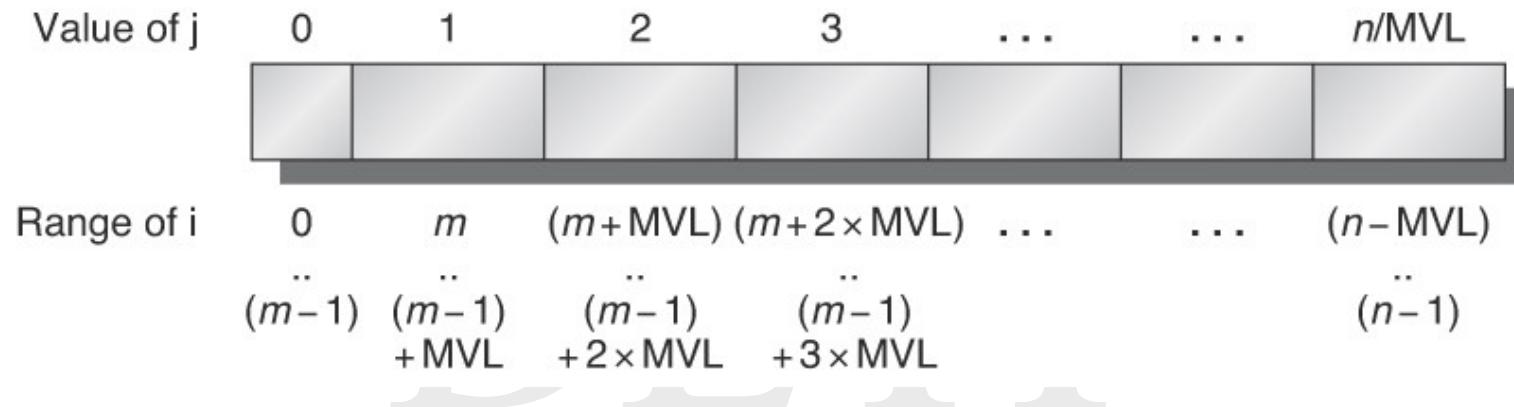
A vector processor has a natural vector length determined by the number of elements in each vector register. This length is unlikely to match the real vector length in a program. Moreover, the length of a particular vector operation in a real program is often unknown at compile time. In fact, a single piece of code may even require different vector lengths, as it happens if it is included in a procedure which has as parameter precisely the number of loop iterations.

The solution to these problems is to create a *vector length register* (VL). The VL controls the length of any vector operation, including loads and stores. The value stored in VL , however, can not be greater than the length of the vector registers, so solutions based on VL only work when the real vector length is less or equal to the *maximum vector length* (MVL), an architecture-based parameter that specifies the number of data elements in the vector registers for the current implementation.

Vector length registers: handling loops not equal to the register size - 2

Application of strip mining to a vector of arbitrary length

Source: Computer Architecture: A Quantitative Approach



When the real vector length is greater than the *maximum vector length* (MVL), a technique known as *strip mining* is applied.

```
b = 0;  
VL = n % MVL;  
for (j = 0; j <= n/MVL; j++)  
{ for (i = b; i < b+VL; i++) // vector operation  
    Y[i] = a * X[i] + Y[i];  
    b += VL;  
    VL = MVL;  
}
```

Vector mask registers: handling conditional statements in vector loops - 1

According to Amdahl's law, the speed up of programs with low to moderate levels of parallelization, in this case vectorization, is very limited. The presence of conditionals, like **if** statements, inside loops is a critical reason for low levels of vectorization because of the introduction of control dependences.

Consider the code

```
for (i = 0; i < VL; i++)
    if (X[i] != 0.0) X[i] = X[i] - Y[i];
```

The loop can not be vectorized in a straightforward manner due to the conditional execution of the body. However, if it can be worked out that the loop is run only for the iterations where $X[i] \neq 0$, then the subtraction operation could be vectorized.

Vector mask registers: handling conditional statements in vector loops - 2

The common extension for this capability is called *common mask control*. Mask registers provide conditional execution of each elemental operation in a vector operation. The *vector mask control* uses a boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a boolean condition to assert whether a scalar instruction should be executed.

When the *vector mask register* (VM) is enabled, any vector instructions operate only on the vector elements whose corresponding entries in VM are one. The vector elements whose corresponding entries in VM are zero, remain unaffected. Clearing VM, that is, setting all its bits to ones, makes subsequent vector instructions to operate on all vector elements.

Vector mask registers: handling conditional statements in vector loops - 3

Assuming that the start addresses of the memory locations where vectors X and Y are stored, are in registers Rx and Ry, the previous loop may be coded as

LV	V1, Rx
LV	V2, Ry
L.D	F0, 0
SNEVS.D	V1, F0
SUBVV.D	V1, V1, V2
SV	Rx, V1

// sets VM[i] to 1, if V1[i] ≠ 0

Compiler writers call *if conversion* the transformation that changes an **if** statement into a straight line code sequence using conditional execution.

Conditional execution does have some overhead, exposed by the need to execute the **S_VS** .instruction. However, even with a significative number of zeros in VM, using *vector mask control* may still be faster than using the scalar mode.

Memory banks: supporting bandwidth for vector load / store units - 1

The behavior of a load / store vector unit is a lot more complicated than that of an arithmetic functional unit. The *start up time* for a load or store operation is the time required to get the first word from memory into a register or from a register into memory. If the remainder of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be one clock cycle because memory bank stalls can reduce the effective throughput.

Typically, penalties for start ups on load / store units are much higher than those for the arithmetic units, over 100 clock cycles on many processors. For VMIPS, a start up time of 12 clock cycles is assumed (the same as for Cray-1). Caching is being used by the most recent vector processors to bring down the latency time of vector loads and stores.

Memory banks: supporting bandwidth for vector load / store units - 2

To maintain an initiation rate of one word fetched or stored per clock cycle, the memory system must be capable of producing or consuming this amount of data. Spreading accesses across multiple independent memory banks usually delivers the desired rate.

This spreading, rather than simple memory interleaving, is relevant because

- most vector computer systems contain multiple vector processors that share the same memory system, meaning that each processor will be generating its own address stream
- the ability to load and store data words whose addresses are not sequential, a feature supported by most vector processors, requires independent bank addressing rather than interleaving
- simultaneous access by multiple load / store units to the memory system requires multiple banks and the capability of independent address control to them.

Memory banks: supporting bandwidth for vector load / store units - 3

The above mentioned reasons, taken together, lead to a large number of independent memory banks.

The largest configuration of Cray T90 (Cray T932) has 32 vector processors, each capable of generating 4 loads and 2 stores per clock cycle. The processors clock cycle is 2.167 ns, while the SRAMs cycle time used in the memory system is 15 ns.

Calculate the minimum number of memory banks required to allow all processors to run at full memory bandwidth.

The maximum number of memory references per clock cycle is equal to 192 (32 processors times 6 references per processor). On the other hand, each SRAM bank is busy for $15 / 2.167 \approx 7$ clock cycles. Hence, a minimum of $192 \times 7 = 1344$ memory banks are needed!

The Cray T932 actually has 1024 memory banks, not sustaining full bandwidth for all processors in simultaneous. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time and thereby providing sufficient bandwidth.

Widening the application area

The following questions must have adequate answers if one aims to widen the type of problems that can be solved efficiently by a vector processor

- how does a vector processor handle multidimensional matrices?
- how does a vector processor handle sparse matrices?
- how does one program a vector computer?

Stride: handling multidimensional arrays - 1

Successive elements of a vector may not be stored in adjacent memory locations.

Consider the code below for matrix multiplication of matrices of size $N \times N$

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    { A[i][j] = 0.0;
        for (k = 0; k < N; k++)
            A[i][j] += B[i][k] * D[k][j];
    }
```

To make it run efficiently on a vector processor, the multiplication of each row of B by each column of D can be vectorized and the inner loop strip-mined with k as the index variable.

To do so, a way must be found on how to address successive elements of D . One should note that when an array is stored in memory within a C environment, it is linearized and laid out in row-major order. This means that successive elements of D , accessed by the iterations in the inner loop, are separated by N . Thus, without reordering the loops, the compiler can not hide the distance between successive elements of D .

Stride: handling multidimensional arrays - 2

The distance separating elements to be gathered in a single register is called *stride*. Once a vector is loaded into a vector register, it acts as if its successive elements are adjacent, that is, a vector processor can handle strides greater than one provided that there are load and store instructions with stride capability. This ability of accessing non sequential memory locations and of reshaping their content into a dense structure is one of the major advantages of a vector processor.

The vector stride, as the vector starting address, can be loaded in a general purpose register and used in special load and store instructions. In VMIPS, the instruction LVWS (load vector with stride) fetches the vector elements, separated in memory by a stride greater than one, into a vector register. Likewise, the instruction SVWS (store vector with stride) does the same in the reverse direction.

Stride: handling multidimensional arrays - 3

Supporting strides greater than one complicates the memory system. Once non-unit strides are introduced, it becomes possible to request accesses from the same memory bank frequently. When multiple accesses contend for a particular bank, a memory bank conflict occurs, therefore, stalling all but one of the accesses.

A bank conflict and, hence, a stall will occur if

$$\frac{\text{number of banks}}{\text{g.c.d.}(\text{stride}, \text{number of banks})} < \text{bank busy time} .$$

Suppose there are 8 memory banks with a bank busy time of 6 clock cycles and that the memory latency is 12 clock cycles. How long will it take to complete a 64-element vector load with a stride of 1 and with a stride of 32?

stride of 1: $12 + 64 = 76$ clock cycles or 1.2 clock cycles per element

stride of 32: $12 + 1 + 6 \times 63 = 391$ clock cycles or 6.1 clock cycles per element.

Notice that, in the second case, all the accesses are performed in the same memory bank.

Gather-scatter: handling sparse matrices - 1

Sparse matrices are [usually very large] matrices where most of its elements are zero. When operating with sparse matrices, it is important to have some means to describe in a compact form the location of its non zero elements so that operations can be concentrated on them and the computation be made run fast.

Assuming a simplified sparse structure, the code below computes the sum of two sparse matrices A and C, where the location of their corresponding non zero elements, n of them, is described by the index vector K

```
for (i = 0; i < N; i++)
    A[K[i]] = A[K[i]] + C[K[i]];
```

Gather-scatter: handling sparse matrices - 2

The primary mechanism in vector processors for supporting sparse matrices is the implementation of *gather-scatter* operations using index vectors. The goal of such operations is to enable moving back and forth between a compressed representation (zero elements are not included) and a normal representation (zero elements are included).

A *gather* operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets stored in the index vector. The outcome is a dense vector stored in a vector register. The elements are then operated in this dense form and the result is sent back to memory by the *scatter* operation using the same index vector.

Hardware support for these operations is present in nearly all modern vector processors. In VMIPS, for instance, the instructions LVI (load vector indexed or *gather*) and SVI (store vector indexed or *scatter*) play this role.

Gather-scatter: handling sparse matrices - 3

Assuming that the start addresses of the memory locations where matrices A and C are stored in normal representation, are in registers Ra and Rc and that register Rk contains the offsets to the corresponding non zero elements of these matrices, the previous loop may be coded as

LV	V _k , R _k
LVI	V _a , (R _a +V _k)
LVI	V _c , (R _c +V _k)
ADDVV.D	V _a , V _a , V _c
SVI	(R _a +V _k), V _a

This technique allows code with sparse matrices to run in vector mode. However, a programmer's directive is required to tell the compiler it is safe to code the loop in this way.

Although indexed loads and stores can be pipelined, they will run much slower than the corresponding non-indexed ones due to the bank conflicts that may occur throughout the memory system.

Programming vector architectures

An advantage of vector architectures is that compilers can inform programmers at compile time whether a specific code portion will vectorize or not, often providing hints as to why it did not vectorize it. This straightforward execution model allows experts in other domains to learn how to improve performance by revising the code or by advising the compiler when it is OK to assume independent loop operations, such as in the gather-scatter case for data transfers. It is precisely this interaction between the compiler and the programmer that simplifies programming of vector computers.

Nowadays, the main factor that affects the success with which a program runs in vector mode is the structure of the program itself. Do particular loops have true data dependences, or some restructuring is possible to get rid of such dependences, are important questions to be dealt with. The answer to them greatly influences the type of algorithms that are chosen and, to some extent, how they are coded.

SIMD instruction set extensions for multimedia - 1

SIMD multimedia extensions appeared following the observation that many media applications operate on narrower data types than the 32-bit processors were optimized for. Many graphics systems used 8 bits to represent each of the three primary colors plus 8 bits for transparency. Depending on the application, audio samples are usually represented in 8 or 16 bits. By partitioning the carry chains within a 256-bit adder, a processor could perform simultaneous operations on short vectors of 32 8-bit operands, 16 16-bit operands, 8 32-bit operands or 4 64-bit operands. The additional cost of such partitioned adders was small.

Summary of typical SIMD multimedia support for 256-bit wide operations

Source: Computer Architecture: A Quantitative Approach

<i>instruction category</i>	<i>operands</i>
unsigned add / subtract	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
maximum / minimum	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
average	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
shift right / left	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
floating point	16 16-bit, 8 32-bit, 4 64 bit, 2 128-bit

SIMD instruction set extensions for multimedia - 2

Like vector instructions, a SIMD instruction specifies the same operation on data vectors. Unlike vector machines with large register banks, SIMD instructions tend to specify fewer operands and, therefore, use much smaller register banks.

In contrast to vector architectures, which offer an elegant instruction set intended to be the target of a vectorizing compiler, SIMD instructions present three major weaknesses

- they fix the number of data operands in the opcode, which has led to the addition of hundreds of instructions in the MMX, SSE and AVX extensions of the x86 architecture
- they do not offer the more sophisticated addressing modes of vector architectures, namely strided and gather-scatter accesses
- they do not usually offer the mask registers to support elemental conditional execution.

These weaknesses make it harder for the compiler to generate SIMD code and increase the difficulty of programming in assembly language.

SIMD instruction set extensions for multimedia - 3

For the x86 architecture, the MMX instructions, added in 1996, have reused the 64-bit floating point registers, so that the basic instructions could perform 8 8-bit operations or 4 16-bit operations simultaneously. These were joined by parallel MAX and MIN operations, a wide variety of masking and conditional instructions, operations typically found in digital signal processors (DSPs) and *ad hoc* instructions, believed to be useful in important media libraries. MMX also reused the floating point data transfer instructions to access memory.

The Streaming SIMD Extensions (SSE), its successor in 1999, added separate registers that were 128-bit wide, so now instructions could perform simultaneously 16 8-bit operations, 8 16-bit operations or 4 32-bit operations. It also performed parallel floating point arithmetic. Since SSE had separate registers, it needed separate data transfer instructions. Intel soon added double precision SIMD floating point data types via SSE2 in 2001, SSE3 in 2004 and SSE4 in 2007. Instructions with 4 single precision floating point operations or 2 double precision floating point operations increased the peak performance of x86 processors, as long as programmers placed the operands side by side. With each generation, new *ad hoc* instructions whose aim was to accelerate specific multimedia functions, were also introduced.

SIMD instruction set extensions for multimedia - 4

The Advanced Vector Extensions (AVX), added in 2010, doubled again the width of the registers to 256 bits and thereby offered instructions that doubled the number of operations on all narrower data types. AVX includes means to extend the registers width to 512 bits and 1024 bits in future generations of the architecture.

The goal of these extensions has been in general to accelerate carefully written libraries rather than for the compiler to produce them. Recent x86 compilers, however, are trying to generate such a code, particularly for floating point intensive applications.

Given these weaknesses, why are Multimedia SIMD Extensions so popular? There are several reasons for it

- it was not too costly to add them to standard arithmetic and they could be easily implemented
- they require little extra state compared to vector architectures, which is important in context switching
- they do not require a large memory bandwidth.

SIMD instruction set extensions for multimedia - 5

AVX instructions for x86 architecture useful in double precision FP programs

Source: Computer Architecture: A Quantitative Approach

<i>AVX instruction</i>	<i>description</i>
VADDPD	add 4 packed double precision operands
VSUBPD	subtract 4 packed double precision operands
VMULPD	multiply 4 packed double precision operands
VDIVPD	divide 4 packed double precision operands
VFMADDPD	multiply and add 4 packed double precision operands
VFMSUBPD	multiply and subtract 4 packed double precision operands
VCMP __	compare 4 packed double precision operands for EQ, NEQ, GT, LT, GE, LE
VMOVAPD	move aligned 4 packed double precision operands
VBROAADCASTSD	broadcast one double precision operand to 4 locations in a 256-bit register

SIMD instruction set extensions for multimedia - 6

MIPS SIMD code for DAXPY

```
L.D      F0,a  
MOV      F1,F0          // copy a to F1 for SIMD MUL  
MOV      F2,F0          // copy a to F2 for SIMD MUL  
MOV      F3,F0          // copy a to F3 for SIMD MUL  
DADDIU   R4,Rx,512  
Loop:  
L.4D     F4,0(Rx)       // F0 = X[i], ..., F3 = X[i+3]  
MUL.4D   F4,F4,F0       // F4 = a*X[i], ..., F7 = a*X[i+3]  
L.4D     F8,0(Ry)       // F8 = Y[i], ..., F11 = Y[i+3]  
ADD.4D   F8,F8,F4       // F8 = a*X[i]+Y[i], ... ,  
                      // F11 = a*X[i+3]+Y[i+3]  
S.4D     F8,0(Ry)       // Y[i] = F8, ..., Y[i+3] = F11  
DADDIU   Rx,Rx,32  
DADDIU   Ry,Ry,32  
DSUBU    R20,R4,Rx  
BNEZ    R20,Loop
```

Programming multimedia SIMD architectures

Given the *ad hoc* nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries, or by writing directly the code in assembly language.

Recent extensions have become more regular, presenting the compiler with a more reasonable target. By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically. However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
 - Chapter 4: *Data-Level Parallelism in Vector, SIMD and GPU Architectures* – Sections 1 to 5
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
 - Chapter 19: *General Purpose Graphic Processing Units* – Sections 1 to 3